DIGIT BOOKS

Programmation en PERL

Traduction de
Philippe Bruhat,
Kai Carver et
Gérald Sédrati

De Larry Wall, Tom Christiansen et Jon Orwant

A Maymak, GDGN, le philosophe de la Népalawi et à Larabs

Pirate de Bard

Programmation

en Perl

LARRY WALL, TOM CHRISTIANSEN
et JON ORWANT

Programmation en Perl

Traduction de Philippe Bruhat, Kai Carver et Gérald Sédrati

Digit Books

Éditeur de livres numériques

Brest

infos@digitbooks.fr

http://www.digitbooks.fr

DIGIT BOOKS

ISBN : 978-2-8150-0172-4

Prix : 35 €

L'édition originale de ce livre a été publiée aux États-Unis par O'Reilly Media sous le titre *Programming Perl, 3rd edition, ISBN 0-596-00027-*.

Couverture : Yves Buraud

Illustration de « P'tit-Bonhomme » de Jules Verne (1893) par Léon Benett.

http://www.digitbooks.fr/catalogue/9782815001724.html

Table des matières

Préface

À la poursuite du bonheur

Perl est un langage qui vous aide à faire votre travail.

Bien sûr, si votre travail consiste à programmer, vous pouvez le faire avec n'importe quel langage informatique « complet », du moins en théorie. Mais nous savons par expérience que les langages informatiques ne diffèrent pas tant par ce qu'ils rendent *possible* que par ce qu'ils rendent *facile*. D'un côté, les langages soi-disants de « quatrième génération » facilitent certaines tâches, mais en rendent d'autres quasiment impossibles. De l'autre côté, certains langages bien connus, « de qualité industrielle », rendent à peu près tout difficile.

Perl est différent. En résumé, Perl est conçu pour simplifier les tâches faciles, sans rendre les tâches difficiles impossibles.

Et quelles sont ces « tâches faciles » qui devraient le rester ? Celles que l'on rencontre tous les jours, bien sûr. Il faut un langage qui facilite la manipulation des nombres, de texte, des fichiers et des répertoires, des ordinateurs et des réseaux, et surtout des programmes. Il devrait être facile de lancer des programmes externes et de scruter leur sortie à la recherche d'informations intéressantes. Il devrait être facile d'envoyer ces informations vers d'autres programmes qui les manipuleront intelligemment. De même, il devrait être facile de développer, de modifier et de mettre au point vos propres programmes. Et, bien sûr, il devrait être facile de compiler et de lancer ces programmes, de les porter facilement, ceci sur n'importe quel système d'exploitation moderne.

Perl fait tout cela, et plus encore.

Au départ conçu comme un langage pour administrer UNIX, Perl a été porté sur la plupart des systèmes d'exploitation. De fait, Perl est l'un des environnements de programmation les plus portables de nos jours. Pour rendre un programme C ou C++ portable, il faut y inclure ces étranges balises `#ifdef` différentes pour chaque système cible. Pour rendre un programme Java portable, il est nécessaire de comprendre toutes les particularités de chaque nouvelle implémentation de Java. Pour porter un script shell il est né-

cessaire de se souvenir de la syntaxe pour chaque version du système cible, et ceci pour chaque commande, et donc de trouver le plus petit dénominateur commun, qui avec un peu de chances fonctionnera partout. Pour porter un programme écrit en Visual Basic, il faudra alors une définition plus souple du mot portable. :-)

Perl, par on ne sait quelle magie, évite de tels problèmes en retenant les côtés positifs de ces langages. Cette magie a plusieurs sources : l'utilité de ces méthodes, la créative communauté Perl et la floraison du logiciel libre. Mais surtout, cette magie vient de son caractère hybride : Perl est un langage métissé et a toujours vu cette diversité non comme une faiblesse mais comme une force. Alors, si vous avez envie de goûter à la liberté, Perl est fait pour vous.

Perl n'a pas de frontières. À l'origine de l'explosion du langage Perl, il y a le désir des premiers programmeurs système Unix de ne garder que le meilleur de cet « ancien monde ». Pour ceux-ci, Perl est une distillation portable de la culture Unix, et une oasis dans un désert sans issues. Ce qui est satisfaisant, c'est que l'on peut tout aussi bien retraverser de l'autre côté : les designers de sites web sont heureux de pouvoir faire fonctionner leurs scripts Perl sans modification sur le serveur de leur société.

La raison pour laquelle Perl est si populaire parmi les programmeurs système et les développeurs de sites web, c'est tout simplement parce qu'ils l'ont découvert en premier ; mais Perl aspire à une plus large audience. Depuis ses débuts, comme simple langage traducteur d'autres langages, Perl s'est développé comme un langage sophistiqué est très général, accompagné d'un système de développement complet avec débogueur, optimiseur, référenceur croisé, compilateur, éditeur orienté syntaxe et toute une bibliothèque de modules en faisant ainsi un véritable langage de programmation. Mais ce qui différencie vraiment Perl, c'est qu'il a toujours su garder une vision simple des choses qui l'intéressent.

Sa puissance et son accessibilité le rendent utilisable dans tous les domaines de la connaissance, de l'ingénierie spatiale à la biologie moléculaire, des mathématiques à la linguistique, du graphisme au traitement des documents, de la gestion des bases de données à celle des réseaux. Perl est utilisé pour traiter rapidement une grande quantité d'informations, aussi bien des séquences ADN que des pages web. Aussi, l'une des plaisanteries de la communauté Perl, est que le prochain crack boursier sera probablement déclenché par un bogue dans un script Perl. (L'avantage est que les analystes boursiers au chômage pourront toujours se reconvertir.)

Il existe de nombreuses raisons au succès de Perl. Le langage Perl fut l'un des premiers projets de la communauté du logiciel libre. Perl est libre et le sera toujours. Grâce à sa licence très ouverte, vous pouvez l'utiliser où bon vous semble : au travail ou dans des applications commerciales, sans restriction et sans redevance. Et s'il survient une difficulté que la communauté Perl ne peut résoudre, vous disposez en dernier recours du code source. Elle n'est pas du genre à vous faire payer des mises à jour. Elle ne mettra pas la clé sous la porte en laissant votre programme orphelin.

Que Perl soit un langage libre l'a sûrement beaucoup aidé. Mais c'est aussi le cas de la plupart des logiciels libres qui prospèrent. Perl autorise tous ces degrés de liberté car c'est un langage dont la personnalité est éclatée. Il s'agit d'un langage à la fois très simple et très riche. Il a pris de bonnes idées presque partout, et les a implantées dans un cadre facile à utiliser. Pour ceux qui l'aiment sans plus, Perl est le *Practical Extraction and Report Language*. Pour ceux qui ne jurent que par lui, Perl est le *Pathologically Eclectic Rubbish*

Lister. Pour les minimalistes du lot, Perl ressemble à une démonstration de redondance inepte. Pas de problème. Le monde a besoin de quelques réductionnistes (surtout chez les physiciens). Les réductionnistes aiment isoler les choses. Nous, nous essayons de leur donner un sens.

Perl est par bien des aspects un langage simple. Point n'est besoin de connaître de nombreuses incantations mystérieuses pour compiler un programme Perl ; il suffit de l'exécuter comme un script shell. Les types et les structures employés par Perl sont faciles à utiliser et à comprendre. Perl n'impose pas de limitation arbitraire à vos données ; vos chaînes et vos tableaux peuvent croître tant qu'ils veulent (tant qu'il reste de la mémoire) et ils sont conçus pour s'adapter à leur croissance. Au lieu de vous forcer à apprendre de nouvelles syntaxes et de nouveaux concepts, Perl emprunte de nombreuses notions à d'autres langages qui peuvent vous être familiers (comme le C, *awk*, *BASIC*, *Python*, l'anglais et le grec). En fait, pratiquement n'importe quel programmeur peut lire un bout de code Perl bien écrit et se faire une idée de son fonctionnement.

Le plus important est le fait que vous n'avez pas besoin de tout connaître de Perl avant de pouvoir écrire des programmes utiles. On peut apprendre Perl « par le petit bout de la lorgnette ». Vous pouvez programmer comme un enfant qui commence à parler, nous vous promettons de ne pas rire. De nombreuses idées de Perl sont empruntées au langage naturel, et l'une des meilleures est qu'il n'y a aucun problème à utiliser un sous-ensemble du langage tant qu'on arrive à ses fins. N'importe quel niveau de compétence technique est acceptable dans la culture Perl. Il n'existe pas de police du langage. Un script Perl est « correct » s'il vous aide à terminer votre travail avant que le patron ne vous mette dehors.

Bien qu'il soit simple par de nombreux aspects, Perl est également un langage riche dont il y a beaucoup à apprendre. C'est le prix à payer pour faciliter ce qui est difficile. Il vous faudra un certain temps pour en assimiler toutes les possibilités, mais vous serez heureux d'avoir accès à ses remarquables capacités quand vous en aurez besoin. Nous avons fait remarquer plus haut que Perl emprunte de nombreuses aptitudes aux shells et au C, mais il possède aussi un sur-ensemble strict de celles de *sed* et de *awk*. Il existe en fait des traducteurs fournis avec Perl pour transformer les anciens scripts *sed* et *awk* en scripts Perl, et vous pouvez donc voir comment les fonctionnalités avec lesquelles vous pouvez déjà être familiers correspondent à celles de Perl.

En raison de cet héritage, Perl était déjà riche alors même qu'il n'était qu'un langage de réduction de données, conçu pour naviguer dans des fichiers, scruter de grandes quantités de texte, créer et obtenir de données dynamiques et afficher des rapports facilement formatés se basant sur ces données. Mais à un moment donné, Perl a commencé à s'épanouir. Il est alors devenu un langage de manipulation de systèmes de fichiers, de gestion de processus, d'administration de bases de données, de programmation client-serveur, de programmation sécurisée, de gestion d'informations basées sur le Web, et même de programmation orientée objet et fonctionnelle. Ces possibilités n'ont pas été simplement collées après coup ; chacune fonctionne en synergie avec les autres, parce que Perl a été conçu dès le départ comme un langage d'intégration.

Mais Perl peut intégrer plus que ses propres capacités. Perl est conçu pour être extensible de façon modulaire. Perl vous permet de rapidement concevoir, programmer, mettre au point et déployer des applications, mais il vous permet également d'étendre aisément les fonctionnalités de ces applications quand le besoin s'en fait sentir. On peut intégrer Perl à d'autres langages, et on peut intégrer d'autres langages à Perl. Le méca-

nisme d'importation de modules permet d'utiliser ces définitions externes comme s'il s'agissait de fonctionnalités propres à Perl. Les bibliothèques orientées objet conservent cette propriété.

Perl vous assiste en bien d'autres manières. Perl compile d'abord votre programme dans un code intermédiaire, alors qu'un langage interprété classique compile et exécute une commande à la fois. Il effectue diverses optimisations comme tout autre compilateur et donne des résultats instantanés concernant entre autres les erreurs de syntaxe et les problèmes de liaison avec les bibliothèques. Une fois que la partie en amont a validé le programme, elle passe le code intermédiaire à l'interpréteur en aval qui l'exécute (ou, si l'on préfère, à une autre routine de traitement capable d'émettre du code C ou du langage machine). Tout ceci peut paraître compliqué, mais le compilateur et l'interpréteur sont assez efficaces, et la plupart d'entre nous comptent le cycle compiler-lancer-déboguer en secondes. Ce cycle rapide associé à une gestion « douce » des erreurs fait de Perl un langage idéal pour le prototypage. Au fur et à mesure que le programme se développe, on peut resserrer les boulons et programmer avec moins de flair mais plus de discipline. Perl vous y aide également, quand vous le lui demandez gentiment.

Perl vous permet aussi d'écrire des programmes plus sûrs. En plus des interfaces de sécurité implémentées par les autres langages, un mécanisme de traçage automatique des source d'insécurité, bloque toute opération dangereuse. Enfin, Perl permet de définir des compartiments spécialement protégés dans lesquels on peut exécuter en toute sécurité du code Perl d'origine douteuse, en masquant les opérations dangereuses.

Mais, paradoxalement, la meilleure aide que Perl puisse vous apporter n'a presque rien à voir avec Perl, mais tout à voir avec les gens qui l'utilisent. Les Perlistes comptent parmi les gens les plus utiles sur terre, vraiment. S'il existe un aspect religieux au mouvement Perl, c'est celui-là. Larry a voulu que la communauté Perl fonctionne comme un bout de paradis, et il semble que son souhait se soit réalisé jusqu'à présent. Puissiez-vous faire de votre mieux pour que cela continue.

Que vous appreniez Perl parce que vous voulez sauver le monde, ou simplement parce que vous êtes curieux, ou encore parce que votre patron vous a dit de vous y mettre, ce livre vous mènera de ses bases les plus élémentaires vers ses labyrinthes les plus obscurs. Et même si nous n'avons pas l'intention de vous apprendre à programmer, le lecteur attentif en retirera un peu de l'art, et de la science, de la programmation. Nous vous encouragerons à développer les trois grandes vertus du programmeur : la *paresse*, l'*impatience* et l'*orgueil*. Tout au long de ce parcours, nous espérons que vous trouverez ce livre assez amusant par certains côtés (et d'un humour ravageur par d'autres). Et si rien de tout cela ne vous suffit, souvenez-vous que l'apprentissage de Perl enrichira votre *curriculum vitae*. Continuez donc votre lecture.

Ce qui est nouveau dans cette édition

Presque tout...

Sauf des morceaux choisis de la précédente édition (et ils étaient nombreux), nous avons entièrement révisé la présente édition pour plusieurs raisons. Premièrement nous avons voulu rendre le livre accessible à un plus large public. Il y a très peu de prérequis. Nous avons volontairement choisi un style vivant pour garder éveillés ceux qui en auraient beaucoup (des prérequis).

Deuxièmement, nous avons présenté les derniers développements de Perl. Même si ceux-ci restent à l'état d'expérience de laboratoire, le noyau central est solidement planté. La foulée à adopter pour suivre le développement de certaines extensions, peut être torride. C'est pourquoi nous vous indiquerons d'aller voir plutôt la documentation en ligne, pour préciser certains aspects, car Perl est un langage de mécanicien et on peut confondre un boulon de 8 avec un boulon de 10.

Enfin, nous avons adapté cette édition pour en faciliter la lecture, en séparant les chapitres en parties introduisant chacune un concept, dans le but de mieux saisir l'ensemble. Cette nouvelle édition s'organise ainsi :

Partie I, Survol de Perl

Le démarrage est toujours difficile. Cette partie présente les idées fondamentales du langage sur un ton informel et est à lire au coin du feu. Il s'agit plus d'un marchepied que d'un didacticiel complet, et elle peut ne pas correspondre aux besoins de tout le monde. Voir la section *Documentation papier* pour les livres qui pourraient mieux convenir.

Partie II, Séance de dissection

Cette partie est un développement à bâtons rompus et en profondeur des mécanismes essentiels du langage à chaque niveau d'abstraction, depuis les types de données, les variables, les expressions régulières jusqu'aux routines, modules et objets. Vous y acquerrez une bonne mesure du fonctionnement de ce langage ainsi que quelques bonnes méthodes de conception de logiciels. (Et si vous n'avez jamais utilisé un langage permettant la reconnaissance des modèles, vous allez avoir un traitement de faveur.)

Partie III, Technique Perl

Il est possible de faire beaucoup de choses avec Perl et ce chapitre vous emmènera beaucoup plus loin. Ici, vous apprendrez comment répondre à tous les défis lancés par votre machine, depuis le traitement des caractères Unicode, la communication entre processus, le multithreading, la compilation, le débogage et l'optimisation de Perl, jusqu'à l'écriture de vos propres extensions en C ou C++ ou l'interfaçage de toute API. Perl est tout à fait adapté pour communiquer avec votre PC, et si besoin avec les autres PC sur Internet.

Partie IV, Culture Perl

Toute culture ayant un langage, la communauté Perl a vite compris qu'elle devait se forger une culture. Nous verrons dans cette partie la programmation Perl dans la vie de tous les jours. Nous verrons comment reconnaître les bons des méchants, et aussi comment nous améliorer pour produire des programmes réutilisables par tout le monde.

Partie V, Références

Nous avons regroupé dans cette partie tous les termes importants de chaque chapitre, depuis les variables et fonctions spéciales, aux modules et pragmas standards. Le glossaire sera particulièrement utile à ceux qui ne sont pas familiers avec le jargon informatique. Par exemple, si vous ignorez le sens du terme « pragma », vous pouvez le découvrir dès maintenant.

Distribution standard

Aujourd'hui, la grande majorité des systèmes d'exploitation comprennent une distribution de Perl en standard. Au moment où ces lignes sont écrites, les systèmes AIX, BeOS, BSDI, Debian, DG/X, DYNIX/ptx, FreeBSD, IRIX, LynxOS, Mac OS X, OpenBSD, RedHat, SINIX, Slackware, Solaris, SuSE, Mandrake et Tru64, sont tous distribués avec une version de Perl en standard. Certaines sociétés fournissent Perl sur un CD de distribution libre ou bien par l'intermédiaire de leur service client. Des sociétés tierces comme ActiveState, proposent des distributions pour différents systèmes y compris Microsoft.

Même si votre distributeur livre Perl en standard, vous aurez probablement envie de compiler et d'installer Perl vous-même. Vous serez ainsi certain d'avoir la dernière version, et de pouvoir installer où bon vous semble la documentation ainsi que les bibliothèques. Vous pourrez aussi choisir parmi les différentes options de compilation tels que le multithreading, la gestion des grands fichiers, ou bien les options de débogage de bas niveau disponible par le sélecteur `-D`. (Le débogage niveau utilisateur est toujours supporté.)

La façon la plus simple d'obtenir les sources de Perl est d'aller à *www.perl.com*, où vous trouverez les informations les plus importantes, ainsi que des liens vers des distributions binaires spécifiques aux plates-formes sans compilateurs C.

Vous pouvez aussi vous rendre directement sur le CPAN (*Comprehensive Perl Archive Network*, décrit au chapitre 22, *CPAN*, à l'adresse *http://www.Perl.com/CPAN* ou bien *http://www.cpan.org*. Si ces adresses sont trop lentes (c'est possible car elles sont *très* populaires), vous pouvez choisir ainsi un miroir plus proche de chez vous. Les adresses url suivantes ont été choisies parmi les centaines de miroirs CPAN dans le monde :

http://www.funet.fi/pub/languages/Perl/CPAN/
ftp://ftp.funet.fi/pub/languages/Perl/CPAN/
ftp://ftp.cs.colorado.edu/pub/Perl/CPAN/
ftp://ftp.cise.ufl.edu/pub/Perl/CPAN/
ftp://ftp.Perl.org/pub/Perl/CPAN/
http://www.Perl.com/CPAN-local
http://www.cpan.org/
http://www.Perl.org/CPAN/
http://www.cs.uu.nl/mirror/CPAN/
http://CPAN.pacific.net.hk/

Les deux premières de la liste, sur le site *funet.fi*, pointent sur le répertoire maître de tous les sites CPAN. Le fichier *MIRRORED.BY* contient la liste de tous les sites pour laisser le choix du site miroir ainsi que du protocole pour le téléchargement (FTP ou HTTP), ce qui est utile lorsque votre réseau est derrière un firewall. Le site multiplexeur *http://www.Perl.com/CPAN* vous aide à faire cette sélection.

Une fois que vous avez récupéré et décompacté le code source dans un répertoire, vous devez lire les fichiers *README* et *INSTALL* pour comprendre comment construire l'exécutable Perl. Il peut aussi y avoir un fichier *INSTALL.platform* où *platform* désigne votre plate-forme système.

Si votre *platform* est un système Unix ou similaire, alors les commandes pour charger, configurer, construire et installer Perl seront celles qui suivent. Premièrement vous devez utiliser une commande pour charger le code source. Avec *ftp* :

```
% ftp ftp://ftp.funet.fi/pub/languages/Perl/CPAN/src/latest.tar.gz
```

(vous pouvez substituer un miroir CPAN plus proche. Si vous habitez en Finlande *c'est* votre miroir CPAN le plus proche.) Si vous ne pouvez utiliser *ftp*, alors télécharger sur le Web avec un navigateur ou bien la ligne de commande suivante :

```
% wget http://www.funet.fi/pub/languages/Perl/CPAN/src/latest.tar.gz
```

Ensuite, il faut décompacter, configurer, construire et installer :

```
% tar zxf latest.tar.gz          ou gunzip, et ensuite tar xf.
% cd Perl-5.6.0                  ou 5.* pour toute version.
% sh Configure -des              Réponses par défaut.
% make test && make install      En mode super-utilisateur.
```

Ces commandes s'effectuent dans un environnement de développement C conventionnel, et nécessitent donc un compilateur C. Pour connaître les différentes mises à jour et versions de Perl pour chaque plate-forme, si vous avez besoin d'une version standard ou d'un portage spécial, explorez le répertoire CPAN *ports*. Des liens sont disponibles pour télécharger ses portages spéciaux ou pour les systèmes ne disposant pas de compilateurs C.

Documentation en ligne

L'importante documentation en ligne de Perl fait partie de la distribution standard. (Voir la section suivante pour la documentation papier.) Chaque module CPAN est accompagné de sa documentation.

Quand nous faisons référence au « manuel Perl », nous parlons de l'ensemble des pages en ligne du manuel Perl, sur votre PC. Le terme *manuel* est purement une convention indiquant un fichier contenant de la documentation (nul besoin d'un programme *man* pour la lire. Elles peuvent même être au format HTML sur les systèmes différents d'Unix.

Le manuel Perl en ligne a été divisé en différentes sections, pour pouvoir facilement trouver ce que l'on cherche sans avoir à parcourir des centaines de pages de texte. Le manuel Perl de base se nomme *perl*, et on y accède avec la commande *man perl*[1]. Ce manuel vous orientera vers d'autres plus spécifiques. Par exemple *man perlre* affichera le manuel des expressions régulières de Perl. La commande *perldoc* fonctionne souvent sur les systèmes ou la commande *man* ne fonctionne pas. Sur Mac, il faut utiliser le programme *Shuck*. Votre distribution peut aussi fournir le manuel Perl au format HTML, ou bien le format d'aide natif du système hôte.

1. Si vous obtenez une page illisible c'est probablement que votre version de manuel est trop ancienne. Vérifiez votre variable d'environnement `MANPATH`. (Essayez la commande *perldoc perl* pour configurer `MANPATH` sur la valeur retournée par la commande *perl -V:man.dir*.)

Consultation du manuel standard

À l'origine du langage, en 1987, le manuel *perl* était un document concis d'environ vingt-quatre pages dactylographiées. La section concernant les expressions régulières n'occupait que deux paragraphes. (avec la connaissance de la commande *egrep* cela pouvait suffire.) D'une certaine façon tout a changé depuis : de la documentation standard, les différents utilitaires, l'information pour le portage entre les différentes plates-formes et les centaines de modules standard, il y a maintenant plus de 1500 pages dactylographiés de documentation réparties entre plusieurs manuels. (Sans compter les modules CPAN que vous installerez.)

D'un autre côté rien n'a changé : le manuel de base *perl* existe toujours, et c'est toujours le point de départ si vous êtes perdu. La différence est que lorsque vous y êtes, vous ne pouvez vous y arrêter. La documentation Perl n'est plus une petite épicerie mais un grand centre commercial avec des centaines de magasins où il est nécessaire de savoir où l'on est pour savoir où l'on va. Une fois que l'on connaît le plan, il est facile de s'orienter.

Voici quelques points de repère que vous pourrez rencontrer le :

Manuel	**Sujet**
perl	Les différents manuels Perl disponibles.
perldata	Les types de données.
perlsyn	La syntaxe Perl.
perlop	Précédence des opérateurs.
perlre	Les expressions régulières.
perlvar	Les variables prédéfinies.
perlsub	Les routines Perl.
perlfunc	Les fonctions prédéfinies.
perlmod	Mise en œuvre des modules.
perlref	Les références.
perlobj	Les objets.
perlipc	La communication interprocessus.
perlrun	L'utilisation des commandes Perl avec sélecteurs.
perldebug	Le débogage.
perldiag	Les messages de diagnostic.

C'est juste un aperçu, mais ce sont les entrées essentielles : si vous voulez connaître un opérateur, *perlop* vous donnera la réponse, et vous cherchez une chose sur les variables prédéfinies, regardez dans *perlvar*. Pour comprendre un message d'erreur, allez dans *perldiag*. Et ainsi de suite.

Les FAQ (questions revenant fréquemment) sont une partie standard du manuel Perl. Elles sont divisées en neuf manuels différents :

Manuel	Sujet
perlfaq1	Les questions générales sur Perl.
perlfaq2	La récupération et l'apprentissage de Perl.
perlfaq3	Les outils de programmation.
perlfaq4	Les manipulation des données.
perlfaq5	Les fichiers et formats.
perlfaq6	Les expressions régulières.
perlfaq7	Les questions d'ordre général sur le langage.
perlfaq8	L'interaction avec le système.
perlfaq9	La connexion au réseau.

Certains manuels rassemblent des notes spécifiques à certaines plates-formes :

Manuel	Sujet
perlamiga	Le portage sur Amiga.
perlcygwin	Le portage sur Cygwin.
perldos	Le portage sur MS DOS.
perlhpux	Le portage HP UX.
perlmachten	Le portage sur Power MachTen.
perlos2	Le portage sur OS/2.
perlos390	Le portage sur OS/390.
perlvms	Le portage sur DEC VMS.
perlwin32	Le portage sur MS-Windows.

(Voir aussi le chapitre 25, *Portabilité*, et le répertoire des *portages* du CPAN déjà décrit pour toute information relative à ce sujet.)

Recherche dans le manuel

N'essayez pas de lire les 1500 pages du manuel, autant chercher une aiguille dans une botte de foin. Il y a un vieux dicton qui dit qu'on ne peut pas grapper des arbres morts.[2] Chacun des manuels Perl, dispose de son propre outil de recherche et de visualisation. Il est possible de chercher une page du manuel en indiquant la commande constituée du nom du manuel suivi d'une expression régulière comme paramètre (voir le chapitre 5, *Recherche de motif*) :

```
% perlop comma
% perlfunc split
% perlvar ARGV
% perldiag 'assigned to typeglob'
```

2. N'oubliez pas le glossaire.

Il est possible d'étendre la recherche sur un manuel entier, par exemple pour chercher dans les FAQ, utilisez la commande *perlfaq* (qui est aussi un manuel) :

```
% perlfaq round
```

La commande *perltoc* (aussi un manuel) cherche dans le sommaire général de tous les manuels :

```
% perltoc typeglob
Perl5005delta: Undefined value assigned to typeglob
Perldata: Typeglobs and Filehandles
Perldiag: Undefined value assigned to typeglob
```

Ou alors, pour trouver une chaîne dans toutes les ressources en ligne du manuel, incluant les en-têtes, les descriptions et exemples, utilisez la commande *perlhelp* :

```
% perlhelp CORE::GLOBAL
```

Voir le manuel *perldoc* pour les détails.

Les autres manuels

Quand nous parlons de documentation non-Perl, comme par exemple dans *getitimer*(2), nous faisons référence aux manuels *getitimer* dans la section 2 du manuel *Unix Programmer's Manual*.[3] Les pages du manuel pour les appels système tel *getitimer* peuvent ne pas être présentes sur les systèmes non Unix, mais ce n'est pas grave car vous ne pouvez pas les utiliser dans ce cas. Si vous en avez vraiment besoin, vous les trouverez facilement publiés sur le Web — une recherche rapide de "+crypt(3) +manual" avec AltaVista vous permettra d'en obtenir plusieurs versions.

Même si les principaux manuels Perl sont installés dans la section 1 du répertoire standard de *man*, nous omettrons l'indice (1) quand nous y ferons référence dans ce livre. Vous les reconnaîtrez néanmoins sans difficulté car ils sont tous de la forme « perl*mumble* ».

Documentation papier

Voici quelques titres de la littérature Perl que nous vous recommandons de :

- *Perl 5 Précis et concis*, 2e ed., de Johan Vromans (O'Reilly, 2000). Ce manuel au format poche est une référence pratique et rapide.
- *Perl en action*, par Tom Christiansen et Nathan Torkington (O'Reilly, 1998). Un livre pour accompagner celui-ci.
- *Elements of Programming with Perl*, Andrew L. Johnson (Manning, 1999). Pour commencer à partir de 0 en utilisant Perl, pour les non programmeurs.

3. La section 2 contient les appels directs des fonctions du système. (et sont appelées « appels systèmes », à ne pas confondre avec les appels à la fonction `system`, ce qui n'a rien à voir. Cependant, les systèmes diffèrent quant à l'implémentation de ces « appels systèmes » entre ceux qui utilisent la fonction `system` et ceux qui font appel aux fonctions de la bibliothèque C, ce qui fait que l'on peut trouver *getitimer*(2) dans la section 3 (Sous Programmes) au lieu de la section 2 (Appels Systèmes).

- *Introduction à Perl*, 2e ed., par Randal Schwartz et Tom Christiansen (O'Reilly, 1997). Pour les administrateurs et programmeurs du système Unix : l'utilisation de 30 % du langage dans 70 % des cas. La version sur le système Microsoft revue par Erik Olson s'appelle *LearningPerl for Win32 Systems*.
- *Perl: The Programmer's Companion*, par Nigel Chapman (Wiley, 1997). Ce livre très concis est destiné aux ingénieurs et programmeurs en informatique et considère Perl indépendamment de la plate-forme, d'une manière complète.
- *Maîtrise des expressions régulières*, par Jeffrey Friedl (O'Reilly, 1997). Bien qu'il n'inclut pas les derniers ajouts sur les expressions régulières, il reste une référence indispensable pour quiconque cherche à mieux comprendre comment celles-ci fonctionnent.
- *Object Oriented Perl*, par Damian Conway (Manning, 1999). Pour les débutants aussi bien que les programmeurs objet avancés, ce livre étonnant développe des techniques simples et exotiques dans le but d'écrire des systèmes objets puissants en Perl.
- *Mastering Algorithms with Perl*, par Jon Orwant, Jarkko Hietaniemi et John Macdonald (O'Reilly, 1999). Toutes les techniques utiles d'un cours sur les algorithmes, sans les preuves ardues. Ce livre couvre les algorithmes fondamentaux dans le domaine des graphes, du traitement de texte, des ensembles et plein de bonnes choses encore.
- *Writing Apache Modules with Perl and C*, par Lincoln Stein et Doug MacEachern (O'Reilly, 1999). Ce guide de la programmation web enseigne comment étendre les capacités du serveur Apache en utilisant le module turbo `mod_Perl` pour l'exécution rapide des scripts CGI ainsi que l' API de ce serveur accessible à Perl.
- *The Perl Journal*, édité par Jon Orwant. Ce magazine trimestriel créé par des programmeurs pour des programmeurs, détaille des trucs et astuces, parle des dernières nouvelles et de plein d'autres choses encore.

Il existe une pléthore de livres et de publications sur le sujet et nous avons sûrement oublié des références (nous avons négligé sans pitié les mauvaises).

En plus de ceux que nous venons de citer, nous vous recommandons les livres suivants. Ils n'ont pas un rapport direct avec le langage mais sont très pratiques comme référence, pour la consultation et l'inspiration.

- *The Art of Computer Programming*, par Donald Knuth, vol. 1, *Fundamental Algorithms* ; vol. 2, *Seminumerical Algorithms* ; et vol. 3, *Sorting and Searching* (Addison-Wesley, 1998).
- *Introduction to Algorithms*, par Cormen, Leiserson et Rivest (MIT Press and McGraw-Hill, 1990).
- *Algorithms in C: Fundamental Data Structures, Sorting, Searching*, 3e ed., par Robert Sedgewick (Addison-Wesley, 1997).
- *The Elements of Programming Style*, par Kernighan et Plauger (Prentice-Hall, 1988).
- *The Unix Programming Environment*, par Kernighan et Pike (Prentice-Hall, 1984).
- *POSIX Programmer's Guide*, par Donald Lewine (O'Reilly, 1991).
- *Advanced Programming in the UNIX Environment*, par W. Richard Stevens (Addison-Wesley, 1992).

- *TCP/IP Illustrated*, vols. 1–3, par W. Richard Stevens, (Addison-Wesley, 1994–1996).
- *Le seigneur des anneaux* de J. R. R. Tolkien

Ressources supplémentaires

L'Internet est une merveilleuse invention et nous découvrons tous les jours comment utiliser tout son potentiel. (Certains préféreront « découvrir » l'Internet à la manière dont Tolkien découvre la « Terre du Milieu. »)

Perl sur le web

Allez voir la page d'accueil de Perl à l'adresse *http://www.Perl.com/*. On y parle des nouveautés dans le monde Perl, vous y trouvez du code source et différents reportages, des articles thématiques, de la documentation, les dates des conférences et bien d'autres choses encore.

Allez aussi voir la page des Mongueurs à l'adresse *http://www.Perl.org* et vous pourrez constater que Perl est comme une herbe sauvage qui pousse partout dans le monde sauf au pôle sud. Des groupes locaux de mongueurs se réunissent régulièrement avec lesquels vous pouvez échanger vos points de vue entre hackers.

Groupe de news Usenet

Les groupes de news Perl sont une source d'informations remarquable, bien que parfois confus. *comp.lang.Perl.announce* est un groupe modéré d'un faible trafic sur les annonces concernant Perl. Il s'agit fréquemment de nouvelles versions, de corrections de bogues, de nouvelles extensions et de nouveaux modules et de FAQ (*Frequently Asked Questions*, questions fréquemment posées[4]).

Le groupe *comp.lang.Perl.misc* traite aussi bien des problèmes techniques que de la philosophie de Perl, des jeux en Perl et de la poésie en Perl. Tout comme Perl lui-même, *comp.lang.Perl.misc* se veut utile, et aucune question n'est trop stupide.[5]

Le groupe *comp.lang.Perl.tk* discute de l'utilisation du fameux module Perl/Tk. Le groupe *comp.lang.Perl.modules* concerne le développement et l'utilisation des modules Perl qui sont le meilleur moyen pour réutiliser le code. Il y a d'autres groupes du type *comp.lang.Perl.un_groupe* qui n'existent pas au moment où nous écrivons ces lignes, alors n'hésitez pas à naviguer.

Si vous n'utilisez pas un lecteur de news classique pour accéder à Usenet, mais plutôt un navigateur web, placez devant le nom du groupe l'en-tête `"news:"` pour accéder à l'un de ceux que vous cherchez. (Cela fonctionne seulement si vous avez un serveur de news) Vous pouvez aussi les chercher à l'aide de AltaVista ou Deja en précisant « `*Perl*` » comme nom du groupe d'utilisateurs.

Un dernier groupe d'utilisateurs qu'il vous sera utile de consulter, au moins si vous êtes

4. Ou Foire Aux Questions.
5. Il est évident que certaines questions sont trop stupides... surtout celles dont la réponse se trouve dans la FAQ.

programmeur CGI, est le groupe *comp.infosystems.www.authoring.cgi*. Même si ce n'est pas à strictement parler un groupe d'utilisateurs Perl, la plupart des programmes d'exemples sont écrits en Perl, à moins que vous n'utilisez le module Apache `mod_Perl`, auquel cas vous pouvez consulter *comp.infosystems.www.servers.unix*.

Rapports de bogues

Au cas — improbable — où vous trouviez un bogue propre à Perl et non à votre programme, essayez de le réduire à un cas de test minimal avant de le rapporter *via* le programme *Perlbug* qui est fourni avec Perl. Voir le lien *http://bugs.Perl.org* pour plus d'informations.

De l'aide en français

Si vous voulez en savoir plus sur Perl dans un cadre sympathique, il existe des groupes d'utilisateurs francophones de Perl. Voici d'ailleurs un message du fondateur d'un de ces groupes :

> J'espère que ce livre apportera satisfaction aux lecteurs francophones qui désirent (mieux) programmer en Perl. Ce n'est pas toujours possible pour quelqu'un dont l'anglais n'est pas la langue maternelle de comprendre toutes les subtilités des points techniques, ainsi que les plaisanteries et les jeux de mots qui sont légion dans la documentation anglaise. Je crois savoir, pour avoir écouté (et parfois répondu) à leurs questions, que les traducteurs de cet ouvrage n'ont pas failli sur les explications des subtilités techniques et aussi ont-ils fait de leur mieux pour traduire les plaisanteries.
>
> Toujours est-il que parfois, même en lisant ce livre de manière assidue, on n'arrive pas à comprendre comment faire marcher un module de CPAN, s'il existe un module CPAN qui fait ce que l'on souhaite faire, ou s'il existe de meilleurs moyens pour effectuer une opération. À ce moment-là, on a envie de discuter avec d'autres personnes. Par la plus forte coïncidence, il existe des groupes d'utilisateurs de Perl. J'ai fondé le premier groupe français il y a environ trois ans, à Paris.
>
> Ce sont des *mongers*, qu'on pourrait traduire par *marchands* ; mais que je préfère écrire *mongueurs* (*mongueuses* au féminin), pour rester plus fidèle à la prononciation anglaise. Nous avons un site web, ainsi qu'une liste de diffusion qui, en octobre 2001, compte environ 150 abonné(e)s. Nous avons également fondé une association, dont le but est de promouvoir l'utilisation de Perl en France, en soutenant la création de nouveaux groupes d'utilisateurs (à ce jour, Toulouse et Marseille nous ont rejoint) et en organisant des conférences ou des cours consacrés à Perl.
>
> C'est pourquoi je vous invite à venir nous rencontrer, sur notre site web (*http://www.mongueurs.net/*), notre liste ou à nos réunions (rendez visite au groupe local près de chez vous ou mieux, fondez-en un !), pour découvrir le côté humain qui se cache derrière le langage qui nous a tant apporté et qui continue à nous fasciner.
>
> David Landgren
> Président de l'association « Les Mongueurs de Perl »
> Octobre 2001

Conventions typographiques

Nous avons consacré une section propre pour décrire certaines conventions telles les conventions de codage décrites dans *Programmation stylée* au chapitre 24, *Techniques couramment employées*. Les conventions lexicales sont indiquées dans le glossaire (notre lexique).

Dans cette édition nous avons utilisé les conventions typographiques suivantes :

Italique

Elle est utilisée pour les URL, manuels, noms de fichier et programmes. Les termes nouveaux sont en italique la première fois qu'ils apparaissent dans le texte. La plupart sont repris plus précisément dans le glossaire.

`Chasse constante`

Elle est utilisée dans les exemples et pour indiquer du code inséré dans du texte normal. Les valeurs sont écrites en `chasse constante` entre guillemets (« »), qui bien sûr ne font pas partie de la valeur.

`Chasse constante en gras`

Elle est utilisée pour les options de commande, ce qui permet de distinguer entre l'option avertissement **`-w`** et l'opérateur `-w` pour le test d'un fichier. Elle est aussi utilisée pour le texte que vous entrez littéralement.

`Chasse constante en italique`

Elle est utilisée pour des termes génériques auxquels vous devez substituer vos valeurs personnelles.

Nous listons beaucoup d'exemples qui sont pour la plupart des bouts de code qui peuvent s'intégrer dans une application plus large. Certains exemples sont des programmes complets et que vous pouvez reconnaître avec l'en-tête de début de ligne `#!`. Les programmes les plus importants commencent par :

```
#!/usr/bin/Perl
```

D'autres programmes doivent aussi être entrés en ligne de commande. Nous avons utilisé le symbole `%` pour indiquer un prompteur shell :

```
% Perl -e ‘print "Hello, world.\n"‘
Hello, world.
```

Ce style représentatif d'une ligne de commande Unix où les simples guillemets représentent la forme la plus « quotée ». La quotation, ainsi que les caractères généraux (*wildcards*), varient d'un système à l'autre. Par exemple beaucoup d'interpréteurs de ligne de commande sous MS DOS et VMS nécessitent la double quotation en lieu et place des simples guillemets lorsqu'il est nécessaire de grouper les arguments avec des espaces ou des caractères généraux de remplacement.

Remerciements

Nous voulons remercier ici publiquement nos relecteurs pour leur patience dans l'édition de cet ouvrage : Todd Miller, Sharon Hopkins Rauenzahn, Rich Rauenzahn, Paul

Marquess, Paul Grassie, Nathan Torkington, Johan Vromans, Jeff Haemer, Gurusamy Sarathy, Gloria Wall, Dan Sugalski et Abigail.

Nous voulons remercier tout particulièrement Tim O'Reilly (et ses associés) d'encourager les auteurs à faire un effort spécial pour rendre leurs livres agréables à lire.

I

Survol de Perl

1

Vue d'ensemble

Démarrage

Nous pensons que Perl est un langage facile à apprendre et à utiliser, et nous espérons vous en convaincre. L'un des aspects les plus agréables de Perl est que l'on n'a pas besoin d'en écrire beaucoup avant d'exprimer vraiment sa pensée. Dans de nombreux langages de programmation, il faut déclarer les types, les variables et les sous-programmes que l'on va utiliser avant d'écrire la première instruction de code exécutable. C'est une bonne approche pour des problèmes complexes demandant des structures de données complexes ; mais pour pour beaucoup de problèmes simples, que l'on rencontre tous les jours, il vaut mieux un langage de programmation dans lequel il suffit d'écrire :

```
print "Salut tout le monde!\n";
```

et où le programme ne fait que cela.

Perl est de ce genre de langage. En fait, cet exemple est un programme complet, et si on le donne à l'interpréteur Perl, il affiche « `Salut tout le monde !` » à l'écran.[1] (Le caractère `\n` produit un saut de ligne en sortie.)

Et voilà. Il n'y a pas non plus grand-chose à écrire *après* ce que l'on veut dire, d'ailleurs. À l'inverse de nombreux langages, Perl estime que s'arrêter sans prévenir à la fin du programme n'est qu'une façon normale d'en sortir. On *peut* bien sûr appeler explicitement la fonction `exit` si on le souhaite, de même que l'on *peut* déclarer certaines variables ou certains sous-programmes, ou même se *forcer* à déclarer toutes les variables et tous les sous-programmes. À vous de voir. En Perl, vous êtes libre de faire Ce Qu'il Faut, quelle qu'en soit votre définition.

Il existe d'autres raisons à la facilité d'emploi de Perl, mais il serait inutile de toutes les énumérer ici... car c'est l'objet de ce livre. Le diable est dans les détails, comme disent les Anglo-saxons, mais Perl essaie aussi de vous sortir de l'enfer où vous pouvez vous retrouver plongé. C'est à tous les niveaux que Perl essaye de vous faire arriver à vos fins avec

1. Ou un script, ou une application, ou un exécutable, ou un machin. Ce qu'on veut.

le minimum d'ennuis et le maximum de plaisir. C'est pourquoi tant de programmeurs Perl affichent un sourire béat.

Ce chapitre constitue un survol de Perl, et nous n'essayons pas de le présenter à la partie rationnelle de votre cerveau. Nous n'essayons pas plus d'être complets ou logiques. Ce sera l'objet du prochain chapitre. Les Vulcains, les androïdes ainsi que les humains de cette espèce, peuvent aller directement au chapitre 29, *Fonctions*, pour avoir toute l'information.

Ce livre présente Perl à l'*autre* partie de votre cerveau, que vous l'appeliez associative, artistique, passionnée, ou simplement spongieuse. Dans ce but, nous présenterons divers aspects de Perl qui vous en donneront une image aussi claire que celle d'un éléphant pour un aveugle. Allons, nous allons essayer de faire mieux. Il s'agit ici d'un chameau, après tout. Espérons qu'au moins un de ces aspects du langage vous mettra en selle.

Langages naturels et artificiels

Les langages ont d'abord été inventés par les humains, pour les humains. Ce fait a parfois été oublié dans les annales de l'informatique.[2] Perl ayant été conçu (pour ainsi dire) par un linguiste occasionnel, il a été inventé pour fonctionner aussi aisément que le langage naturel. Cela recouvre plusieurs notions, puisque le langage naturel fonctionne simultanément à plusieurs niveaux. Nous pourrions énumérer ici nombre de ces principes linguistiques, mais le plus important est que les choses simples doivent le rester et que les choses complexes ne doivent pas être impossibles. Cela peut sembler évident, mais de nombreux langages informatiques ne remplissent pas ces deux conditions.

Les langages naturels permettent les deux parce que les gens essayent continuellement d'exprimer des choses simples et des choses complexes, ce qui fait que le langage évolue pour gérer les deux. Perl a été conçu avant tout pour évoluer, et c'est ce qui s'est produit. De nombreuses personnes ont contribué à l'évolution de Perl au cours des ans. Nous plaisantons souvent en disant que le chameau est un cheval conçu par un comité, mais si l'on y pense, le chameau est parfaitement adapté à la vie dans le désert. Il a évolué pour devenir relativement autosuffisant. (En revanche, l'évolution de l'odeur du chameau est discutable. Cela vaut pour Perl.)

Quand quelqu'un prononce le mot « linguistique », on pense en général soit à des mots, soit à des phrases. Mais les mots et les phrases ne sont que deux manières pratiques de « morceler » la parole. Ils peuvent tous deux être scindés en unités sémantiques plus petites, ou combinés en unités plus grandes. Et la signification de chaque unité dépend largement du contexte syntaxique, sémantique et pragmatique dans lequel se trouve l'unité. Le langage naturel comporte des mots de diverses sortes, des noms, des verbes, etc. Si je dis « pile » tout seul, il y a de fortes chances pour que vous pensiez qu'il s'agisse d'un nom. Mais je peux l'employer autrement : comme un verbe, comme un adverbe, etc., selon le contexte. Si je pile devant une pile de piles à minuit pile, j'ai intérêt à disposer d'une pile.[3]

2. Plus précisément, ce fait a parfois dû être rappelé.

3. Vous trouvez sans doute que ces considérations linguistiques manquent de sex-appeal. Le but est simplement de vous montrer en quoi Perl est différent d'un langage informatique classique.

De même, Perl évalue les mots de façon différente dans des contextes différents. Nous le verrons plus tard. Souvenez-vous que Perl essaie de comprendre ce que vous dites ; il est à l'écoute. Perl essaie vraiment d'aller jusqu'au bout. Dites ce que vous pensez, et en général, Perl le saisira (à moins d'écrire des choses absurdes, bien sûr ; l'analyseur de Perl comprend le Perl bien mieux que le Français ou le Swahili).

Mais revenons aux noms. Un nom peut désigner un objet précis, ou il peut nommer une classe d'objets générique. La plupart des langages de programmation font cette distinction, sauf que nous appelons *valeur* une chose précise et *variable* une chose générique. Une valeur existe quelque part, peu importe où, mais une variable est associée à une ou plusieurs valeurs durant son existence. Ce qui interprète la variable doit garder la trace de cette association. Cet interpréteur peut se trouver dans votre cerveau, ou dans votre ordinateur.

Syntaxe d'une variable

Une variable n'est qu'un endroit pratique pour conserver quelque chose, un endroit qui a un nom, ce qui fait que vous pouvez retrouver vos petits quand vous y revenez plus tard. Tout comme dans la vie réelle, il existe de nombreux endroits pour ranger des choses, certains privés et certains publics. Certains lieux sont temporaires et d'autres sont permanents. Les informaticiens adorent parler de « portée » des variables, mais c'est à peu près tout ce qu'ils entendent par là. Perl connaît de nombreux moyens de traiter les problèmes de portée, ce que vous serez ravis d'apprendre le moment venu. (Regardez l'emploi des adjectifs `local`, `my` et `our` dans le chapitre 29, si vous êtes curieux, ou bien regardez *Déclarations avec portée* dans le chapitre 4, *Instructions et déclarations*.)

Mais on classifie plus efficacement les variables par le type de données qu'elles peuvent représenter. De même qu'en français, la première distinction se fait entre le singulier et le pluriel. Les chaînes et les nombres sont des bouts de données singulières, alors que les listes de chaînes ou de nombres sont plurielles (et quand nous arriverons à la programmation orientée objet, vous verrez qu'un objet peut paraître singulier de l'extérieur, mais pluriel de l'intérieur, de même qu'une classe d'élèves). Nous appelons *scalaire* une variable singulière, et *tableau* une variable plurielle. Comme une chaîne peut être stockée dans une variable scalaire, nous pourrons écrire une version un peu plus longue (et mieux commentée) de notre premier exemple comme ceci :

```
$phrase = "Salut tout le monde !\n";  # Donner une valeur à une variable.
print $phrase;                        # Afficher la variable.
```

Remarquez que nous n'avons pas eu à prédéfinir le genre de la variable `$phrase`. Le caractère `$` dit à Perl que `phrase` est une variable scalaire, c'est-à-dire qu'elle contient une valeur singulière. Une variable tableau, en revanche, commencerait par le caractère `@` (on peut remarquer que `$` est un « S » stylisé, pour scalaire, alors que `@` est un « a » stylisé pour « array » , tableau en anglais).

Perl connaît d'autres types de variables aux noms bizarres comme « hachage », « handle » et « typeglob ». Tout comme les scalaires et les tableaux, ces types de variables sont également précédés par des caractères étranges. Dans un souci d'exhaustivité, voici la liste de tous les caractères étranges que vous pourrez rencontrer :

Type	Car.	Exemple	Est un nom pour
scalaire	`$`	`$cents`	Une valeur propre (nombre, chaîne).
tableau	`@`	`@large`	Une liste de valeurs, indexées par numéro.
hachage	`%`	`%interet`	Un groupe de valeurs indexées par chaîne.
sous-programme	`&`	`&comment`	Un bout de code Perl appelable.
typeglob	`*`	`*truc`	Tout ce qui est appelé `truc`.

Pour certains « puristes » des langages, ces caractères étranges sont une raison d'abhorrer Perl. Cette raison est superficielle. Ces caractères procurent de nombreux avantages : les variables peuvent être interpolées dans des chaînes sans syntaxe supplémentaire. Les scripts Perl sont faciles à lire (pour ceux qui ont pris la peine d'apprendre Perl !) parce que les noms se distinguent des verbes, et que de nouveaux verbes peuvent être ajoutés au langage sans abîmer d'autres scripts (nous vous avions dit que Perl était conçu pour évoluer). Et l'analogie avec les noms n'a rien de frivole ; il existe de nombreux précédents dans divers langages naturels qui requièrent des marqueurs de noms grammaticaux. Enfin, c'est notre avis !

Singularités

D'après notre exemple, on voit que l'on peut assigner une nouvelle valeur à un scalaire avec l'opérateur `=`, comme dans de nombreux autres langages informatiques. On peut affecter n'importe quelle forme de valeur scalaire à une variable SCALAIRE : entier, nombre à virgule flottante, chaîne, et même des choses ésotériques comme des références à d'autres variables ou à des objets. Il existe de nombreuses façons de générer ces valeurs à assigner.

Comme dans le shell UNIX[4], il est possible d'employer différents mécanismes de protection (*quoting*) pour fabriquer différents types de valeurs. Les doubles apostrophes font une *interpolation de variable*[5] et une *interprétation de l'antislash* (Comme en transformant `\t` en tabulation, `\n` en saut de ligne, `\001` en contrôle-A, etc., dans la tradition de nombreux programmes UNIX, alors que les apostrophes suppriment à la fois l'interpolation et l'interprétation. Et les apostrophes inverses exécutent un programme externe et renvoient la sortie du programme, ce qui fait que l'on peut capturer une ligne unique contenant toutes les lignes de la sortie.

```
$reponse = 42;                  # un entier
$pi = 3.14159265;               # un nombre "réel"
$avocats = 6.02e23;             # notation scientifique
$animal = "Chameau";            # chaîne
$signe = "Mon $animal et moi";  # chaîne avec interpolation
```

4. Nous parlons ici de tous les clones UNIX comme BSD, Linux et Unix.

5. Parfois appelée « substitution » par les programmeurs shell, mais nous préférons réserver ce mot pour quelque chose d'autre en Perl. Il vaut donc mieux parler d'interpolation. Nous employons ce terme dans son sens textuel (« Ce passage est une interpolation gnostique ») et non dans son sens mathématique (« Ce point du graphe est une interpolation entre deux autres points »).

```
$cout = 'Cela coûte $100';      # chaîne sans interpolation
$pourquoi = $parceque;          # une autre variable
$x = $taupes * $avocats;        # une expression
$exit = system("vi $x");        # code de retour d'une commande
$cwd = `pwd`;                   # chaîne générée par une commande
```

Et même si nous n'avons pas encore vu les valeurs exotiques, nous devons signaler que les scalaires peuvent référencer d'autres types de données incluant les sous-programmes et les objets.

```
$ary = \@montableau;            # référence à un tableau nommé
$hsh = \%monhachage;            # référence à un hachage nommé
$sub = \&monsousprog;           # référence à un sous-programme

$ary = [1,2,3,4,5];             # référence à un tableau anonyme
$hsh = {Na=>19, Cl=>35};        # référence à un hachage anonyme
$sub = sub { print $pays };     # référence à un sous-programme anonyme

$fido = new Chameau "Amelia";   # référence à un objet
```

Les variables non initialisées se mettent à exister quand le besoin s'en fait sentir. D'après le principe de moindre surprise, elles sont créées avec une valeur nulle, que ce soit `""` ou `0`. Selon l'endroit où elles sont utilisées, les variables sont automatiquement interprétées comme des chaînes, des nombres ou les valeurs « vrai » et « faux » (communément appelées valeurs booléennes). Les opérateurs attendent certains types de valeurs en paramètres, et nous dirons donc que ces opérateurs « fournissent » ou « procurent » un contexte scalaire à ces paramètres. Nous serons parfois plus spécifiques et dirons qu'il fournit un contexte numérique, un contexte de chaîne ou un contexte booléen à ces paramètres (nous parlerons plus loin du contexte de liste, qui est l'inverse du contexte scalaire). Perl convertit automatiquement les données sous la forme requise par le contexte courant, non sans raison. Par exemple, supposons que nous ayons écrit ceci :

```
$chameaux = '123';
print $chameaux + 1, "\n";
```

La valeur originelle de `$chameaux` est une chaîne, mais elle est convertie en un nombre pour y ajouter `1`, puis reconvertie en chaîne pour être affichée : `"124"`. Le saut de ligne, représenté par `"\n"`, est également dans un contexte de chaîne, mais puisqu'il est déjà une chaîne, aucune conversion n'est nécessaire. Mais remarquez que nous avons dû employer des apostrophes doubles ; les apostrophes simples (`'`) autour de `\n` auraient donné une chaîne à deux caractères constituée d'un antislash suivi d'un `n`, ce qui n'est en rien un saut de ligne.

D'une certaine manière, les apostrophes simples et doubles représentent donc un autre moyen de spécifier le contexte. L'interprétation du contenu d'une chaîne protégée dépend de la forme de protection utilisée. Nous verrons plus loin certains autres opérateurs qui fonctionnent comme des protections de façon syntaxique, mais qui utilisent la chaîne d'une manière spéciale comme pour la correspondance de motifs ou la substitution. Ils fonctionnent tous comme des apostrophes doubles. Le contexte d'*apostrophe double* est le contexte « interpolatif » de Perl et est fourni par de nombreux opérateurs qui ne ressemblent pas à des guillemets.

De la même manière, une référence se comporte comme telle lorsque vous êtes dans un contexte de « déréférencement », sinon elle agit comme une simple valeur scalaire. Par exemple nous écririons :

```
$fido = new Chameau "Amelia";
if (not $fido) { die "chameau mort"; }
$fido->seller();
```

Ici nous créons une référence à l'objet Chameau et la plaçons dans la variable `$fido`. À la ligne suivante nous testons `$fido` dans un contexte booléen, et nous levons une exception si sa valeur est fausse, ce qui dans ce cas voudrait dire que le constructeur `new Chameau` a échoué à la création de l'objet Chameau. Mais à la dernière ligne, nous utilisons `$fido` comme référence pour trouver la méthode `seller()` appliquée à l'objet référencé par `$fido`, qui se trouve être un Chameau, et donc Perl cherche la méthode `seller()` pour l'objet Chameau. Nous reviendrons sur ce sujet. Souvenez-vous juste pour le moment que le contexte est important en Perl, car il sert à lever toute ambiguïté sans faire aucune déclaration, contrairement à beaucoup d'autres langages.

Pluralités

Certains types de variables contiennent plusieurs valeurs qui sont logiquement liées entre elles. Perl connaît deux types de variables multivaleurs : les tableaux et les *hashes*, tables de hachage (ou hachages). Ils se comportent comme des scalaires par de nombreux aspects. Ils se mettent à exister quand le besoin s'en fait sentir et ne contiennent alors rien. Quand on leur assigne une valeur, ils fournissent un contexte de *liste* du côté droit de l'assignation.

On emploie un tableau quand on veut chercher quelque chose par son numéro. On emploie une table de hachage quand on veut chercher quelque chose par son nom. Ces deux concepts sont complémentaires. On voit souvent des gens utiliser un tableau pour traduire les numéros de mois en noms, et un hachage correspondant pour traduire les noms en numéros (les hachages ne se limitent évidemment pas aux seuls nombres. Par exemple, il est possible d'avoir un hachage qui traduise les noms de mois en nom de pierres porte-bonheur).

Tableaux. Un tableau est une liste ordonnée de scalaires, indicée par la position du scalaire dans la liste. Celle-ci peut contenir des nombres, ou des chaînes, ou un mélange des deux (elle peut aussi contenir des références à d'autres listes ou hachages). Pour assigner une liste de valeurs à un tableau, il suffit de regrouper les variables (par des parenthèses) :

```
@maison = ("lit", "chaise", "table", "poele");
```

Inversement, si l'on utilise `@maison` dans un contexte de liste, comme ce que l'on trouve à droite d'une assignation, on obtient la même liste qui a été entrée. On peut donc assigner quatre variables depuis le tableau comme ceci :

```
($Procuste, $porteurs, $multiplication, $gratter) = @maison;
```

C'est ce que l'on appelle des assignations de liste. Elles se produisent logiquement en parallèle, et il est possible de permuter deux variables en écrivant :

```
($alpha,$omega) = ($omega,$alpha);
```

Tout comme en C, les tableaux démarrent à zéro ; quand on parle des quatre premiers

éléments d'un tableau, leur énumération court de 0 à 3.[6] Les indices de tableaux sont entourés de crochets [comme ceci], et si l'on veut sélectionner un seul élément du tableau, on s'y réfère par `$maison[`*n*`]`, où *n* est l'indice (un de moins que le numéro de l'élément) que l'on désire ; voyez l'exemple ci-dessous. Puisque l'élément considéré est un scalaire, il est toujours précédé d'un `$`.

Si l'on veut valoriser un seul élément de tableau à la fois, on écrit l'assignation précédente comme suit :

```
$maison[0] = "lit";
$maison[1] = "chaise";
$maison[2] = "table";
$maison[3] = "poele";
```

Comme les tableaux sont ordonnés, il existe plusieurs opérations utiles s'y rapportant, comme les opérations de pile, `push` et `pop`. Une pile n'est après tout qu'une liste ordonnée, avec un début et une fin. Surtout une fin. Perl considère que la fin de la liste est le haut d'une pile (bien que la plupart des programmeurs Perl pensent qu'une liste est horizontale, le haut de la pile étant à droite).

Hachages. Un hachage (*hash* en anglais) est un ensemble *non ordonné* de scalaires indexé par une valeur chaîne associée à chaque scalaire. C'est pourquoi les hachages sont souvent appelés « tableaux associatifs ». La principale raison pour laquelle leur appellation a changé est que « tableau associatif » est long à écrire pour les paresseux, et nous en parlons tellement souvent que nous avons décidé de prendre un nom plus bref.[7] Ensuite, le terme « hachage » souligne le fait qu'il s'agit de tables non ordonnées. Elle sont en fait implémentées grâce à un système de recherche dans une table de hachage, ce qui leur donne leur rapidité, rapidité qui reste inchangée quel que soit le nombre de valeurs présentes. On ne peut pas faire de `push` ou de `pop` sur un hachage, parce que cela n'a aucun sens. Un hachage n'a ni début, ni fin. Les hachages sont néammoins très puissants et très utiles. Tant que vous ne penserez pas en terme de hachages, vous ne penserez pas en Perl.

Puisque les clefs d'un tableau associatif ne sont pas ordonnées, vous devez fournir la clef aussi bien que sa valeur lorsque celui-ci est rempli. Vous pouvez aussi bien lui affecter une liste comme pour un tableau ordinaire, mais chaque *paire* d'éléments sera interprétée comme la paire clef/valeur. Les tableaux associatifs utilisent le symbole `%` pour s'identifier. (Pour mémoire, si vous regardez attentivement le caractère `%`, vous pouvez voir la clef et sa valeur avec entre deux le /.)

Supposons que nous voulions traduire les abréviations des noms de jours en leurs correspondants complets. Nous pourrions écrire cette assignation de liste :

```
%longjour = ("Dim", "Dimanche", "Lun", "Lundi", "Mar", "Mardi",
             "Mer", "Mercredi", "Jeu", "Jeudi", "Ven",
             "Vendredi", "Sam", "Samedi");
```

6. Si cela vous semble bizarre, pensez à l'indice comme à un offset, un déplacement, c'est-à-dire le nombre d'éléments qui le précédent. Il n'y en a évidemment aucun avant le premier élément, qui a donc un offset de zéro. C'est ainsi que pensent les ordinateurs (c'est du moins ce que nous pensons).

7. Si tant est que l'on puisse qualifier de « bref » quoi que ce soit ayant trait aux hachages...

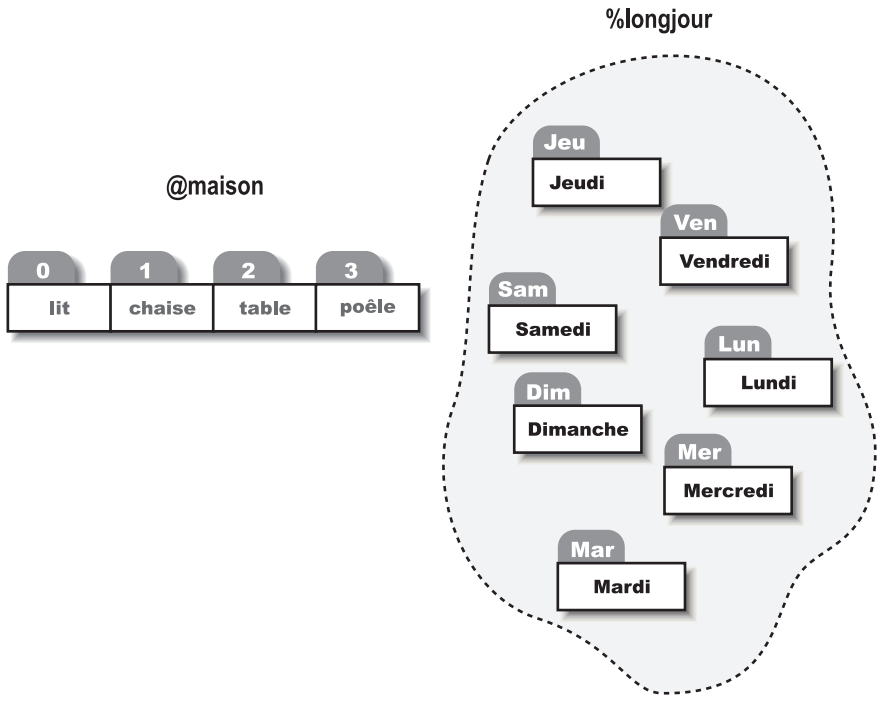


Figure 1-1. Un tableau et un hachage

Comme il est parfois difficile de lire un hachage ainsi défini, Perl fournit la séquence => (égal, supérieur) comme alternative à la virgule. Cette syntaxe (associée à un formatage créatif) facilite la distinction entre clefs et valeurs.

```
%longjour = (
    "Dim" => "Dimanche",
    "Lun" => "Lundi",
    "Mar" => "Mardi",
    "Mer" => "Mercredi",
    "Jeu" => "Jeudi",
    "Ven" => "Vendredi",
    "Sam" => "Samedi",
);
```

Il est non seulement possible d'assigner une liste à un hachage, comme nous l'avons fait ci-dessus, mais si l'on emploie un hachage dans un contexte de liste, celui-ci convertira le hachage en une liste de paires clef/valeurs dans un ordre quelconque. La plupart du temps, on se contente d'extraire une liste des seules clefs, grâce à la fonction (bien nommée) `keys`. Elle est également non ordonnée mais peut être facilement triée au besoin grâce à la fonction (bien nommée) `sort`. On peut alors utiliser les clefs ordonnées pour accéder aux valeurs dans un ordre déterminé.

Les hachages n'étant qu'une forme évoluée de tableau, il suffit d'en sélectionner un élément individuel en entourant la clef par des accolades. C'est ainsi que pour trouver la

valeur associée à Mer dans le hachage ci-dessus, il faut écrire `$longjour{"Mer"}`. Remarquez encore une fois qu'il s'agit d'une valeur scalaire et qu'il faut donc écrire `$` et non `%`.

D'un point de vue linguistique, la relation codée dans un hachage est génitive ou possessive, comme les mots « de », « du » ou « d' » en français. La femme *d'*Adam est Eve, et nous pouvons écrire :

```
$femme{"Adam"} = "Eve";
```

Complexités

La famille des tableaux et des hachages permet de représenter des structures de données simples et élégantes, mais dans la réalité le problème ne se pliera pas toujours à cette simplicité. Il est parfois nécessaire de construire des structures de données non linéaires. Perl nous permet de réaliser ce type de structure d'une manière très simple. Il permet en effet de manipuler de simples scalaires qui se réfèrent à des tableaux. Nous le faisons tous les jours dans le langage courant lorsque nous utilisons un mot par exemple aussi complexe que « gouvernement » pour représenter une entité aussi complexe et inscrutable. (C'est un exemple parmi d'autres.)

Pour développer notre exemple précédent, supposons que nous parlons des femmes de Jacob et Adam. Jacob a eu quatre femmes. Nous pourrions penser l'écrire de cette façon :

```
$femme{"Jacob"} = ("Léa", "Rachel", "Bilha", "Zilpa");        # MAUVAIS
```

Mais cette représentation n'est pas bonne car même les parenthèses ne sont pas assez puissantes pour transformer une liste en scalaire. (Les parenthèses sont utilisées pour le groupage syntaxique et les virgules pour la séparation syntaxique.) Il faut indiquer à Perl que cette liste doit être un scalaire. Les crochets sont parfaitement adaptés à cette fin :

```
$femme{"Jacob"} = ["Léa", "Rachel", "Bilha", "Zilpa"];        # ok
```

Cette ligne crée un tableau anonyme et affecte sa référence à l'élément associatif `$femme{"Jacob"}`. Ainsi nous avons un tableau associatif nommé contenant un tableau anonyme. C'est comme ça que Perl traite les tableaux multidimensionnels et les structures de données imbriquées. Comme avec les types tableaux, on peut aussi affecter des éléments séparés comme ceci :

```
$femme{"Jacob"}[0] = "Léa";
$femme{"Jacob"}[1] = "Rachel";
$femme{"Jacob"}[2] = "Bilha";
$femme{"Jacob"}[3] = "Zilpa";
```

Comme vous pouvez le voir, cela ressemble beaucoup à un tableau multidimensionnel avec des coordonnées de chaîne puis numérique. Visualisons maintenant une structure de données imbriquées sous forme d'arbre où nous listons aussi tous les fils de chacune des femmes de Jacob. Dans ce cas nous voulons utiliser le tableau associatif dans un contexte scalaire. Pour cela utilisons les accolades. (La valeur de la clef est un tableau entre crochets. Maintenant nous avons un tableau à l'intérieur d'une association, elle-même association.)

```
$filsDesFemmesDe{"Jacob"} = {
    "Léa"    => ["Ruben", "Siméon", "Lévi", "Juda", "Issachar", "Zabulon"],
```

```
    "Rachel" => ["Joseph", "Benjamin"],
    "Bilha"  => ["Dan", "Nephtali"],
    "Zilpa"  => ["Gad", "Aser"],
};
```

Les lignes suivantes sont équivalentes :

```
$fils_des_femmes_de{"Jacob"}{"Léa"}[0]    = "Ruben";
$fils_des_femmes_de{"Jacob"}{"Léa"}[1]    = "Siméon";
$fils_des_femmes_de{"Jacob"}{"Léa"}[2]    = "Lévi";
$fils_des_femmes_de{"Jacob"}{"Léa"}[3]    = "Juda";
$fils_des_femmes_de{"Jacob"}{"Léa"}[4]    = "Issachar";
$fils_des_femmes_de{"Jacob"}{"Léa"}[5]    = "Zabulon";
$fils_des_femmes_de{"Jacob"}{"Rachel"}[0] = "Joseph";
$fils_des_femmes_de{"Jacob"}{"Rachel"}[1] = "Benjamin";
$fils_des_femmes_de{"Jacob"}{"Bilha"}[0]  = "Dan";
$fils_des_femmes_de{"Jacob"}{"Bilha"}[1]  = "Nephtali";
$fils_des_femmes_de{"Jacob"}{"Zilpa"}[0]  = "Gad";
$fils_des_femmes_de{"Jacob"}{"Zilpa"}[1]  = "Aser";
```

On peut constater sur cet exemple que nous avons rajouté une dimension supplémentaire à notre tableau. Perl nous laisse le choix, mais la représentation interne est identique.

Le point important à noter ici est que Perl nous permet de représenter une structure de données complexes comme un simple scalaire. Cette encapsulation nous permet de construire une structure orientée objet. Lorsque nous avions invoqué le constructeur de `Chameau` comme ceci :

```
$fido = new Chameau "Amelia";
```

nous avions créé un objet `Chameau` représenté par le scalaire `$fido`. Mais la structure intime de l'objet `Chameau` est plus compliquée. En tant que concepteurs d'objets, nous ne sommes pas concernés par ceci, à moins d'être nous-mêmes les implémenteurs des méthodes de cette classe. Mais, généralement, un objet comme celui-ci serait représenté par un tableau associatif contenant les attributs de `Chameau`, tels son nom (« `Amelia` » dans ce cas, et non pas « `fido` »), le nombre de bosses (ce que nous n'avons pas spécifié, mais il vaut probablement 1, regardez la couverture du livre).

Simplicité

Si votre tête ne tourne pas après avoir lu cette dernière section alors vous avez une grosse tête. La plupart des humains sont rebutés par les structures compliquées du type gouvernement ou du type arbre généalogique. Pour cette raison nous disposons dans notre langage parlé d'artifices qui cachent cette complexité sous-jacente. La technique la plus probante est la clôture du champ sémantique, lexicalisation, *topicalization* en anglais, qui permet de s'accorder sur les termes du langage utilisé. La conséquence immédiate est une meilleure communication. La plupart d'entre nous utilisent cette technique couramment pour basculer d'un contexte sémantique vers un autre lorsque nous changeons simplement de sujet de conversation.

Perl dispose dans sa palette de ce type d'outil. L'un de ceux-ci est fourni par la simple déclaration `package`. Supposons que notre sujet est le `Chameau` de Perl. On déclenche le module `Chameau` par la déclaration :

```
package Chameau;
```

Tout symbole déclaré ensuite sera préfixé par le nom du module « `Chameau::` ». Ainsi si on écrit :

```
package Chameau;
$fido = &fetch();
```

on écrit en réalité `$Chameau::fido` et la fonction appelée est `&Chameau::fetch`, (nous parlerons des verbes plus tard). Si dans le cours du déroulement du code, l'interpréteur rencontre le code suivant :

```
package Chien;
$fido = &fetch();
```

il ne pourra confondre les noms réels car `$fido` est en fait `$Chien::fido`, et non `$Chameau::fido`. En informatique cela signifie un *espace de nommage*. Il n'a pas de limite cardinale. Perl n'en connaît qu'un à la fois. La simplification est le choix libre du programmeur.

Une chose remarquable est que dans :

```
$fido = new Chameau "Amelia";
```

nous invoquons le verbe `&new` dans le package `Chameau`, qui a pour nom plein `&Chameau::new`. et par

```
$fido->seller();
```

nous invoquons la routine `&Chameau::seller`, car `$fido` pointe sur un `Chameau`. C'est la programmation objet en perl.

La déclaration `package Chameau`, marque le début d'un paquetage, ici le paquetage Chameau. Parfois il suffit d'utiliser les noms et les verbes d'un paquetage existant. On utilise alors la déclaration `use`, qui charge le module dans l'interpréteur. Cela *doit* s'écrire :

```
use Chameau;
```

Puis on peut utiliser nos méthodes

```
$fido = new Chameau "Amelia";
```

Perl est WYSIWYG. Sinon Perl ne connaîtrait pas `Chameau`.

La chose intéressante ici est qu'il est juste nécessaire de connaître les capacités d'un `Chameau`. Il suffit de trouver le module sur *CPAN* :

```
use Le::Cool::Module;
```

Alors on peut actionner les verbes de ce module dans le champ du sujet.

La lexicalisation Perl, comme dans le langage naturel, donne un nom au champ lexical. Certains modules définissent des ensembles lexicaux et uniquement cela, pour l'interpréteur lui-même. Ces modules spéciaux se nomment *pragmas*. On verra souvent l'utilisation du pragma `strict` de cette manière :

```
use strict;
```

Le pragma `strict` permet d'ajouter une norme pour l'évaluation des termes par Perl. Cette norme oblige à préciser certains aspects comme la portée des variables du code, ce qui est utile dans le développement de grands projets. En effet, Perl est optimisé pour l'écriture de petits modules, mais avec cette directive de compilation il devient possible

de gérer des projets importants. On peut en effet l'ajouter à tout moment dans des modules élémentaires, ce qui permet de les faire interagir dans un même champ lexical. (Et autoriser du même coup une approche objet dans le développement de l'application elle-même.)

Verbes

Comme tous les langages informatiques impératifs, de nombreux verbes de Perl correspondent à des commandes : elles disent à l'interpréteur de Perl de faire quelque chose. En revanche, comme dans un langage naturel, les significations des verbes de Perl tendent à s'éparpiller dans plusieurs directions selon le contexte. Une instruction commençant par un verbe est en général purement impérative et évaluée entièrement pour ses effets secondaires. Nous appelons souvent ces verbes *procédures*, surtout quand ils sont définis par l'utilisateur. Un verbe fréquemment rencontré (que vous avez d'ailleurs déjà vu) est la commande `print` :

```
print "La femme d'Adam est ", $femme{'Adam'}, ".\n";
```

L'effet secondaire produit la sortie désirée.

Mais il existe d'autres « modes » que le mode impératif. Certains verbes permettent d'interroger l'interpréteur sur certaines valeurs comme dans les conditionnelles comme `if`. D'autres verbes traduisent leurs paramètres d'entrée en valeurs de retour, tout comme une recette vous indique comment transformer des ingrédients de base en un résultat (peut-être) comestible. Nous appelons souvent ces verbes des *fonctions*, par déférence envers des générations de mathématiciens qui ne savent pas ce que signifie le mot « fonctionnel » en langage naturel.

Un exemple de fonction interne est l'exponentielle :

```
$e = exp(1);   # 2.718281828459, ou à peu près
```

Mais Perl ne fait pas de distinction entre les procédures et les fonctions. Les termes seront employés de façon interchangeable. Les verbes sont parfois appelés sous-programmes (*subroutines*) quand ils sont définis par l'utilisateur, ou opérateurs (s'ils sont prédéfinis). Mais appelez-les comme il vous plaira ; ils renvoient tous une valeur, qu'elle soit ou non significative, que vous pouvez ou non choisir d'ignorer.

Au fur et à mesure que nous avançons, nous verrons d'autres exemples du comportement de Perl qui le font ressembler à un langage naturel. Nous avons déjà insidieusement présenté certaines notions de langage mathématique, comme l'addition et les indices, sans parler de la fonction exponentielle. Perl est aussi un langage de contrôle, un langage d'intégration, un langage de traitement de listes et un langage orienté objet. Entre autres.

Mais Perl est aussi un bon vieux langage informatique. Et c'est ce à quoi nous allons maintenant nous intéresser.

Un exemple de notation

Supposons que vous ayez un ensemble de notes pour chaque membre d'une classe d'élèves. Vous aimeriez une liste combinée de toutes les notes pour chaque étudiant, plus leur moyenne. Vous disposez d'un fichier texte (appelé, quelle imagination,

« notes ») qui ressemble à ceci :

```
Xavier 25
Samantha 76
Hervé 49
Peter 66
Olivier 92
Thierry 42
Fred 25
Samantha 12
Hervé 0
Corinne 66
etc.
```

Le script ci-dessous rassemble toutes leurs notes, détermine la moyenne de chaque étudiant et les affiche dans l'ordre alphabétique. Ce programme suppose, plutôt naïvement, qu'il n'existe pas deux Corinne dans votre classe. C'est-à-dire que s'il y a un deuxième élément pour Corinne, le programme supposera qu'il s'agit d'une autre note de la première Corinne.

Au fait, les numéros de lignes ne font pas partie du programme, toute ressemblance avec le BASIC mise à part.

```
 1: #!/usr/bin/perl
 2:
 3: open(NOTES, "notes") or die "Ouverture des notes impossible : $!\n";
 4: while ($ligne = <NOTES>) {
 5:     ($etudiant, $note) = split(/ /, $ligne);
 6:     $notes\{$etudiant\} .= $note . " ";
 7: }
 8:
 9: foreach $etudiant (sort keys %notes) {
10:     $scores = 0;
11:     $total = 0;
12:     @notes = split(/ /, $notes{$etudiant});
13:     foreach $note (@notes) {
14:         $total += $note;
15:         $scores++;
16:     }
17:     $moyenne = $total / $scores;
18:     print "$etudiant: $notes{$etudiant}\tMoyenne : $moyenne\n";
19: }
```

Ne louchez pas ainsi. Nous devons avouer que si cet exemple illustre beaucoup de ce que nous avons décrit jusqu'à présent, il comprend certaines choses que nous allons maintenant expliquer. Mais si vous regardez *un peu* dans le vague, vous pouvez commencer à voir apparaître certains motifs intéressants. Vous pouvez vous livrer à toutes les suppositions imaginables, et nous vous dirons plus tard si vous aviez raison.

Nous pourrions vous dire d'essayer de le lancer, mais peut-être ne savez-vous pas comment faire.

Comment faire

Bien. Vous devez être en train de vous demander comment lancer un programme Perl. La réponse la plus courte est que vous le donnez au programme interpréteur de Perl, qui s'appelle justement *perl* (notez le *p* minuscule). Une réponse plus complète commence comme ceci : Il Existe Plus D'Une Façon De Faire.[8]

La première manière d'invoquer *perl* (qui fonctionne sur la plupart des systèmes d'exploitation) est d'appeler explicitement *perl* depuis la ligne de commande.[9] Si vous vous trouvez sur une version d'UNIX et que ce que vous faites est relativement simple, vous pouvez utiliser l'option **-e** (le % de l'exemple suivant représentant une invite standard de shell, il ne faut pas le taper) :

```
% perl -e 'print "Salut tout le monde!\n";'
```

Sous d'autres systèmes d'exploitation, il faudra peut-être jouer des apostrophes. Mais le principe de base est le même : vous essayez de rentrer tout ce que Perl doit savoir en 80 colonnes.[10]

Pour les scripts plus longs, vous pouvez utiliser votre éditeur favori (ou un autre) pour écrire vos commandes dans un fichier, puis, si le script s'appelle « gradation », vous entrez :

```
% perl gradation
```

Vous invoquez toujours explicitement l'interpréteur Perl, mais au moins, vous n'avez pas à tout mettre sur la ligne de commande à chaque fois. Et vous n'avez pas à jouer des apostrophes pour plaire au shell.

La façon la plus commode d'invoquer un script est de simplement l'appeler par son nom (ou de cliquer sur son icone), et de laisser le système d'exploitation trouver l'interpréteur tout seul. Sous certains systèmes, il peut exister un moyen d'associer certaines extensions de fichier ou certains répertoires à une application particulière. Sur les systèmes UNIX qui supportent la notation `#!` (appelée *shebang*), la première ligne du script peut être magique et dire au système d'exploitation quel programme lancer. Mettez une ligne ressemblant[11] à la ligne 1 de notre exemple dans votre programme :

```
#!/usr/bin/perl
```

Tout ce qu'il vous reste à écrire est maintenant :

```
% gradation
```

Ce qui ne fonctionne évidemment pas parce que vous avez oublié de rendre le script exé-

8. There's More Than One Way To Do It : c'est le slogan de Perl, et vous allez être fatigué de l'entendre, à moins que vous ne soyez l'Expert Local, auquel cas vous serez fatigué de le répéter. Il est quelquefois raccourci en TMTOWTDI, que l'on prononce « tim-toady ». Mais vous pouvez le prononcer de la façon qui vous plaît. Après tout, TMTOWTDI.

9. En supposant que vous disposez d'une ligne de commande. Si vous travaillez avec un vieux Mac, vous aurez besoin de faire une mise à jour vers le système OS X.

10. Ce genre de script est souvent appelé *one-liner*, « une-ligne ». S'il vous arrive de fréquenter d'autres programmeurs Perl, vous découvrirez que certains d'entre eux sont friands de une-lignes. Perl a parfois été qualifié de « langage en écriture seule » à cause de ces voyous.

11. Si *Perl* ne se trouve pas dans */usr/bin*, vous devrez changer la ligne `#!`.

cutable (voir la page de manuel de *chmod*(1)[12] et qu'il n'est pas dans votre PATH. Dans ce cas, vous devrez fournir un nom de chemin complet pour que votre système d'exploitation sache comment trouver votre script. Quelque chose comme :

```
% ../bin/gradation
```

Enfin, si vous avez la malchance de travailler sur un ancien système UNIX qui ne reconnaît pas la ligne magique `#!`, ou si le chemin vers votre interpréteur est plus long que 32 caractères (une limite interne sur certains systèmes), vous pourrez contourner le problème comme ceci :

```
#!/bin/sh -- # perl, pour arrêter de boucler
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    if 0;
```

Certains systèmes d'exploitation peuvent demander une modification de ce qui précède pour gérer, */bin/sh*, *DCL*, *COMMAND.COM*, ou tout ce qui vous tient lieu d'interpréteur de commandes. Demandez à votre Expert Local.

Tout au long de ce livre, nous nous contenterons d'employer `#!/usr/bin/perl` pour représenter toutes ces notions et toutes ces notations, mais vous saurez à quoi vous en tenir.

Une astuce : quand vous écrivez un script de test, ne l'appelez pas *test*. Les systèmes UNIX ont une commande *test* interne, qui sera exécutée à la place de votre script. Appelez-le plutôt *essai*.

Une autre astuce : pendant que vous apprenez Perl, et même quand vous croyez savoir ce que vous faites, nous vous suggérons d'utiliser l'option **`-w`**, surtout pendant le développement. Cette option active toutes sortes de messages d'avertissement utiles et intéressants (dans le désordre). L'option **`-w`** peut être placée sur la ligne « magique », comme ceci :

```
#!/usr/bin/perl -w
```

Maintenant que vous savez lancer votre programme Perl (à ne pas confondre avec le programme *perl*), revenons à notre exemple.

Handles de fichiers

À moins qu'il ne s'agisse d'un projet d'IA modélisant un philosophe solipsiste, votre programme a besoin de communiquer avec le monde extérieur. Aux lignes 3 et 4 de notre exemple de notation, on voit le mot `NOTES`, qui illustre un autre type de données de Perl, le *handle de fichier*. Un handle de fichier est un nom que l'on donne à un fichier, un périphérique, une socket ou un pipe pour que vous puissiez vous souvenir de ce dont

12. Bien que Perl ait son lot de notations bizarres, celle-ci doit tout à UNIX. *chmod*(1) signifie que vous devez vous reporter à la page de manuel de la commande chmod à la section 1 de votre manuel UNIX. Si vous entrez soit `man 1 chmod`, soit `man -s 1 chmod` (selon votre type d'UNIX), vous devriez trouver tout ce que sait votre système de la commande *chmod*. (Bien sûr, si votre type d'UNIX est « Pas UNIX ! » , vous devrez vous référer à la documentation de votre système pour la commande équivalente, à supposer que vous en soyez pourvu. Votre consolation étant que, si cette commande existe, elle aura un meilleur nom que *chmod*.)

vous parlez, et pour dissimuler certaines complexités, notamment relatives au tampons. (En interne, les handles de fichiers sont similaires aux streams d'un langage comme C ou C++, ou aux canaux d'E/S du BASIC.)

Les handles de fichiers facilitent la gestion des entrées et des sorties depuis plusieurs sources et vers plusieurs destinations. C'est sa capacité à communiquer avec de nombreux fichiers et de nombreux processus qui fait de Perl un bon langage d'intégration.[13]

On crée un handle de fichier et on l'attache à un fichier par la fonction `open`. `open` prend deux paramètres : le handle de fichier et le nom du fichier auquel on veut l'associer. Perl vous donne aussi quelques handles de fichier prédéfinis (et déjà ouverts). `STDIN` est le canal d'entrée standard du programme, alors que `STDOUT` en est le canal de sortie standard. Et `STDERR` est un canal de sortie supplémentaire pour que votre programme puisse faire des remarques à part pendant qu'il transforme (ou essaye de transformer) votre entrée en sortie.[14]

Comme l'on peut utiliser la fonction `open` pour créer des handles de fichiers à des fins diverses (entrées, sorties, pipes), vous devrez pouvoir spécifier le comportement que vous désirez. Il suffit d'ajouter des caractères au nom du fichier.

```
open(SESAME, "nomdefichier");         # lire depuis un fichier existant
open(SESAME, "<nomdefichier ");       #   (idem, explicitement)
open(SESAME, ">nomdefichier ");       # créer un fichier et y écrire
open(SESAME, ">>nomdefichier ");      # ajouter à un fichier existant
open(SESAME, "| sortie_de_commande"); # mettre en place un filtre de sortie
open(SESAME, "entrée de commande |"); # mettre en place un filtre d'entrée
```

On voit que le nom en question est quelconque. Une fois ouvert, le handle de fichier SESAME peut être utilisé pour accéder au fichier, ou au pipe, jusqu'à ce qu'il soit explicitement fermé (avec, on s'en doutait, `close(SESAME)`), ou si le handle de fichier est attaché à un autre fichier par un autre `open` sur le même handle.[15]

Une fois le handle de fichier ouvert en entrée (ou si vous désirez utiliser `STDIN`), vous pouvez lire une ligne grâce à l'opérateur de lecture de ligne, `<>`. Il est également connu

13. Sans oublier les qualités suivantes : il travaille sur 8 bits sans problème, il est intégrable, et on peut y intégrer d'autres choses *via* ses modules d'extension. Il est concis et travaille sans problème en réseau. Il est pour ainsi dire conscient de son environnement. Il est possible de l'invoquer de bien des manières (comme nous l'avons déjà vu). Mais surtout, le langage lui-même est assez souple pour qu'il soit possible de le faire « circuler » autour d'un problème. On revient encore à ce fameux concept de TMTOWTDI.

14. Ces handles de fichier sont typiquement attachés à votre terminal et vous pouvez donc entrer des données vers votre programme et en voir le résultat, mais ils peuvent aussi se rattacher à des fichiers (ou équivalents). Perl fournit ces handles prédéfinis parce que le système d'exploitation les fournit déjà, d'une façon ou d'une autre. Sous UNIX, les processus héritent de l'entrée, de la sortie et de l'erreur standard de leur processus père, typiquement un shell. L'une des tâches d'un shell est de mettre en place ces flux d'E/S afin que le processus fils n'ait pas à s'en préoccuper.

15. L'ouverture d'un handle de fichier déjà ouvert ferme implicitement le premier fichier, le rendant ainsi inaccessible à ce handle, et ouvre un fichier différent. Il faut faire attention à ce que l'on veut vraiment faire. Cela se produit parfois accidentellement, comme quand l'on écrit `open($handle, $fichier)`, et que `$handle` contient une chaîne constante. Assurez-vous d'assigner `$handle`, ou vous ne ferez qu'ouvrir un nouveau fichier sur le handle nul.

sous le nom d'opérateur *diamant* en raison de sa forme. L'opérateur diamant entoure le handle de fichier `<SESAME>` dont on veut lire des lignes.[16] Un exemple utilisant le handle de fichier `STDIN` pour lire une réponse fournie par l'utilisateur ressemblerait à ceci :

```
print STDOUT "Entrez un nombre : ";       # demander un nombre
$nombre = <STDIN>;                        # entrer le nombre
print STDOUT "Le nombre est $nombre\n";   # afficher le nombre
```

Avez-vous vu ce que nous venons de vous glisser sous les pieds ? Qu'est-ce que ce `STDOUT` fait dans ces instructions `print` ? C'est simplement une des façons possibles d'utiliser un handle de fichier. Un handle de fichier peut être fourni comme premier argument de l'instruction `print` et, s'il est présent, il indique où est la sortie. Dans ce cas, le handle de fichier est redondant, car la sortie aurait, de toute manière, été `STDOUT`. De même que `STDIN` est l'entrée par défaut, `STDOUT` est la sortie par défaut (nous l'avons enlevé en ligne 18 de l'exemple de notation pour éviter de vous embrouiller).

Nous avons fait autre chose. Si vous essayez l'exemple ci-dessus, vous pouvez voir apparaître une ligne vide supplémentaire. En effet, la lecture n'enlève pas automatiquement le saut de ligne de votre ligne d'entrée (qui serait par exemple « `9\n` »). Au cas où vous voudriez enlever le saut de ligne, Perl fournit les fonctions `chop` et `chomp`. `chop` enlève (et renvoie) le dernier caractère qu'on lui passe sans se poser de question, alors que `chomp` n'enlève que le marqueur de fin d'enregistrement (en général, `"\n"`) et renvoie le nombre de caractères enlevés. Cette expression est souvent employée pour lire une seule ligne :

```
chop($nombre = <STDIN>);    # lire le nombre et enlever le saut de ligne
```

ce qui est identique à

```
$number = <STDIN>;          # lire le nombre
chop($number);              # enlever le saut de ligne
```

Opérateurs

Comme nous l'avons laissé entendre plus haut, Perl est également un langage mathématique. C'est vrai à différents niveaux, depuis les opérations logiques au niveau binaire jusqu'aux manipulations de nombres et d'ensembles, de prédicats plus vastes à diverses abstractions. Et comme nous le savons tous depuis que nous avons étudié les mathématiques à l'école, les mathématiciens adorent les symboles étranges. Pire encore, les informaticiens ont inventé leur propre version de ces symboles. Perl en connaît un certain nombre, mais vous pouvez respirer car la plupart sont directement empruntés au C, au FORTRAN, à *sed* ou à *awk*, et seront donc familiers aux utilisateurs de ces langages.

Vous pouvez vous féliciter de connaître plus de termes lexicaux, autant de point d'entrée vers les autres langages.

Les opérateurs internes de Perl peuvent être classés par nombre d'opérandes, en opérateurs unaires, binaires et ternaires. Ils peuvent être classés par opérateurs infixes ou pré-

16. L'opérateur diamant par défaut, `<>`, lit des lignes depuis tous les fichiers spécifiés sur la ligne de commande, ou depuis `STDIN` si aucun n'a été spécifié (ce comportement est standard pour de nombreux programmes « filtres » UNIX).

fixes, mais aussi par le type d'objet avec lesquels ils travaillent, comme les nombres, les chaînes ou les fichiers. Nous vous donnerons plus loin une table de tous les opérateurs, mais en voici quelques-uns par lesquels vous pourrez commencer.

Quelques opérateurs d'arithmétique binaire

Les opérateurs arithmétiques font exactement ce que l'on peut attendre d'eux après les avoir étudiés à l'école.

Exemple	Nom	Résultat
`$a + $b`	Addition	Somme de `$a` et de `$b`.
`$a * $b`	Multiplication	Produit de `$a` et de `$b`.
`$a % $b`	Modulo	Reste de `$a` divisé par `$b`.
`$a ** $b`	Puissance	`$a` puissance `$b`.

Oui, nous avons bien laissé de côté la soustraction et la division. Mais vous devez pouvoir deviner comment celles-ci fonctionnent. Essayez-les et voyez si vous aviez raison (ou trichez et cherchez dans le chapitre 3, *Opérateurs unaires et binaires*.) Les opérateurs arithmétiques sont évalués dans l'ordre habituel (c'est-à-dire la puissance avant la multiplication, et la multiplication avant l'addition). Les parenthèses servent à modifier l'ordre.

Opérateurs sur les chaînes

Il existe également un opérateur d'« addition » pour les chaînes, qui effectue leur concaténation. Contrairement à d'autres langages qui le confondent avec l'addition arithmétique, Perl définit un opérateur distinct (`.`) pour cette opération.

```
$a = 123;
$b = 456;
print $a + $b;     # affiche 579
print $a . $b;     # affiche 123456
```

Il existe aussi un opérateur de « multiplication » pour les chaînes, également appelée *répétition*. Il s'agit encore d'un opérateur distinct (`x`) qui le différencie de la multiplication numérique :

```
$a = 123;
$b = 3;
print $a * $b;     # affiche 369
print $a x $b;     # affiche 123123123
```

L'opérateur de répétition est un peu inhabituel en ce qu'il prend une chaîne pour son argument de gauche et un nombre pour celui de droite. Remarquez également comment Perl convertit automatiquement les nombres en chaînes. Tous les nombres ci-dessus pourraient être protégés par des apostrophes doubles avec le même résultat. La conversion interne se serait par contre produite dans le sens inverse (c'est-à-dire de chaînes vers des nombres).

Il faut encore penser à quelques détails. La concaténation de chaînes est également implicite dans l'interpolation qui se produit dans des chaînes entre doubles apostrophes. Quant une liste de valeurs est affichée, les chaînes de caractères sont en fait concaténées. Les trois instructions suivantes produisent donc le même résultat :

```
print $a . ' est égal à ' . $b . "\n";    # opérateur point
print $a, ' est égal à ', $b, "\n";       # liste
print "$a est égal à $b\n";               # interpolation
```

Le choix de celui qu'il convient d'utiliser ne dépend que de vous.

L'opérateur `x` peut sembler relativement inutile à première vue, mais il trouve son utilité dans des cas comme celui-ci :

```
print "-" x $lrgecr, "\n";
```

Ceci trace une ligne à travers l'écran, à supposer que sa largeur soit `$lrgecr`.

Opérateurs d'affectation

Bien qu'il ne s'agisse pas vraiment d'un opérateur mathématique, nous avons déjà employé en de nombreuses occasions l'opérateur d'assignation simple, `=`. Essayez de vous souvenir que `=` signifie « est mis à » au lieu de « égale » (il existe aussi un opérateur d'égalité mathématique `==` qui signifie « égale » , et si vous commencez à réfléchir tout de suite à la différence entre les deux, vous aurez beaucoup moins mal à la tête plus tard. L'opérateur `==` est comme une fonction qui renvoie une valeur booléenne, alors que `=` est comme une procédure évaluée dont l'effet est de modifier une variable).

De même que les opérateurs précédents, les opérateurs d'assignation sont des opérateurs binaires infixes, ce qui veut dire qu'ils ont un opérande de chaque côté de l'opérateur. L'opérande de droite peut être n'importe quelle expression, mais celui de gauche doit être une *lvalue* correcte (ce qui, traduit en bon français, signifie un emplacement de stockage valide, comme une variable ou un élément de tableau). L'opérateur d'assignation le plus courant est l'assignation simple. Il détermine la valeur de l'expression à sa droite, puis met la variable à sa gauche à cette valeur :

```
$a = $b;
$a = $b + 5;
$a = $a * 3;
```

Remarquez que la dernière assignation se réfère deux fois à la même variable ; une fois pour le calcul et une autre pour l'assignation. Rien d'extraordinaire à cela, mais cette opération est assez commune pour qu'elle connaisse une abréviation (empruntée au C). Si l'on écrit :

```
lvalue opérateur= expression
```

cela sera évalué comme si c'était :

```
lvalue = lvalue opérateur expression
```

sauf que la lvalue n'est pas calculée deux fois. On ne perçoit de différence que si l'évaluation de la *lvalue* a un effet secondaire. Mais quand il y a une différence, l'opérateur fait généralement ce que vous vouliez.

On peut, par exemple, écrire :

```
$a *= 3;
```

ce qui se lit « multiplier $a par 3 et l'affecter à $a ». Presque tous les opérateurs binaires de Perl le permettent, et même certains de ceux qui ne le peuvent pas en C :

```
$ligne .= "\n";  # Ajouter un saut de ligne à $ligne.
$fill x= 80;     # Répéter $fill 80 fois sur elle-même.
$val ||= "2";    # Mettre $val à 2 si ce n'est pas déjà le cas.
```

La ligne 6 de notre exemple[17] de notation contient deux concaténations de chaînes, dont l'une est un opérateur d'affectation. Et la ligne 14 contient un +=.

Quel que soit le type d'opérateur d'affectation utilisé, la valeur finale qui est renvoyée est celle de l'affectation complète[18]. Ceci ne surprendra pas les programmeurs C qui le savait déjà en écrivant

```
$a = $b = $c = 0;
```

pour initialiser à 0 toutes les variables.

Ce qui par contre les surprendra est que l'affectation Perl retourne en lvalue la variable elle-même, de telle sorte qu'il est possible en une instruction de le modifier deux fois, c'est pourquoi nous pouvons écrire :

```
($temp -= 32) *= 5/9;
```

pour faire la conversion en ligne Fahrenheit vers Celsius. C'est aussi pourquoi nous avons pu écrire plus haut dans ce chapitre :

```
chop($nombre = <STDIN>);
```

pour que la valeur finale de $nombre soit modifiée par chop. Pour résumer, on peut utiliser cette fonctionalité chaque fois que l'on veut faire une copie suivie d'une modification de variable.

Ce truc peut être utilisé pour faire deux choses sur la même ligne Perl.

Opérateurs arithmétiques unaires

Comme si $variable += 1 n'était pas assez court, Perl emprunte au C un moyen encore plus concis d'incrémenter une variable. Les opérateurs d'autoincrémentation et d'autodécrémentation ajoutent (ou soustraient) un à la valeur de la variable. Ils peuvent être placés avant ou après cette variable, ce qui détermine le moment de leur évaluation.

Exemple	Nom	Résultat
++$a, $a++	Autoincrémentation	Ajouter 1 à $a.
--$a, $a--	Autodécrémentation	Soustraire 1 de $a.

Si l'on place l'un des opérateurs auto avant la variable, il s'agit d'une variable préincrémentée (ou prédécrémentée). Sa valeur sera modifiée avant son référencement. S'il est

17. Vous ne l'avez pas oublié ?

18. Ce qui diffère du Pascal, par exemple, où l'affectation est une instruction et n'a pas de valeur. Nous avons dit que l'affectation est comme un procédure, mais souvenez-vous qu'en Perl les procédures aussi ont une valeur.

placé après la variable, il s'agit d'une variable post-incrémentée (ou post-décrémentée) et sa valeur est modifiée après son utilisation. Par exemple :

```
$a = 5;        # $a prend la valeur 5
$b = ++$a;     # $b prend la valeur incrémentée de $a, soit 6
$c = $a--;     # $c prend la valeur 6, puis $a passe à 5
```

La ligne 15 de notre exemple de notation incrémente de un le nombre de notes, afin que nous puissions calculer la moyenne. Nous utilisons un opérateur de postincrémentation (`$scores++`), mais cela n'a pas d'importance dans ce cas puisque l'expression se trouve dans un contexte vide, ce qui n'est qu'une façon particulière de dire que l'expression n'est évaluée que pour l'effet secondaire de l'incrémentation de la variable. La valeur renvoyée est jetée au panier.[19]

Opérateurs logiques

Les opérateurs logiques, que l'on connaît également sous le nom d'opérateurs « court-circuit », permettent au programme de prendre des décisions en fonction de critères multiples sans utiliser les conditionnelles imbriquées. On les appelle court-circuit car ils arrêtent l'évaluation de leur argument de droite si l'argument de gauche suffit à déterminer la valeur globale.

Perl a en fait deux ensembles d'opérateurs logiques : l'un d'eux, plutôt poussiéreux, emprunté au C, et un superbe ensemble tout neuf d'opérateurs à précédence ultra-basse qui analysent plus comme ce qu'on en attend (mais ils se comportent de la même façon une fois analysés). Perl possède deux ensembles d'opérateurs logiques, l'un dit traditionnel emprunté au C, l'autre avec une précédence encore plus faible venant du BASIC. L'un a la précédence la plus forte des opérateurs du langage, l'autre la plus faible. Souvent les expressions sont équivalentes, c'est une question de préférences. (Pour une comparaison, voir la section *And, or, not et xor logiques* au chapitre 3.) Bien qu'ils ne sont pas interchangeables, à cause de la précédence, une fois parsés ils s'exécutent avec le même code. La précédence précise l'étendue locale de arguments.

Tableau 1-1. Opérateurs logiques

Exemple	Nom	Résultat
`$a && $b`	Et	`$a` si `$a` est faux, `$b` sinon.
`$a || $b`	Ou	`$a` si `$a` est vrai `$b` sinon.
`! $a`	Non	Vrai si `$a` n'est pas vrai.
`$a and $b`	Et	`$a` si `$a` est faux, `$b` sinon.
`$a or $b`	Ou	`$a` si `$a` est vrai, `$b` sinon.
`not $a`	Non	Vrai si `$a` n'est pas vrai.
`$a xor $b`	Ou exclusif	Vrai si `$a` ou `$b` est vrai, mais pas les deux.

19. Et en fait, l'optimisateur le remarque et optimise la post-incrémentation en préincrémentation, qui est un peu plus efficace à l'exécution (ce que vous n'avez pas vraiment besoin de savoir, mais nous pensions que cela pouvait vous intéresser).

Puisque les opérateurs logiques « court-circuitent », ils sont souvent utilisés pour exécuter du code de manière conditionnelle. La ligne suivante (de notre exemple de notation) essaie d'ouvrir le fichier « notes ». Si elle peut ouvrir le fichier, elle saute à la ligne suivante du programme. Dans le cas contraire, elle affiche un message d'erreur et arrête l'exécution.

```
open(NOTES, "notes") or die "Ouverture du fichier notes impossible: $!\n";
```

Littéralement, « Ouvrir notes ou mourir ! » est encore un exemple de langage naturel, et les opérateurs de court-circuit préservent le flux visuel. Les actions importantes sont listées en bas à gauche de l'écran et les actions secondaires sont cachées à droite. (La variable `$!` contient le message d'erreur renvoyé par le système d'exploitation ; voir le chapitre 28, *Noms spéciaux*.) Bien sûr, ces opérateurs logiques peuvent aussi être employés dans des constructions plus traditionnelles, comme les instructions `if` et `while`.

Opérateurs de comparaison

Les opérateurs de comparaison, ou relationnels, nous indiquent la relation existant entre deux valeurs scalaires (nombres ou chaînes). Il existe deux ensembles d'opérateurs, dont l'un effectue les comparaisons numériques et l'autre des comparaisons de chaînes. (Dans les deux cas, les arguments seront d'abord transtypés, « contraints », pour prendre le bon type.) Le tableau suivant suppose que `$a` et `$b` sont respectivement les arguments de gauche et de droite.

Comparaison	Numérique	Chaîne	Valeur de retour
égal	`==`	`eq`	Vrai si `$a` est égal à `$b`.
Différent	`!=`	`ne`	Vrai si `$a` n'est pas égal à `$b`.
Inférieur à	`<`	`lt`	Vrai si `$a` est plus petit que `$b`.
Supérieur à	`>`	`gt`	Vrai si `$a` est plus grand que `$b`.
Inférieur ou égal à	`<=`	`le`	Vrai si `$a` est inférieur ou égal à `$b`.
Supérieur ou égal à	`>=`	`ge`	Vrai si `$a` est supérieur ou égal à `$b`.
Comparaison	`<=>`	`cmp`	0 si égal, 1 si `$a` sup., -1 si `$b` sup.

On peut penser que les deux derniers opérateurs (`<=>` et `cmp`) sont entièrement redondants. Vous avez raison. Cependant, ils sont incroyablement utiles dans les sous-programmes de tri (*sort*).[20]

20. Certains voient en la redondance le mal incarné car elle interdit à un langage d'être minimaliste, ou « orthogonal ». Mais Perl n'est pas orthogonal, il est « diagonal ». Nous entendons par ceci que Perl ne vous force pas à tourner sans arrêt à angle droit. Il vaut parfois mieux suivre l'hypothénuse du triangle pour aller où l'on veut. Qui dit TMTOWTDI dit courts-circuits. Qui dit courts-circuits dit efficacité.

Opérateurs de tests de fichier

Les opérateurs de tests de fichier permettent de tester si certains attributs de fichier sont positionnés avant de faire aveuglément n'importe quoi avec ces fichiers. Par exemple, il vaut mieux s'assurer que le fichier */etc/passwd* existe déjà avant de l'ouvrir en création, effaçant ainsi tout ce qui s'y trouvait auparavant.

Exemple	Nom	Résultat
`-e $a`	Existe	Vrai si le fichier nommé par `$a` existe.
`-r $a`	Lecture	Vrai si le fichier nommé par `$a` est accessible en lecture.
`-w $a`	Écriture	Vrai si le fichier nommé par `$a` est accessible en écriture.
`-d $a`	Répertoire	Vrai si le fichier nommé par `$a` est un répertoire.
`-f $a`	Fichier	Vrai si le fichier nommé par `$a` est un fichier régulier.
`-T $a`	Fichier texte	Vrai si le fichier nommé par `$a` est un fichier texte.

En voici quelques exemples :

```
-e "/usr/bin/perl" or warn "Perl est mal installé\n";
-f "/boot" and print "Félicitations, nous devons être sous Unix BSD\n";
```

Remarquez qu'un fichier régulier n'est pas la même chose qu'un fichier texte. Les fichiers binaires comme */vmunix* sont des fichiers réguliers, mais ne sont pas des fichiers texte. Les fichiers texte sont l'inverse des fichiers binaires, alors que les fichiers réguliers sont l'inverse des fichiers « irréguliers » comme les répertoires et les périphériques.

Il existe *beaucoup* d'opérateurs de test de fichier, dont un grand nombre n'ont pas été cités. La plupart sont des opérateurs booléens unaires : ils ne prennent qu'un seul opérande, un scalaire qui donne un fichier ou un handle de fichier, et ils renvoient une valeur vraie ou fausse. Quelques-uns renvoient une valeur plus complexe comme la taille du fichier ou son âge, mais vous pourrez les rechercher en temps utile dans la section *Opérateurs unaires nommés et de test de fichier* au chapitre 3.

Structures de contrôle

Jusqu'ici, et hormis notre programme de notation, tous nos exemples étaient complètement linéaires ; toutes les commandes étaient exécutées dans l'ordre. Nous avons vu quelques exemples d'utilisation des opérateurs court-circuit permettant l'exécution conditionnelle d'une commande. Bien que certains programmes linéaires s'avèrent extrêmement utiles (c'est le cas de beaucoup de scripts CGI), on peut écrire des programmes bien plus puissants grâce aux expressions conditionnelles et aux mécanismes de boucles, que l'on appelle structures de contrôle. Perl peut donc être considéré comme un langage de contrôle.

Mais pour contrôler le cours des choses, il faut être en mesure de décider, et pour décider des choses, il faut distinguer le vrai du faux.

Qu'est-ce que la vérité ?

Nous avons déjà parlé de la vérité[21] et nous avons mentionné que certains opérateurs renvoient une valeur vraie ou fausse. Avant de continuer, nous devons expliquer exactement ce que nous entendons par là. Perl ne traite pas tout à fait la vérité comme les autres langages informatiques, mais son comportement prend tout son sens avec un peu d'habitude. (En fait, nous espérons qu'il prendra tout son sens après avoir lu ce qui suit.)

À la base, Perl estime que la vérité est « évidente en soi ». C'est une façon un peu spécieuse de dire que l'on peut évaluer presque n'importe quoi pour connaître sa valeur de vérité. Perl emploie en pratique des définitions de la vérité qui dépendent du type de ce que l'on est en train d'évaluer. Il se trouve qu'il existe beaucoup plus de types de vérité que de types de non-vérité.

La vérité en Perl est toujours évaluée dans un contexte scalaire (sinon, aucun transtypage n'est effectué). Voici donc les règles des divers types de valeurs qu'un scalaire peut contenir :

1. Toute chaîne est vraie sauf `""` et `"0"`.
2. Tout nombre est vrai sauf 0.
3. Toute référence est vraie.
4. Toute valeur indéfinie est fausse.

En fait, les deux dernières règles peuvent être dérivées des deux premières. Toute référence (règle 3) pointe vers quelque chose avec une adresse, et donne un nombre ou une chaîne contenant cette adresse, qui n'est jamais nulle. Et toute valeur indéfinie (règle 4) donne 0 ou la chaîne nulle.

Et d'une certaine manière, il est possible de faire dériver la règle 2 de la règle 1 si l'on pose que tout est une chaîne. Encore une fois, aucun transtypage n'est effectué pour évaluer la vérité, mais même si c'était le cas, une valeur numérique de 0 donnerait simplement la chaîne `"0"` et serait fausse. Tout autre nombre donnerait autre chose et serait vrai. Quelques exemples nous permettront de saisir un peu mieux ce que cela implique :

```
0          # deviendrait la chaîne "0", donc faux
1          # deviendrait la chaîne "1", donc vrai
10 - 10    # 10-10 vaut 0, deviendrait la chaîne "0", donc faux
0.00       # devient 0, et donc la chaîne "0", donc faux
"0"        # la chaîne "0", donc faux
""         # une chaîne nulle, donc faux
"0.00"     # la chaîne "0.00", ni vide, ni vraiment "0", donc vrai
"0.00" + 0 # le nombre 0 (transtypé par +), donc faux
\$a        # une référence à $a, donc vrai, même si $a est faux
undef()    # une fonction renvoyant la chaîne indéfinie, donc faux
```

Comme nous avons déjà dit que la vérité était évaluée dans un contexte scalaire, on peut se demander quelle serait la valeur d'une liste. En fait, il n'existe *pas* d'opération renvoyant une liste dans un contexte scalaire. Toutes renvoient une valeur scalaire et il suffit d'appliquer les règles de vérité à ce scalaire. Il n'y a donc pas de problème, tant que l'on sait ce qu'un opérateur donné renvoie dans un contexte scalaire.

21. À dire vrai, ce n'est pas tout à fait exact.

Les instructions if et unless

Nous avons vu plus haut comment un opérateur logique pouvait fonctionner comme un conditionnel. Une forme un peu plus complexe des opérateurs logiques est l'instruction `if`. Celui-ci évalue une condition de vérité et exécute un bloc si la condition est vraie.

```
if ($debug_level > 0) {
    # Quelque chose ne va pas. Il faut avertir l'utilisateur.
    print "Debug: Danger, Arthur Accroc, danger!\n";
    print "Debug: La réponse est '54' au lieu de '42'.\n";
}
```

Un bloc consiste en une ou plusieurs instructions regroupées par un ensemble d'accolades. Puisque l'instruction `if` exécute un bloc, les accolades sont nécessaires par définition. Dans un langage comme le C, on constate une certaine différence. Les accolades sont optionnelles s'il n'y a qu'une ligne de code, ce qui n'est pas le cas en Perl.

Parfois, il ne suffit pas d'exécuter un bloc quand une condition est remplie. On désire également exécuter un autre bloc quand la condition n'est *pas* remplie. Bien que l'on puisse évidemment employer deux instructions `if`, l'une étant la négation de l'autre, Perl fournit une solution plus élégante. Après le bloc, `if` peut prendre une deuxième condition optionnelle, appelée `else`, qui n'est exécutée que si la conditionnelle est fausse (les vétérans de l'informatique n'en seront pas surpris).

En d'autres occasions, vous pouvez même avoir plus de deux choix possibles. En ce cas, il est possible d'ajouter une conditionnelle `elsif` pour les autre choix possibles (les vétérans de l'informatique s'étonneront à bon droit de la façon d'écrire « `elsif` », mais personne ici ne s'en excusera).

```
if ($city eq "Lille") {
    print "Lille est au nord de Paris.\n";
}
elsif ($city eq "Nancy") {
    print "Nancy est à l'est de Paris.\n";
}
elsif ($city eq "Bayonne") {
    print "Bayonne est au sud-ouest de Paris.  Et il y fait plus chaud!\n";
}
else {
    print "Je ne sais pas où se trouve $city, désolé.\n";
}
```

Les clauses `if` et `elsif` sont calculées chacune à leur tour, jusqu'à ce que l'une d'entre elles soit vraie ou que la condition `else` soit atteinte. Quand l'une de ces conditions est vraie, son bloc est exécuté et toutes les branches restantes sont sautées. Parfois, on ne veut faire quelque chose que si la condition est fausse, et rien si elle est vraie. Un `if` vide avec un `else` n'est pas très propre, et la négation d'un `if` peut être illisible ; il est dommage d'écrire « faire quelque chose si pas ceci est vrai ». Dans ce cas, on peut utiliser l'instruction `unless`.

```
unless ($destination eq $maison) {
    print "Je ne rentre pas à la maison.\n";
}
```

Il n'existe pas de `elsunless`. Il s'agit là d'une caractéristique.

Constructions itératives (en boucle)

Perl possède quatre types principaux d'instructions itératives : `while`, `until`, `for` et `foreach`. Ces instructions permettent à un programme Perl d'exécuter de façon répétitive le même code pour différentes valeurs.

Les instructions while et until

Les instructions `while` et `until` fonctionnent de la même manière que les instructions `if` et `unless`, sauf qu'ils répètent l'exécution du bloc en bouclant. D'abord, la partie conditionnelle d'une instruction est vérifiée. Si la condition est remplie (si elle est vraie pour un `while` ou fausse pour un `until`), le bloc de l'instruction est exécuté.

```
while ($tickets_vendus < 10000) {
    $dispo = 10000 - $tickets_vendus;
    print "$dispo tickets sont disponibles. Combien en voulez -vous: ";
    $achat = <STDIN>;
    chomp($achat);
    $tickets_vendus += $achat;
}
```

Remarquez que si la condition originelle n'est pas remplie, on ne rentrera jamais dans la boucle. Par exemple, si nous avons déjà vendu 10 000 tickets, nous voulons que la ligne suivante du programme affiche quelque chose comme :

```
print "Ce spectacle est complet, revenez plus tard.\n";
```

Dans notre exemple de notation, on voit à la ligne 4 :

```
while ($ligne = <NOTES>) {
```

ce qui assigne la ligne suivante à la variable `$ligne`, et comme nous l'avons expliqué plus haut, renvoie la valeur de `$ligne` afin que la condition de l'instruction `while` puisse évaluer la vérité de `$ligne`. On peut se demander si Perl obtient un « faux »\ sur les lignes vides et sort prématurément de la boucle. La réponse est non. La raison en est simple, si vous vous reportez à tout ce que nous en avons dit. L'opérateur d'entrée de ligne laisse le saut de ligne à la fin de la chaîne, ce qui fait qu'une ligne vide comporte la valeur `"\n"`. Et vous savez que `"\n"` n'est pas une des valeurs canoniques de fausseté. La condition est donc vraie et la boucle continue même pour les lignes vides.

En revanche, quand nous atteignons la fin du fichier, l'opérateur d'entrée de ligne renvoie la valeur indéfinie, qui donne toujours faux. Et la boucle se termine ainsi au moment désiré. Ce programme Perl n'a pas besoin d'un test explicite sur la fonction `eof`, parce les opérateurs d'entrée sont conçus pour fonctionner sans douleur dans un contexte conditionnel.

En fait, presque tout est conçu pour fonctionner sans douleur dans un contexte conditionnel. Par exemple, un tableau dans un contexte scalaire renvoie sa longueur. On voit donc fréquemment ce genre de choses :

```
while (@ARGV) {
    process(shift @ARGV);
}
```

La boucle sort automatiquement quand `@ARGV` est épuisé. L'opérateur `shift` retire un élément de la liste argument à chaque boucle et retourne cet élément. La boucle s'arrête

automatiquement lorsque le tableau `@ARGV` est vide, donc de longueur 0 qui a pour valeur booléenne 0.[22]

L'instruction for

Un autre traitement itératif est la boucle `for`. Une boucle `for` tourne exactement comme la boucle `while`, mais paraît très différente (bien qu'elle dise quelque chose aux programmeurs C).

```
for ($vendu = 0; $vendu < 10000; $vendu += $achat) {
    $dispo = 10000 - $vendu;
    print "$dispo tickets sont disponibles.  Combien en voulez-vous: ";
    $achat = <STDIN>;
    chomp($achat);
}
```

La boucle `for` prend trois expressions entre les parenthèses de la boucle : une expression pour initialiser l'état de la variable de boucle, une condition pour tester la variable et une expression pour modifier l'état de la variable. Quand la boucle commence, la variable est initialisée et la condition est vérifiée. Si elle est vraie, le bloc est exécuté. Quand le bloc se termine, l'expression de modification est exécutée, la condition est de nouveau vérifiée et si elle est vraie, le bloc est réexécuté avec les nouvelles valeurs. Tant que la condition reste vraie, le bloc et l'expression de modification continuent à être exécutés. (Notez que seule la valeur de l'expression du milieu est évaluée et mémorisée, les deux autres ne sont utilisées que comme effet de bord, et ne sont pas mémorisées.)

L'instruction foreach

La dernière des principales instructions itératives est l'instruction `foreach`. `foreach` est utilisée pour exécuter le même code pour chacun des éléments d'un ensemble connu de scalaires, comme un tableau :

```
foreach $user (@users) {
    if (-f "$home{$user}/.nexrc") {
        print "$user est un type bien... il utilise un vi qui
               comprend Perl!\n";
    }
}
```

À la différence des conditionnelles `if` et `while`, qui induisent un contexte scalaire dans l'expression, l'instruction `foreach` induit un contexte de liste dans l'expression entre parenthèses. Ainsi l'expression s'évalue en une liste (même si cette liste ne contient qu'un scalaire). Chaque élément de la liste est tour à tour mis dans la variable de boucle et le bloc de code est exécuté une fois par élément. Remarquez que la variable de boucle devient une référence à l'élément lui-même, et non une copie de l'élément. La modification de cette variable modifie donc le tableau originel.

---

22. C'est ainsi qu'on raisonne en Perl. Il n'est pas nécessaire de comparer 0 avec 0 pour savoir que c'est faux. Si certains langages vous y obligent, ne vous égarez pas à faire la comparaison explicite `while (@ARGV != 0)`. C'est une perte de temps pour vous et le calculateur, aussi bien que pour la maintenance.

On trouve beaucoup plus de boucles `foreach` dans un programme Perl typique que dans des boucles `for`, car il est très facile en Perl de générer les listes que demande `foreach`. Une tournure que l'on rencontre fréquemment est une itération bouclant sur les clefs triées d'un hachage :

```
foreach $clef (sort keys %hash) {
```

Ce qui est exactement le cas de la ligne 9 de notre exemple de notation.

Pour s'en sortir : next et last

Les opérateurs `next` et `last` permettent de modifier le flux de votre boucle. Il n'est pas rare de rencontrer un cas spécial ; on peut vouloir le sauter, ou quitter la boucle quand on le rencontre. Par exemple, si l'on gère des comptes UNIX, on peut vouloir sauter les comptes système comme *root* ou *lp*. L'opérateur `next` permet de sauter à la fin de l'itération courante et d'en commencer une nouvelle. L'opérateur `last` permet de sauter à la fin du bloc comme si la condition du test avait renvoyé faux, par exemple dans le cas où l'on cherche un compte spécifique et que l'on veut quitter aussitôt qu'on l'a trouvé.

```
foreach $user (@users) {
    if ($user eq "root" or $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Compte spécial trouvé.\n";
        # Traitement
        last;
    }
}
```

Il est possible de sortir de boucles imbriquées en les étiquetant et en spécifiant celles dont on veut sortir. Les modificateurs d'instruction (une autre forme de conditionnelle dont nous n'avons pas encore parlé) associés à ceci peuvent fournir des sorties de boucles très lisibles, pour autant que le français ou l'anglais soient lisibles :

```
LIGNE: while ($ligne = <ARTICLE>) {
    last LIGNE if $ligne eq "\n";  # arrêt sur la première ligne vide
    next LIGNE if /^#/;            # sauter les lignes de commentaire
    # ici, votre publicité
}
```

Vous devez vous dire, « Une minute, là, qu'est ce que ce truc bizarre, `^#`, entre ces espèces de cure-dents ? Cela ne ressemble pas à du langage naturel. ». Vous avez mille fois raison. Il s'agit d'une recherche de correspondance contenant une expression rationnelle (bien qu'elle soit plutôt simple). Et c'est ce dont parle la section suivante. Perl est avant tout un langage de traitement de texte, et les expressions rationnelles sont au cœur de ces fonctionnalités.

Expressions régulières

Les *expressions régulières* (aussi expressions rationnelles, regexps ou bien RE) sont utilisées par la plupart des processeurs de texte du monde UNIX : *grep* et *findstr*, *sed* et *awk*,

et les éditeurs tels *vi* et *emacs*. Une expression régulière est l'écriture concise d'un ensemble de chaînes.[23]

La plupart des autres langages intègrent aussi le traitement des RE, mais aucun d'eux ne le fait à la manière de Perl. Les expressions rationnelles sont employées de plusieurs manières dans Perl. Elles sont utilisées en premier lieu dans des conditionnelles pour déterminer si une chaîne correspond à un motif donné. Quand on voit quelque chose qui ressemble à `/machin/`, on sait qu'il s'agit d'un opérateur de *recherche de correspondance* ordinaire.

```
if (/Windows 95/) { print "Il est temps de faire une mise à jour ?\n" }
```

Ensuite, si l'on peut retrouver des motifs dans une chaîne, on peut les remplacer par quelque chose d'autre. Par exemple, `s/machin/bidule/` demande de substituer « bidule » à « machin », si cela est possible. Nous appelons cela l'opérateur de *substitution*. Une RE peut aussi s'évaluer en un booléen, mais c'est l'effet de bord qui est utilisé la plupart du temps.

```
s/Windows/Linux/;
```

Enfin, les motifs peuvent non seulement spécifier l'emplacement de quelque chose, mais également les endroits où il ne se trouve pas. L'opérateur `split` emploie donc une expression rationnelle pour spécifier les emplacements d'où les données sont absentes. Autrement dit, l'expression rationnelle définit les délimiteurs qui séparent les champs de données. Notre exemple de notation en comporte quelques exemples simples. Les lignes 5 et 12 éclatent les chaînes sur le caractère espace pour renvoyer une liste de mots. Mais `split` permet d'éclater sur tout délimiteur spécifié par une expression rationnelle.

```
($bon, $brute, $truand) = split(/,/, "vi,emacs,teco");
```

Il existe de nombreux modificateurs utilisables dans chacun de ces contextes pour faire des choses plus exotiques comme l'indifférenciation des majuscules et des minuscules en recherchant des chaînes de caractères, mais ce sont là des détails que nous couvrirons au chapitre suivant.

L'usage le plus simple des expressions rationnelles consiste à rechercher une expression littérale. Dans le cas des éclatements que nous venons de mentionner, nous avons recherché un simple espace. Mais si l'on recherche plusieurs caractères à la fois, ils doivent tous correspondre en séquence. C'est-à-dire que le motif correspond à une sous-chaîne, comme l'on peut s'y attendre. Admettons que nous désirions montrer toutes les lignes d'un fichier HTML représentant des liens vers d'autres fichiers HTML (en excluant les liens FTP). Imaginons que nous travaillions en HTML pour la première fois et que nous soyons donc un peu naïfs. Nous savons que ces liens contiendront toujours la chaîne « `http:` ». Nous pouvons boucler dans le fichier comme ceci :[24]

---

23. Une bonne introduction est le livre de Jeffrey Friedl, *Maîtrise des expressions régulières* (Éditions O'Reilly).

24. Cela ressemble beaucoup à la commande UNIX `grep 'http:' fichier`. Sous MS-DOS, il est possible d'employer la commande *find*, mais elle ne sait pas traiter d'expression plus complexe. (Cependant, le programme — improprement nommé — *findstr* de Windows NT comprend les expressions rationnelles.)

```
while ($ligne = <FICHIER>) {
    if ($ligne =~ /http:/) {
        print $ligne;
    }
}
```

Ici, le =~ (opérateur de liaison de motif) indique à Perl qu'il doit rechercher l'expression rationnelle `http:` dans la variable `$ligne`. S'il trouve l'expression, l'opérateur renvoie une valeur vraie et le bloc (une commande d'affichage) est exécuté.

Au fait, si l'on n'utilise pas l'opérateur de liaison =~, Perl recherchera un motif par défaut au lieu de `$ligne`. Ce motif par défaut n'est en fait qu'une variable spéciale qui prend le nom curieux de `$_`. En fait, de nombreux opérateurs emploient la variable `$_` et un expert de la programmation Perl peut écrire ce qui précède comme :

```
while (<FICHIER>) {
    print if /http:/;
}
```

(Hmm, un autre modificateur d'instruction semble être apparu ici. Insidieuses bestioles...)

Tout cela est bien pratique, mais si nous voulons trouver tous les liens, et non les seuls HTTP ? Nous pouvons en donner une liste comme « `http:` », « `ftp:` », « `mailto:` », etc. Mais cette liste peut être longue, et que faire si un nouveau type de lien est ajouté ?

```
while (<FILE>) {
    print if /http:/;
    print if /ftp:/;
    print if /mailto:/;
    # Et après?
}
```

Comme les expressions rationnelles décrivent un ensemble de chaînes, nous pouvons nous contenter de décrire ce que nous cherchons : un certain nombre de caractères alphabétiques suivis d'un deux-points. Dans le dialecte des expressions rationnelles (le regexpais ?), ce serait `/[a-zA-Z]+:/`, où les crochets définissent une *classe de caractères*. Les `a-z` et `A-Z` représentent tous les caractères alphabétiques (le tiret représentant l'intervalle de caractères entre le caractère de début et celui de fin, inclus). Et le `+` est un caractère spécial disant « un ou plusieurs exemplaires du machin qui se trouve avant moi ». Il s'agit d'un *quantificateur*, qui indique combien de fois quelque chose peut être répété. (Les barres obliques ne font pas vraiment partie de l'expression rationnelle, mais plutôt de l'opérateur de correspondance. Elles n'agissent que comme délimiteurs de l'expression rationnelle).

Certaines classes, comme la classe alphabétique, étant fréquemment utilisées, Perl définit des cas spéciaux, qui comprennent :

Nom	Définition ASCII	Caractère
espace	`[ \t\n\r\f]`	`\s`
Caractère de mot	`[a-zA-Z_0-9]`	`\w`
Chiffre (digit)	`[0-9]`	`\d`

Remarquez qu'ils ne correspondent qu'à des caractères *simples*. Un `\w` recherchera un unique caractère de mot et non un mot complet. (Vous souvenez-vous du quantificateur `+` ? On peut écrire `\w+` pour rechercher un mot.) Perl fournit également la négation de ces classes en employant le caractère en majuscule, comme `\D` pour un caractère qui n'est pas un chiffre.

(Il faut remarquer que `\w` n'est pas toujours équivalent à `[a-zA-Z_0-9]`. Certains *locales* définissent des caractères alphabétiques supplémentaires hors de la séquence ASCII, et `\w` les respecte.)[25] Certaines langues définissent des caractères alphabétiques au-delà de la séquence ASCII classique. Le code `\w` les prend en compte. Les récentes versions de Perl connaissent aussi le codage UNICODE avec les propriétés numériques et Perl traite ce codage avec les propriétés en conséquence. (Il considère aussi les idéogrammes comme des caractères `\w`.)

Il existe une autre classe de caractères très spéciale, que l'on écrit « . », qui recherchera tout caractère.[26] Par exemple, `/a./` trouve toute chaîne contenant un « `a` » qui n'est pas le dernier caractère de la chaîne. Il trouvera donc « `at` » ou « `am` », ou même « `a+` », mais non « `a` » puisqu'il n'y a rien après le « `a` » qui puisse correspondre au point. Comme il cherche le motif n'importe où dans la chaîne, il le trouvera dans « `oasis` » et dans « `chameau` », mais non dans « `sheba` ». Il trouvera le premier « `a` » de « `caravane` ». Il pourrait trouver le deuxième, mais il s'arrête après la première correspondance en cherchant de gauche à droite.

Quantificateurs

Les caractères et les classes de caractères dont nous avons parlé recherchent des caractères uniques. Nous avons déjà mentionné que l'on pouvait rechercher plusieurs caractères « de mot » avec `\w+` pour correspondre à un mot complet. Le `+` est un de ces quantificateurs, mais il y en a d'autres (tous sont placés après l'élément quantifié).

La forme la plus générale de quantificateur spécifie le nombre minimal et le nombre maximal de fois auquel un élément peut correspondre. Les deux nombres sont mis entre accolades, séparés par une virgule. Par exemple, pour essayer de trouver les numéros de téléphone d'Amérique du Nord, `/\d{7,11}/` rechercherait au moins 7 chiffres, mais au plus 11 chiffres. Si un seul chiffre se trouve entre les accolades, il spécifie à la fois le minimum et le maximum ; c'est-à-dire que le nombre spécifie le nombre exact de fois que l'élément peut être répété. (Si l'on y réfléchit, tous les éléments non quantifiés ont un quantificateur implicite `{1}`.)

Si l'on met le minimum et la virgule en omettant le maximum, ce dernier passe à l'infini. En d'autres termes, le quantificateur indique le nombre minimum de caractères, et tout ceux qu'il trouvera après cela. Par exemple, `/\d{7}/` ne trouvera qu'un numéro de téléphone local (d'Amérique du Nord, à sept chiffres), alors que `/\d{7,}/` correspondra à tous les numéros de téléphones, même les internationaux (sauf ceux qui comportent moins de 7 chiffres). Il n'existe pas de façon spécifique d'écrire « au plus » un certain nombre de caractères. Il suffit d'écrire, par exemple, `/.{0,5}/` pour trouver au plus cinq caractères quelconques.

---

25. Ce qui est bien pratique pour rechercher les caractères accentués en français. (N.d.T.)

26. Mais il ne trouvera pas un saut de ligne. Quand on y pense, un « . » ne correspond pas non plus à un saut de ligne dans *grep*(1).

Certaines combinaisons de minimum et de maximum apparaissant fréquemment, Perl définit des quantificateurs spéciaux. Nous avons déjà vu +, qui est identique à {1,}, c'est-à-dire « au moins une occurrence de l'élément qui précède ». Il existe aussi *, qui est identique à {0,}, ou « zéro occurrence ou plus de l'élément qui précède », et ?, identique à {0,1}, ou « zéro ou une occurrence de l'élément qui précède » (l'élément qui précède est donc optionnel).

Il faut savoir deux ou trois choses concernant la quantification. D'abord, les expressions rationnelles de Perl sont, par défaut, *avides*. Cela signifie qu'elles essayent de trouver une correspondance tant que l'expression entière correspond toujours. Par exemple, si l'on compare /\d+ à « 1234567890 », cela correspondra à la chaîne entière. Il faut donc spécialement surveiller l'emploi de « . », tout caractère. On trouve souvent une chaîne comme :

```
larry:JYHtPhO./NJTU:1OO:1O:Larry Wall:/home/larry:/bin/tcsh
```

où on essaye de trouver « larry » avec /.+:/. Cependant, comme les expressions rationnelles sont avides, ce motif correspond à tout ce qui se trouve jusqu'à « /home/larry » inclus. On peut parfois éviter ce comportement en utilisant une *négation* de classe de caractères, par exemple avec /[\^:]+:/, qui indique de rechercher un ou plusieurs caractères *non*-deux-points (autant que possible), jusqu'au premier deux-points. C'est le petit chapeau qui inverse le sens de la classe de caractères.[27] L'autre point à surveiller est que les expressions rationnelles essayent de trouver une correspondance *dès* que possible. Ce point est même plus important que l'avidité. Le balayage se produisant de gauche à droite, cela veut dire que le motif cherchera la correspondance la plus à gauche possible, même s'il existe un autre endroit permettant une correspondance plus longue (les expressions rationnelles sont avides, mais elles voient à court terme). Par exemple, supposons que l'on utilise la commande de substitution s/// sur la chaîne par défaut (à savoir la variable $_), et que l'on veuille enlever une suite de *x* du milieu de la chaîne. Si l'on écrit :

```
$_ = "fred xxxxxxx barney";
s/x*//;
```

cela n'aura aucun effet. En effet, le x* (voulant dire zéro caractères « x » ou plus) trouvera le « rien » au début de la chaîne, puisque la chaîne nulle est large de zéro caractères et que l'on trouve une chaîne nulle juste avant le « f » de « fred ».[28]

Encore une chose à savoir. Par défaut, les quantificateurs s'appliquent à un caractère unique les précédant, ce qui fait que /bam{2}/ correspond à « bamm » mais pas à « bambam ». Pour appliquer un quantificateur à plus d'un caractère, utilisez les parenthèses. Il faut employer le motif /(bam){2}/ pour trouver « bambam ».

Correspondance minimale

Si l'on ne voulait pas de « correspondances avides » dans les anciennes versions de Perl, il fallait employer la négation d'une classe de caractères (et en fait, on obtenait encore une correspondance avide, bien que restreinte).

27. Désolé, nous n'avons pas inventé cette notation, ne nous en voulez pas. Il s'agit simplement de la façon d'écrire les expressions rationnelles dans la culture UNIX.

28. Même les auteurs s'y font prendre de temps en temps.

Les versions modernes de Perl permettent de forcer une correspondance minimale par l'emploi d'un point d'interrogation après tout quantificateur. Notre recherche de nom d'utilisateur serait alors `/.*?:/`. Ce `.*?` essaye alors de correspondre à aussi *peu* de caractères que possible, au lieu d'autant que possible, et il s'arrête donc au premier deux-points au lieu du dernier.

Pour enfoncer le clou

Chaque fois que l'on cherche à faire correspondre un motif, celui-ci essayera partout jusqu'à ce qu'il trouve une correspondance. Une *ancre* permet de restreindre l'espace de recherche. Une ancre est d'abord quelque chose qui ne correspond à « rien », mais il s'agit là d'un rien d'un type spécial qui dépend de son environnement. On peut aussi l'appeler une règle, une contrainte ou une assertion. Quelle que soit son appellation, elle essaie de faire correspondre quelque chose de longueur zéro, et réussit ou échoue. (En cas d'échec, cela veut surtout dire que le motif ne trouve pas de correspondance de cette façon. Il essaie alors d'une autre manière, s'il en existe.)

La chaîne de caractères spéciale `\b` correspond à une limite de mot, qui est définie comme le « rien » entre un caractère de mot (`\w`) et un caractère de non-mot (`\W`), dans le désordre. (Les caractères qui n'existent pas de part et d'autre de la chaîne sont appelés des caractères de non-mot). Par exemple,

```
/\bFred\b/
```

correspond à « `Le Grand Fred` » et à « `Fred le Grand` » , mais ne correspond pas à « `Frederic le Grand` » parce que le « `de` » de Frederic ne contient pas de limite de mot.

Dans une veine similaire, il existe également des ancres pour le début et la fin d'une chaîne. S'il s'agit du premier caractère d'un motif, le chapeau (`^`) correspond au « rien » au début de la chaîne. Le motif `/^Fred/` correspondrait donc à « `Frederic le Grand` » et non à « `Le Grand Fred` », alors que `/Fred^/` ne correspondrait à aucun des deux (et ne correspondrait d'ailleurs pas à grand-chose). Le signe dollar (`$`) fonctionne comme le chapeau, mais il correspond au « rien » à la fin de la chaîne au lieu du début.[29]

Vous devez maintenant être en mesure de deviner ce que veut dire :

```
next LIGNE if /^#/;
```

C'est bien sûr « Aller à la prochaine itération de la boucle `LIGNE` si cette ligne commence par un caractère # ».

Nous avons dit plus haut que la séquence `\d{7,11}` coïncide avec un nombre de 7 à 11 chiffres de long ce qui est vrai mais incomplet: quand cette séquence est utilisée dans une expression comme `/\d{7,11}/`, ça n'exclue pas d'autres chiffres après les 11 premiers. Donc, le plus souvent, lorsque vous utilisez des quantificateurs, vous utiliserez des ancres de chaque côté.

29. Tout ceci est un peu simpliste, car nous supposons ici que la chaîne ne contient qu'une ligne. ^ et $ sont en fait des ancres au début et en fin de ligne et non pas de chaîne. Nous essaierons de clarifier tout cela au chapitre 5, *Recherche de motif* (pour autant que l'on puisse clarifier tout cela).

Références arrières

Nous avons déjà mentionné que l'on pouvait employer des parenthèses pour grouper des caractères devant un quantificateur, mais elles peuvent aussi servir à se souvenir de parties de ce qui a été trouvé. Une paire de parenthèses autour d'une partie d'expression rationnelle permet de se souvenir de ce qui a été trouvé pour une utilisation future. Cela ne change pas le motif de recherche, et `/\d+` et `/(\d+)/` chercheront toujours autant de chiffres que possibles, mais dans le dernier cas, ceux-ci seront mémorisés dans une variable spéciale pour être « référencés en arrière » plus tard.

La façon de référencer la partie mémorisée de la chaîne dépend de l'endroit d'où on opère. Dans la même expression rationnelle, on utilise un antislash suivi d'un entier. Il correspond à une paire de parenthèses donnée, déterminée en comptant les parenthèses gauches depuis la gauche du motif, en commençant par un. Par exemple, pour rechercher quelque chose ressemblant à une balise HTML (comme « `<B>Gras</B>` »), on peut utiliser `/<(.*?)>.*?<\/\1>/`. On force alors les deux parties du motif à correspondre exactement à la même chaîne de caractères, comme au « `B` » ci-dessus.

Hormis l'expression rationnelle elle-même, comme dans la partie de remplacement d'une substitution, la variable spéciale est utilisée comme s'il s'agissait d'une variable scalaire normale nommée par l'entier. Donc, si l'on veut échanger les deux premiers mots d'une chaîne, par exemple, on peut employer :

```
s/(\S+)\s+(\S+)/$2 $1/
```

Le côté droit de la substitution équivaut à une sorte de chaîne protégée par des apostrophes doubles, ce qui explique pourquoi l'on peut y interpoler des variables, y compris des références arrières. Ce concept est puissant : l'interpolation (sous contrôle) est une des raisons pour lesquelles Perl est un bon langage de traitement de texte. Entre autres raisons, on trouve bien entendu les motifs de correspondance. Les expressions rationnelles savent parfaitement extraire des éléments et l'interpolation sert à les rassembler. Peut-être existe-t-il enfin un espoir pour Humpty Dumpty.

Traitement des listes

Nous avons vu plus haut que Perl connaît deux principaux contextes, le contexte scalaire (pour gérer les singularités) et le contexte de liste (pour gérer les pluralités). La plupart des opérateurs traditionnels que nous avons décrits jusqu'ici étaient strictement scalaires dans leur opération. Ils prennent toujours des arguments singuliers (ou des paires d'arguments singuliers pour les opérateurs binaires), et produisent toujours un résultat singulier, même dans un contexte de liste. Donc, si l'on écrit ceci :

```
@tableau = (1 + 2, 3 - 4, 5 * 6, 7 / 8);
```

on sait que la liste de droite contient exactement quatre valeurs, car les opérateurs mathématiques ordinaires produisent toujours des valeurs scalaires, même dans le contexte de liste fourni par l'assignation à un tableau.

Cependant, d'autres opérateurs Perl peuvent produire soit un scalaire, soit une liste, selon le contexte. Ils « savent » si vous attendez d'eux un scalaire ou une liste. Mais comment pouvez-*vous* le savoir ? C'est en fait très simple une fois que vous vous êtes familiarisés avec quelques concepts clefs.

Premièrement, le contexte de liste doit être fourni par quelque chose dans l'« environnement ». Dans l'exemple ci-dessus, c'est l'assignation de liste qui le fournit. Nous avons vu ci-avant que la liste dans l'instruction de boucle `foreach` fournit le contexte de liste. L'opérateur `print` aussi. Il est possible de tous les identifier d'un coup.

Si vous regardez les nombreuses représentations syntaxiques des opérateurs dans le reste de ce livre, divers opérateurs sont définis pour prendre une *LISTE* en argument. Ce sont les opérateurs qui *fournissent* un contexte de liste. Tout au long de ce livre, *LISTE* est employé comme le terme technique signifiant « une construction syntaxique fournissant un contexte de liste ». Par exemple, si l'on regarde `sort`, on trouve le résumé de syntaxe suivant :

```
sort LISTE
```

Ce qui veut dire que `sort` procure un contexte de liste à ses arguments.

Deuxièmement, à la compilation, tout opérateur qui prend une *LISTE* fournit un contexte de liste à *chaque* élément syntaxique de cette *LISTE*. Chaque opérateur ou entité de haut niveau de la *LISTE* sait qu'il est censé fournir la meilleure liste possible. Cela signifie que si l'on écrit :

```
sort @mecs, @nanas, autres();
```

`@mecs`, `@nanas` et `autres()` savent tous qu'ils sont censés fournir une valeur de liste.

Enfin, à l'exécution, chacun de ces éléments de *LISTE* fournit sa liste, puis (ceci est important) toutes les listes isolées sont rassemblées, bout à bout, en une liste unique. Et cette liste unique, unidimensionnelle, est ce qui est finalement donné à la fonction qui voulait une *LISTE*. Si donc `@mecs` contient `(Fred, Barney)`, `@nanas` contient `(Wilma, Betty)`, et que la fonction `autre()` renvoie la liste à un seul élément `(Dino)`, la *LISTE* que voit `sort` devient :

```
(Fred,Barney,Wilma,Betty,Dino)
```

et `sort` renvoie la liste :

```
(Barney,Betty,Dino,Fred,Wilma)
```

Certains opérateurs produisent des listes (comme `keys`), certains les consomment (comme `print`) et d'autres les transforment en d'autres listes (comme `sort`). Les opérateurs de cette dernière catégorie sont considérés comme des *filtres* ; mais contrairement au shell, le flux de données s'effectue de droite vers la gauche, puisque les opérateurs de liste agissent sur les éléments qui sont passés à leur droite. On peut empiler plusieurs opérateurs de liste à la fois :

```
print reverse sort map {lc} keys %hash;
```

Toutes les clefs de `%hash` sont prises et renvoyées à la fonction `map`, qui met chacune d'entre elles en minuscule par l'opérateur `lc`, et les passe à la fonction `sort` qui les trie et les repasse à la fonction `reverse`, qui renverse l'ordre des éléments de la liste, qui les passe enfin à la fonction `print`, qui les affiche.

C'est, on le voit, beaucoup plus facile à décrire en Perl qu'en français.

Nous ne pouvons lister tous les exemples où l'utilisation de la structure de liste produit un code plus naturel à lire. Mais revenons sur les RE un moment. Dans un contexte de liste, toutes les coïncidences sont mémorisées dans les éléments successifs de la liste. Cherchons, par exemple, toutes les chaînes de la forme « 12:59:59 am ». On peut le faire comme ceci :

```
($hour, $min, $sec, $ampm) = /(\d+):(\d+):(\d+) *(\w+)/;
```

Ce qui est une façon pratique d'instancier plusieurs variables en même temps. On peut aussi écrire :

```
@hmsa = /(\d+):(\d+):(\d+) *(\w+)/;
```

et mettre ces 4 valeurs dans un tableau. Étrangement, par la séparation de l'action entre les RE et les expressions Perl, le contexte de liste augmente les possibilités du langage. Nous ne le voyons pas, mais Perl est un véritable langage orthogonal en plus d'être diagonal. Allez, on fait une pause !

Ce que vous ne savez pas ne vous fera pas (trop) de mal

Enfin, permettez-nous de revenir encore une fois au concept de Perl en tant que langage naturel. Les usagers d'un langage naturel ont le droit d'avoir des niveaux d'aptitude différents, de parler différents sous-ensembles de ce langage, d'apprendre au fur et à mesure et, en général, de l'utiliser avant d'en connaître toutes les arcanes. Vous ne connaissez pas tout de Perl, tout comme vous ne connaissez pas tout du français. Mais ceci est Officiellement OK dans la culture Perl. Vous pouvez employer utilement Perl même si nous ne vous avons pas encore appris à écrire vos propres sous-programmes. Nous avons à peine commencé à expliquer que Perl pouvait être vu comme un langage de gestion système, ou comme un langage de prototypage rapide, ou comme un langage réseau, ou comme un langage orienté objet. Nous pourrions écrire des chapitres entiers sur ces sujets (et à vrai dire, c'est déjà le cas).

Mais au bout du compte, vous devrez créer votre propre vision de Perl. C'est votre privilège, en tant qu'artiste, de vous infliger à vous-même la douleur de la créativité. Nous pouvons vous apprendre *notre* façon de peindre, mais nous ne pouvons pas vous apprendre *votre* façon. TMTOWTDI : il existe plus d'une façon de faire.

Amusez-vous comme il convient.

II

Séance de dissection

2

Composants de Perl

Où nous étudions les éléments de l'ensemble.

Dans les prochains chapitres nous progresserons du particulier vers le général, approche ascendante, en commençant par décrire les composants élémentaires qui permettront d'aborder les structures plus élaborées, un peu à la manière dont les atomes forment les molécules. L'inconvénient est que vous n'en aurez pas nécessairement une image globale sans auparavant avoir été submergé par de nombreux détails, mais l'avantage est que vous comprendrez les exemples au fur et à mesure. (si vous préférez l'approche inverse, retournez le livre et lisez ce chapitre à l'envers).

Chaque chapitre est construit sur le précédent ce qui nécessite une lecture linéaire, (vous devrez donc faire attention si vous êtes du genre à sautiller d'une page à l'autre).

Vous êtes invité à utiliser les différentes annexes à la fin du livre. (Ce qui n'est pas du sautillement.) Chaque mot distingué par une police `type` se trouve au chapitre 29, *Fonctions*. Bien que nous avons essayé de ne pas privilégier un système d'exploitation, si vous êtes peu au courant de la terminologie Unix et si vous rencontrez un mot qui semble dire autre chose que ce que vous pensez, reportez-vous au glossaire. Si cela ne marche pas, essayez l'index.

Atomes

Le plus petit élément visible du langage est le caractère, celui que l'on peut visualiser dans un éditeur par exemple. Au départ, Perl ne faisait pas la distinction entre octets et caractères ASCII, mais pour des raisons d'internationalisation, il faut bien distinguer les deux.

Le code Perl peut être écrit exclusivement en ASCII, mais autorise aussi l'utilisation d'un codage sur 8 ou 16 bits, qu'il soit celui d'une langue ou bien de tout autre codage défini régulièrement, à condition de le faire dans des chaînes litérales uniquement. Bien sûr, Perl ne vérifiera pas le contenu du texte mais saura simplement qu'un caractère est codé sur 16 bits.

Comme expliqué au chapitre 15, *Unicode*, Perl implémente Unicode.[1] C'est transparent pour l'utilisateur du langage qui peut en utiliser aussi bien dans des identificateurs (noms de variables, etc) que dans des chaînes litérales. Pour Perl, tous les caractères ont la même taille de référence, par exemple 1, quelle que soit la représentation interne qu'il en fait ensuite. En principe, il code les caractères sous forme UTF-8 et donc un caractère peut avoir une forme interne sur 1, 2, 3... octets. (L'Unicode smiley U-263A est une séquence de 3 octets.)

Mais poursuivons l'analogie physique un peu plus avant avec les atomes. Les caractères sont atomiques de la même façon que les atomes codent les différents éléments. Un caractère est composé de bits et d'octets mais si on le désintègre (dans un accélérateur de caractères, sans doute), il perd toutes ses propriétés propres. De la même manière que les neutrons composent l'atome d'Uranium U-238, les octets qui composent le caractère smiley U-263A sont un détail de son implémentation.

Il faut donc bien distinguer le mot « caractère » du mot « octet ». C'est pourquoi on peut le préciser à l'interpréteur avec la directive `use bytes`. (Voir le chapitre 31, *Modules de pragmas*). De toute manière, Perl saura par lui-même faire le codage sur 8 bits quand il le faudra.

Maintenant passons dans la prochaine dimension.

Molécules

Perl est un langage *informel*, ce qui ne veut pas dire qu'il n'a pas de forme, mais plutôt dans le sens usuel du terme, c'est-à-dire un langage qu'on peut écrire avec des espaces, tabulations et caractères de nouvelle ligne partout où vous voulez sauf dans les unités syntaxiques.

Par définition, un symbole ne contient pas d'espace. Un *symbole* est une unité syntaxique avec un sens pour l'interpréteur, comme pour un mot dans une phrase, mais qui peut contenir d'autres caractères que des lettres tant que cette unité n'est pas rompue. (Ce sont de vraies molécules qui peuvent utiliser toutes sortes d'atomes. Par exemple, les nombres et les opérateurs mathématiques sont des symboles. Un *identificateur* est un symbole qui débute par une lettre ou un souligné et qui ne contient que des lettres, chiffres ou soulignés. Un symbole ne peut contenir d'espace puisque le caractère espace couperait le symbole en deux nouveaux symboles, tout comme un caractère espace, en français couperait un mot en deux nouveaux mots.[2]

Bien que le caractère espace soit nécessaire entre deux symboles, les espaces ne sont obligatoires qu'entre deux symboles qui, sinon, seraient pris pour un seul. À cet effet, tous les espaces sont équivalents. Un commentaire est compté comme un espace. Les sauts de ligne ne sont différents des espaces qu'entre des délimiteurs, dans certains formats et pour des formes orientées ligne de protection. Plus précisément le caractère de saut de ligne ne délimite pas une instruction comme en FORTRAN ou en Python. Les instructions en Perl se terminent par le caractère « `;` » comme en C.

1. Aussi enthousiaste que nous soyons au sujet d'Unicode, la plupart de nos exemples sont en ASCII, étant donné que tout le monde n'a pas nécessairement un éditeur de texte Unicode.

2. Dans les chaînes litérales, c'est le guillemet qui délimite l'unité et non l'espace, on peut donc y inclure ce caractère.

Les caractères d'espace Unicode sont autorisés dans un programme Perl Unicode moyennant certaines précautions. Si vous utilisez les caractères spéciaux Unicode séparateurs de paragraphe et de ligne, d'une manière différente que ne le fait votre éditeur de texte, les messages d'erreur seront plus difficile à interpréter. C'est mieux d'utiliser les caractères de saut de ligne classiques.

Les symboles sont détectés d'une manière large ; l'interpréteur Perl lit le symbole le plus long possible lors de la lecture du code. Si vous voulez qu'il distingue deux symboles il suffit d'insérer un espace blanc entre eux. (La tendance est d'insérer plusieurs espaces pour rendre le code plus lisible.)

Un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne. Un commentaire équivaut à un espace et joue le rôle de séparateur. Perl n'interprète pas une ligne placée en commentaire.[3]

Si une ligne, et c'est une autre originalité lexicale, commence par = à un endroit où une instruction serait légale, Perl ignore tout de cette ligne jusqu'à la prochaine contenant `=cut`. Le texte ignoré est supposé être du POD, ou *Plain Old Documentation* (des programmes Perl permettent de convertir ces commentaires au format des pages de manuel, ou en documents LaTeX, HTML et bientôt XML). D'une manière complémentaire, l'analyseur lexical extrait le code Perl d'un module et ignore les balises pod, ce qui permet de conserver la documentation à l'intérieur des modules. Voir le chapitre 26, *POD*, pour les détails sur pod et la documentation multiligne.

Mais seul le premier # est nécesaire, les autres ne sont que de la décoration, bien que la plupart des codeurs l'utilisent pour faire des effets visuels comme avec le langage C ou ils font une utilisation abondante du caractère *.

En Perl, comme en chimie ou en linguistique, on construit des structures de plus en complexes à partir d'éléments simples. Par exemple, une *instruction* est une séquence de symboles sur le mode impératif. On peut combiner une série d'instructions pour former un *bloc* délimité par des accolades. Les blocs peuvent eux-mêmes constituer d'autres blocs. Certains blocs fonctionnels tels que les *sous-programmes* peuvent être combinés en *modules* eux-mêmes combinés en *programmes*, nous verrons tout ceci dans les prochains chapitres. Mais créons encore des symboles avec nos caractères.

Types internes

Avant de construire des types plus complexes, nous devons introduire quelques abstractions, plus précisément trois types de données.

Chaque langage dispose de ses propres types de données et à la différence des autres, Perl en possède peu ce qui réduit les confusions possibles. Prenons par exemple le langage C qui dispose des types suivants : `char`, `short`, `int`, `long`, `long long`, `bool`, `wchar_t`, `size_t`, `off_t`, `regex_t`, `uid_t`, `u_longlong_t`, `pthread_key_t`, `fp_exception_field_type`, et ainsi de suite. Ce sont juste les types entiers! Il y a ensuite les nombres flottants, les pointeurs et les chaînes.

3. En fait ce n'est pas exact. L'analyseur lexical de Perl regarde s'il existe un sélecteur du type `#!` (voir le chapitre 19, *L'interface de la ligne de commande*). Il peut aussi analyser les numéros de ligne produits par différents prépocesseurs (voir la section *Générer du Perl dans d'autres langages* au chapitre 24, *Techniques couramment employées*).

Tous ces types compliqués n'en font qu'un en Perl : le scalaire. (Les types simples de données sont suffisants pour les besoins usuels, et il est possible de créer ses propres types en utilisant la programmation objet en Perl, voir le chapitre 12, *Objets*.) Les trois types de base en Perl sont : les *scalaires*, les *tableaux* de scalaires et les *hachages* de scalaires (connus aussi sous le nom de *tableaux associatifs*). On peut aussi les appeler *structures de données*.

Les scalaires constituent le type fondamental à partir duquel les structures plus complexes peuvent être construites. Un scalaire peut contenir une seule valeur simple, typiquement une chaîne ou un nombre. Les éléments de ce type simple peuvent être combinés à l'intérieur des deux autres types composés. Un tableau est une liste ordonnée de scalaires accessibles par un indice numérique (les indices démarrent à zéro). Contrairement aux autres langages, Perl traite les indices négatifs en comptant à partir de la fin du type indexé — aussi bien des chaînes que des listes. Un hachage est un ensemble *non ordonné* de paires clef/valeur accessibles par une clef de type chaîne utilisée pour accéder à la valeur scalaire associée. Les variables ont toujours l'un de ces trois types. (À part les variables, Perl comprend des bidules plus ou moins confidentiels tels que des handles de fichiers et de répertoires, des sous-programmes, des typeglobs, des formats et des tables de symboles.)

Laissons la théorie et passons à la pratique du langage pour en voir les possibilités. Nous allons donc écrire du code et pour cela vous présenter les termes constituant les expressions en Perl régies par des règles syntaxiques. Nous utilisons la terminologie *terme* quand nous parlerons des unités syntaxiques, un peu comme dans les équations mathématiques.

La fonction des termes en Perl est de fournir des valeurs pour des opérateurs tels l'addition ou la multiplication. Mais à la différence d'une équation, Perl interprète l'expression dans une action logique de la machine. Une des actions les plus courantes est la sauvegarde d'une valeur en mémoire :

```
$x = $y;
```

C'est un exemple de l'opérateur d'*affectation* (non pas l'opérateur de test d'égalité des valeurs numériques, ce qui s'écrit == en Perl). L'affectation prend la valeur de `$y` et la place dans la variable `$x`. Notez que le terme `$x` n'est pas utilisé pour sa valeur mais pour sa fonction de mémorisation. (L'ancienne valeur de `$x` disparaît avec l'affectation.) Nous disons que `$x` est une *lvalue*, parce qu'elle est cible d'une affectation à gauche d'une expression. Et nous disons que `$y` est une *rvalue* car à droite dans ce type d'expression.

Il y a un troisième type de valeur appelée *temporaire* qu'il est nécessaire de comprendre. Considérons l'expression simple suivante :

```
$x = $y + 1;
```

Perl prend la rvalue `$y` et lui ajoute la rvalue `1`, ce qui produit une valeur temporaire qui est ensuite affectée à la lvalue `$x`. Ces valeurs temporaires sont placées dans une structure interne appelée *pile*.[4]

4. Une pile fonctionne à la manière de ces piles d'assiettes dans les restaurants — on peut empiler (`push`) ou bien dépiler (`pop`) une assiette (le push et le pop étant deux termes bien établis dans la science informatique)

Les termes d'une expression empilent des valeurs, alors que ses opérateurs (dont nous parlerons dans le prochain chapitre) dépilent des valeurs et peuvent aussi empiler des résultats pour le prochain opérateur. Dans la pile, une expression produit autant de valeurs qu'elle en consomme. Nous reviendrons sur les valeurs temporaires.

Certains termes sont exclusivement des rvalues comme `1`, certains autres peuvent jouer les deux rôles, comme la variable de l'exemple précédent nous l'a montré. C'est ce que nous allons voir dans la prochaine section.

Variables

Il y existe trois types de variables correspondant aux trois types abstraits mentionnés plus haut. Chacun d'eux est préfixé par un *caractère spécial*.[5] Les variables scalaires sont toujours définies avec un caractère `$` initial, même si elles font référence à des scalaires à l'intérieur d'un tableau ou d'un hachage. Ce caractère fonctionne un peu comme l'article défini singulier (le ou la). Nous avons donc :

Construction	**Signification**
`$jours`	Valeur scalaire simple `$jours`.
`$jours[28]`	29e élément du tableau `@jours`.
`$jours{'Fev'}`	Valeur « `Fev` » du hachage `%jours`.

Noter que vous pouvez utiliser le même nom pour `$jours`, `@jours`, et `%jours`, Perl ne les confondra pas.

Il existe d'autres caractères spéciaux qui s'appliquent sur les scalaires et sont utilisés dans des cas précis. Ils ressemblent à ceci :

Construction	**Signification**
`${jours}`	Identique à `$jours` en levant l'ambiguïté devant les caractères alphanumériques.
`$Calendrier::jours`	Une autre variable `$jours`, dans le paquetage `Calendrier`.
`$#jours`	Dernier index du tableau `@jours`.
`$jours->[28]`	29e élément du tableau pointé par la référence `$jours`.
`$jours[0][2]`	Tableau multidimensionnel.
`$jours{2000}{'Fev'}`	Hachage multidimensionnel.
`$jours{2000,'Fev'}`	Émulation de hachage multidimensionnel.

Les tableaux ou les tranches de tableaux (ainsi que les tranches de hachage) débutent par le caractère `@`, qui fonctionne un peu comme l'article défini pluriel (les) :

5. Encore des termes couramment usités en informatique

Construction	Signification
`@jours`	équivaut à `($jours[0], $jours[1],... $jours[n])`.
`@jours[3,4,5]`	équivaut à `($jours[3], $jours[4], $jours[5])`.
`@jours[3..5]`	équivaut à `@jours[3,4,5]`.
`@jours{'jan','fev'}`	équivaut à `($jours{'jan'}, $jours{'fev'})`.

Les hachages débutent par `%` :

Construction	Signification
`%jours`	`@(jan => 31, fev => $bissextil e? 29: 28, ...)` .

Chacun de ces neuf exemples peut servir de lvalue, c'est-à-dire qu'ils spécifient une adresse à laquelle on peut, en autres choses, loger une valeur. Avec les tableaux, les hachages et les tranches de tableaux ou de hachages, la lvalue fournit un moyen simple d'affecter plusieurs valeurs d'un seul coup :

```
@days = 1 .. 7;
```

Noms

Nous avons parlé de variables pour mémoriser des valeurs mais il faut aussi mémoriser les noms et définitions de ces variables. En théorie, cela s'appelle des *espaces de nommage*. Perl dispose de deux sortes d'espaces de nommages : la *table des symboles* et les *portées lexicales*.[6] Il peut y avoir un nombre quelconque de tables et de portées lexicales et l'intersection des deux ensembles est nulle, pour chaque instance de l'interpréteur Perl. Nous détaillerons chacun de ces types tout au long de cet ouvrage. Disons simplement pour le moment que les tables de symboles sont des hachages à portée globale qui contiennent la table des symboles des variables globales (y compris les hachages des autres tables de symboles). A contrario, les portées lexicales sont des espaces *anonymes* non inclus dans une table, mais liés à un bloc de code du programme courant. Ils contiennent des variables accessibles uniquement dans le bloc en cours. (C'est pourquoi on utilise le terme *portée*.) Le mot *lexical* veut dire relatif au *bloc* de texte courant (ce qui n'a rien à voir avec ce qu'un lexicographe entend par là. Ne vous fâchez pas.).

Dans chaque espace de nommage, global ou lexical, chaque type de variable dispose de son propre sous-espace de nommage déterminé par le caractère spécial affecté au type. Il est possible, sans risque de conflit, d'utiliser le même nom pour une variable scalaire, un tableau ou un hachage (ou encore, pour un handle de fichier, un nom de sous-programme, un label ou votre lama familier). Cela signifie que `$machin` et `@machin` sont deux variables différentes. Cela veut aussi dire que `$machin[1]` est un élément de `@machin` et non une partie de `$machin`. Bizarre, vous avez dit bizarre ? Mais c'est normal, bizarrement.

6. On les appelle plutôt *paquetages* et *pads* quand il s'agit de l'implémentation spécifique de Perl, mais les expressions à rallonge sont les termes génériques utilisés dans l'industrie du logiciel, et nous les utiliserons donc. Désolé.

Pour l'appel d'un sous-programme le caractère `&` est facultatif. Généralement un nom de sous-programme n'est pas une lvalue, quoique dans les versions récentes de Perl un sous-programme peut s'évaluer à une lvalue de telle sorte que l'on peut lui affecter une valeur en retour.

Parfois on veut accéder aux variables d'un symbole donné, « `sym` » par exemple. Il suffit d'utiliser le caractère spécial * où l'astérisque indique tout type de variable (scalaire, tableau, hachage). On les appelle *typeglobs* ; ils ont plusieurs fonctions. Ils peuvent être utilisés comme lvalues. L'importation des modules se fait par l'affectation des *typeglobs*. Nous reviendrons sur ce thème plus tard.

Perl, comme tout langage informatique, dispose d'une liste de mots réservés qu'il reconnaît comme termes spéciaux. Cependant, les noms de variables commençant par un caractère spécial, il n'y a pas de confusion possible avec les premiers. D'autres types de noms ne commencent pas par ces caractères : les descripteurs de fichiers et les étiquettes. Avec ceux-ci attention à ne pas entrer en conflit avec des mots réservés. Nous vous recommandons pour ces types, d'utiliser des majuscules pour leurs noms. Par exemple, si vous écrivez `open(LOG, logfile)` plutôt que le regrettable `open(log, "logfile")`. Perl comprendra que vous ne parlez pas de la fonction log et des logarithmes,[7] et prévient les futurs conflits avec les nouvelles versions de Perl. Le nom des modules utilisateurs commence avec une majuscule alors que les modules pragmas prédéfinis sont écrits en minuscules. Quand nous aborderons la programmation orientée objet, vous verrez que les noms de classes sont en capitales pour la même raison.

Comme vous pouvez le déduire du précédent paragraphe, la casse est importante dans les identificateurs — `TRUC`, `Truc`, et `truc` sont trois noms différents en Perl. Un identificateur commence avec une lettre ou un souligné, il peut contenir des lettres et des chiffres, y compris les caractères Unicode, et peut avoir une longueur comprise entre 1 et 251 inclus. Les idéogrammes Unicode sont des lettres mais nous vous déconseillons des les utiliser si vous ne savez pas les lire. Voir le chapitre 15.

Les noms qui suivent ces caractères spéciaux ne sont pas nécessairement des identificateurs. Ils peuvent commencer par un chiffre auquel cas l'identificateur ne peut contenir que des chiffres comme dans `$123`. Les noms qui commencent par tout autre caractère qu'une lettre, un chiffre ou souligné (comme `$?` ou `$$`), sont limités à ce caractère et sont prédéfinis en Perl. Par exemple `$$` est l'ID du processus courant et `$?` est l'indicateur d'état retourné par le dernier processus fils créé.

Dans la version 5.6 de Perl est implémentée une syntaxe extensible pour les noms de variable interne. Toute variable de la forme `${^NAME}` est une variable spéciale réservée par Perl. Tous ces noms spéciaux sont dans la table principale de symboles. Voir le chapitre 28, *Noms spéciaux*, pour les exemples.

On pourrait penser que noms et identificateurs désignent la même chose, mais lorsqu'on parle de *nom*, c'est du nom absolu qu'il s'agit, c'est-à-dire celui qui indique à quelle table de symboles il appartient. De tels noms sont formés par une série d'identificateurs séparés par le symbole `::` :

```
$Doc::Aide::Perl::Classe::chameau
```

7. On applique ici un des principes de Perl qui prétend que des choses différentes doivent être représentées différemment, ce qui rend le code plus lisible (à comparer avec les langages qui obligent des choses différentes à avoir la même apparence, au détriment de la lisibilité.

Ce qui fonctionne comme un nom absolu de fichier :

```
/Doc/Aide/Perl/Classe/chameau
```

En Perl, ce chemin absolu indique les tables de symboles imbriquées, et le dernier identificateur est le nom de la variable elle-même, contenu dans la dernière table imbriquée. Par exemple, pour la variable de l'exemple précédent, la table de symbole se nomme `Doc::Aide::Perl::Classe`, et le nom de la variable dans cette table est $chameau. (la valeur de cette variable est bien entendu « `beige` ».)

Une table de symboles en Perl se nomme aussi un *paquetage*. Donc ces variables sont aussi appelées variables de paquetage. Les variables de paquetage sont dites privées relativement à leur paquetage, mais sont globales dans le sens où les paquetages sont eux-mêmes globaux, c'est-à-dire qu'il suffit de nommer le paquetage pour y accéder, ce qui est difficile à faire par inadvertance. Par exemple, un programme qui référence `$Doc::categorie` demande la variable `$categorie` dans le paquetage `Doc::`, qui n'a rien à voir avec la variable `$Livre::categorie`. Voir le chapitre 10, *Paquetages*.

Les variables attachées à une portée lexicale ne sont pas contenues dans un paquetage, et ne contiennent pas la séquence `::`. (Les variables à portée lexicale sont déclarées avec `my`.)

Recherche des noms

La question est donc comment Perl devine le chemin absolu d'un nom ? Comment trouve-t-il par exemple le nom de `$categorie` ? Voici les étapes successives suivies par l'interpréteur Perl dans la phase de lecture du code, pour résoudre les noms dans le contexte courant :

1. Premièrement, Perl inspecte le bloc immédiatement contenant, pour une déclaration du type `my` (ou `our`) (Voir le chapitre 29, ainsi que la section *Déclarations avec portée* au chapitre 4, *Instructions et déclarations*). S'il trouve cette déclaration, alors la variable est à portée lexicale et n'existe dans aucun — elle existe uniquement dans le bloc courant. Une portée lexicale est anonyme et ne peut donc être atteinte hors du bloc la contenant.[8]
2. S'il ne trouve pas, Perl cherche une variable à portée lexicale dans le bloc contenant le précédent et s'il la trouve, il la marque comme appartenant à ce bloc de niveau supérieur. À noter que dans ce cas, sa portée comprend le bloc de la première étape. Dans le cas inverse, il répète cette deuxième étape jusqu'à la sortie du bloc racine, le plus grand contenant les précédents.
3. Quand il n'y a plus de blocs contenants, l'interpréteur examine l'*unité de compilation* entière, ce qui peut être le fichier entier ou la chaîne en cours de compilation dans le cas de l'expression `eval` *`CHAINE`*. Dans ce dernier cas, la portée lexicale est la chaîne elle-même et non un bloc entre accolades. Si Perl ne trouve pas la variable

8. Si la déclaration `our` est utilisée à la place, cela revient à déclarer un alias de variable de paquetage. Le code extérieur peut accéder à cette variable mais seulement par l'intermédiaire du paquetage où la variable est définie. Pour le reste une définition `our` fonctionne exactement comme une définition `my`. C'est utile pour limiter l'usage des globales avec la directive `use strict` (voir la directive `strict` au chapitre 31). Mais il est préférable d'utiliser `my` quand une variable globale n'est pas nécessaire.

dans la portée lexicale de la chaîne, il considère qu'il n'y a plus de bloc et retourne à l'étape 2 en partant de la portée lexicale du bloc `eval` *`STRING`*.

4. À ce point, Perl n'a pas encore trouvé de déclaration pour notre variable (ni `my`, ni `our`). Perl abandonne la recherche d'une variable lexicale, et suppose que la variable recherchée est une variable de paquetage. Le degré suivant de recherche est donc le paquetage. Si la directive `strict` est active, le compilateur provoque une erreur, (sauf si la variable est prédéfinie ou importée dans le paquetage courant), car il ne peut exister de variable globale sans qualifiant. Il cherche la déclaration `package` toujours dans la portée lexicale, et s'il la trouve, insère le nom du paquetage trouvé au début de la variable.
5. S'il ne trouve pas de déclaration de paquetage dans la portée lexicale, Perl recherche la variable dans le paquetage `main` qui contient tous les autres. En l'absence de déclaration contraire, `$categorie` veut dire `$::categorie`, qui veut dire `$main::categorie`. (`main` étant un paquetage dans le paquetage de plus haut niveau, cela veut aussi dire `$::main::categorie` ou encore `$main::main::categorie` et `$::main::main::categorie` et ainsi de suite. On en verra toute l'utilité dans *Tables de symboles* au chapitre 10.)

Il y a plusieurs conséquences que nous devons souligner ici :

- Le fichier étant la plus grande portée lexicale, une variable à portée lexicale ne peut être vue en dehors du fichier dans lequel elle est déclarée. Les portées de fichier ne sont pas imbriquées.
- Tout code Perl compilé appartient à au moins une portée lexicale et exactement un paquetage. La portée obligatoire est le fichier de code lui-même. Les blocs contenant définissent les portées successives. Tout code Perl est compilé dans un paquetage exactement, et si la déclaration du paquetage a une portée lexicale, le paquetage n'en a pas, il est global.
- Une variable sans qualifiant peut être cherchée dans différentes portées lexicales, mais dans un seul paquetage, qui est le paquetage actif (déterminé par la portée lexicale de la variable).
- Un nom de variable ne peut avoir qu'une portée. Bien que deux portées sont actives au même moment (lexicale et paquetage), une variable ne peut exister que dans une seule à la fois.
- Un nom de variable sans qualifiant ne peut exister que dans une place mémoire, soit dans la première portée lexicale, soit dans le paquetage courant, mais pas les deux.
- La recherche s'arrête dès que la place mémoire est trouvée, et les autres places mémoire qui auraient été trouvées si la recherche s'était poursuivie, ces places mémoire donc sont néanmoins inaccessibles.
- La place mémoire d'une variable typique peut être complètement déterminée à la compilation.

Maintenant on sait comment le compilateur Perl résoud les noms, mais parfois on ne *connaît* pas ce nom au moment de la compilation : par exemple dans le cas d'une *indirection*. Dans ce cas, Perl fournit un mécanisme qui permet de remplacer une variable par une expression qui retourne une *référence* à cette variable. Par exemple, au lieu de dire :

```
$categorie
```

vous diriez :

```
${ une_expression() }
```

et si la fonction `une_expression()` retourne une référence à la variable `$categorie` (ou bien la chaîne `"categorie"`), cela fonctionne de la même manière. Par contre, si la fonction retourne `$thesaurus` ce sera cette variable à la place. Cette syntaxe est la plus générale (et la moins lisible) des formes d'indirection, mais nous verrons des variantes plus pratiques au chapitre 8, *Références*.

Valeurs scalaires

Qu'il soit référencé directement ou indirectement, dans une variable, un tableau ou bien une variable temporaire, un scalaire contient toujours une valeur simple qui peut être un nombre, une chaîne ou une référence à une autre donnée. Il peut aussi n'avoir aucune valeur auquel cas il est dit *indéfini*. Les scalaires sont sans type, même s'ils peuvent référencer toutes sortes de données : il n'est pas nécessaire de les déclarer.[9]

Perl mémorise les chaînes comme des séquences de caractères sans contrainte de longueur ni de contenu. Nul besoin de déclarer une taille à l'avance et tout caractère peut y être inclus, y compris le caractère nul. Les nombres sont représentés sous forme d'entiers signés si possible, ou bien sous forme de nombres à virgule flottante en double précision dans le format natif de la machine. Les valeurs flottantes ne sont pas infiniment précises, c'est pourquoi certaines comparaisons de la forme `(10/3 == 1/3*10)` échouent mystérieusement.

Perl convertit les scalaires en différents sous-types suivant les besoins ; le numérique peut être traité comme du caractère et inversement, Perl faisant Ce Qu'il Faut. Pour convertir du caractère en numérique, Perl utilise la fonction C *atof*(3). Pour passer du numérique à une chaîne de caractères, il fait l'équivalent de *sprintf*(3) avec le format `"%.14g"` sur la plupart des machines. La conversion d'une chaîne non numérique comme `truc` convertit le literal à la valeur 0 ; s'ils sont actifs, les avertissements sont affichés sinon rien. Voir le chapitre 5, pour des exemples permettant d'identifier le contenu d'un chaîne.

Alors que les chaînes de caractères et les numériques sont interchangeables dans à peu près tous les cas, les références sont d'un genre différent. Elles sont fortement typées, ce sont des pointeurs non convertibles (transtypables) comprenant des compteurs de références et des invocations de destructeurs internes. Ils sont utilisables pour créer des types de données complexes, incluant vos propres objets. Ceci dit, les références restent des scalaires, car ce qui importe n'est pas tant la compexité d'une structure que le fait de la considérer comme un simple scalaire.

Par *non convertibles*, nous voulons dire que l'on ne peut pas, par exemple, convertir une

9. Les versions futures de Perl permettront de déclarer les trois types `int`, `num` et `str`, pour lever toute ambiguité pour l'optimiseur de code. Ces types seront utilisés dans du code critique qui doit s'exécuter rapidement, et nous n'entrerons pas dans le détail maintenant. Le mécanisme de pseudo-hash utilise ces types optionels à la manière d'un langage plus fortement typé. Voir le chapitre 8 pour aller plus loin.

référence de tableau en une référence de hachage. Les références ne sont pas convertibles en un autre type pointeur. Cependant, en utilisant une référence en numérique ou en caractère, on obtient une valeur numérique ou une chaîne de caractères, ce qui permet de se souvenir du caractère unique d'une référence, même si le « référencement » de la valeur est perdu au cours de la copie depuis la vraie référence. On peut comparer différentes références ou tester si elles sont définies. Mais on ne peut en faire beaucoup plus puisqu'il n'existe pas de moyen de convertir du numérique ou du caractère en référence. En principe, si Perl n'oblige pas à faire de l'arithmétique de pointeurs — ou carrément l'interdit —, ce n'est pas un problème. Voir le chapitre 8 pour en connaître plus sur les références.

Littéraux numériques

Les littéraux numériques sont spécifiés par n'importe quel format courant[10] de virgule flottante ou d'entier :

```
$x = 12345;              # entier
$x = 12345.67;           # virgule flottante
$x = 6.02E23;            # notation scientifique
$x = 4_294_967_296;      # souligné pour la lisibilité
$x = 0377;               # octal
$x = 0xffff;             # hexadécimal
$x = 0b1100_0000;        # binaire
```

Comme Perl utilise la virgule comme séparateur de liste, elle ne peut servir comme séparateur des milliers pour les grands nombres. Pour améliorer leur lisibilité, Perl permet d'utiliser à la place le caractère souligné. Celui-ci ne fonctionne qu'avec des constantes numériques définies dans le programme, et non pour les chaînes de caractères traitées comme du numérique ou pour les données externes lues par ailleurs. De même, les caractères initiaux `0x` en hexadécimal et `0` en octal marchent *uniquement* avec les constantes. La conversion automatique d'une chaîne en un nombre ne reconnaît pas ces préfixes — vous devez faire une conversion explicite[11] avec la fonction `oct` (qui marche aussi pour les données hexadécimales à condition d'indiquer `0x` ou `0b` au début).

Chaînes littérales

Les littéraux alphanumériques sont habituellement délimités par des apostrophes simples ou doubles. Ils fonctionnent un peu comme les *quotes* (les délimiteurs de protection) sous UNIX : les apostrophes doubles permettent une interpolation des antislashs et des variables, les apostrophes simples non. (`\'` et `\\` permettent d'inclure un antislash dans une chaîne entre apostrophes simples). Si vous voulez inclure tout autre séquence

10. Courant dans la culture UNIX, n'est-ce pas. Si votre culture est différente, bienvenue dans la nôtre !

11. On s'imagine parfois que Perl devrait de lui-même convertir toutes les données. Mais il existe beaucoup trop de nombres décimaux avec des zéros à gauche dans le monde pour que Perl puisse le faire automatiquement. Par exemple, le code postal des bureaux de O'Reilly & Associates à Cambridge, MA, est `02140`. Le facteur aurait tendance à s'énerver si votre programme d'adressage de courrier transformait `02140` en la valeur décimale `1120`.

telle que \n (saut de ligne), vous devez utiliser les apostrophes doubles. (Les séquences antislashs sont des séquences d'échappement car elles permettent d'« échapper » temporairement à l'interprétation normale des caractères.)

Une chaîne entre apostrophes simples doit être séparée du mot précédent par un espace car ce caractère est valide — mais archaïque — dans un identificateur. On utilise plutôt la séquence :: plus visuelle: $main'var et $main::var représentent la même variable, la deuxième étant beaucoup plus lisible.

Les chaînes entre guillemets permettent l'interpolation de différents types de caractères bien connus dans les autres langages. Ils sont listés au *tableau 2-1*.

Tableau 2-1. Caractères échappés avec antislash

Code	Signification
\n	Saut de ligne (généralement LF).
\r	Retour chariot (généralement CR).
\t	Tabulation horizontale.
\f	Saut de page.
\b	Retour arrière.
\a	Alerte (bip).
\e	Caractère ESC.
\033	ESC en octal.
\x7f	DEL en hexadécimal.
\cC	Control-C.
\x{263a}	Unicode (smiley).
\N{*NAME*}	Caractère nommé.

La notation \N{*NOM*} est utilisée avec la directive use charnames décrite au chapitre 31. Cette notation permet d'utiliser les noms symboliques comme \N{GREEK SMALL LETTER SIGMA}, \N{greek:Sigma}, ou \N{sigma} — suivant la directive utilisée. Voir aussi le chapitre 15.

Il existe de plus des séquences d'échappement pour modifier la casse des caractères qui suivent. Voir le *tableau 2-2*.

Tableau 2-2. Échappements de casse

Code	Signification
\u	Force le caractère suivant en majuscule (« titlecase » en Unicode).
\l	Force le caractère suivant en minuscule.
\U	Force tous les caractères suivants en majuscules.
\L	Force tous les caractères suivants en minuscules.
\Q	Préfixe avec antislash tous les caractères non alphanumériques.
\E	Fin de \U, \L, ou \Q.

Il est aussi possible d'insérer le caractère de saut de ligne dans une chaîne qui peut donc s'étendre sur plus d'une ligne. Ceci peut être utile, mais aussi dangereux si vous oubliez un guillemet, auquel cas l'erreur sera annoncée à la prochaine occurrence de guillemet qui peut se trouver beaucoup plus loin dans votre fichier source. Heureusement ceci provoque immédiatement une erreur de syntaxe sur la même ligne, et Perl est suffisamment futé pour vous avertir qu'il s'agit peut-être d'une chaîne qui n'est pas terminée au bon endroit, et de plus il vous indique la ligne où la chaîne a dû commencer.

Parallèlement aux séquences d'échappement ci-dessus, les chaînes entre doubles apostrophes permettent une *interpolation des variables* scalaires et des listes de valeurs. Cela signifie que les valeurs de certaines variables peuvent être directement insérées dans une chaîne. C'est en fait une forme pratique de concaténation de chaîne.[12] L'interpolation ne peut être effectuée que pour des variables scalaires, la totalité d'un tableau (mais pas d'un hachage), des éléments unitaires d'un tableau ou d'un hachage, ou enfin, des tranches (des sélections multiples d'éléments) de tableaux ou de hachages. En d'autres termes, seules les expressions commençant par `$` ou `@` peuvent être interpolées, car ce sont les deux caractères (avec l'antislash) que l'analyseur de chaîne recherche. À l'intérieur d'une chaîne, un caractère `@` ne faisant pas partie de l'identifiant d'un tableau ou d'un hachage doit être inclus dans une séquence d'échappement avec un antislash (`@`), sous peine d'une erreur de compilation. Bien qu'un hachage complet spécifié par `%` ne puisse être interpolé dans une chaîne, un élément ou une sélection multiples d'éléments d'un hachage peuvent l'être car ils commencent respectivement par les caractères `$` et `@`.

Le bout de code suivant affiche `"Le prix est de $100."`.

```
$Prix = '$100';                     # non interpolé
print "Le prix est de $Prix.\n";    # interpolé
```

Comme dans certains shells, les accolades autour d'un identifiant permettent de le différencier des autres caractères alphanumériques : `"How ${verb}able!"`. En fait un identifiant entre accolades est forcément interprété en chaîne, comme les chaînes clef d'un hachage. Par exemple,

```
$jours{'fev'}
```

peut aussi être écrit :

```
$jours{fev}
```

et les apostrophes sont censées être automatiquement présentes. Mais toute chose plus complexe dans l'indice sera interprétée comme une expression.

et vous devriez les mettre entre guillemets simples :

```
$jours{'29 Février'}     # Ok.
$jours{"29 Février"}     # Ok. "" pas besoin d'interpoler.
$jours{ 29 Février }     # mauvais, provoque un erreur de lecture.
```

En particulier, il est nécessaire d'utiliser les simples guillemets dans les tranches de tableau :

12. Avec les warnings activés, Perl peut retourner une erreur sur des valeurs indéfinies interpolées dans des chaînes concaténes ou jointes, même implicitement, le compilateur les créant pour vous.

```
@jours{'Jan','Fev'}      # Ok.
@jours{"Jan","Fev"}      # Ok aussi.
@jours{ Jan,  Fev }      # erreur si la directive use strict est active
```

Exceptés les indices de tableaux interpolés ou de hachages, il n'existe pas de niveaux multiples d'interpolation. En particulier, contrairement aux attentes de nombreux programmeurs shell, les apostrophes inverses ne permettent *pas* d'interpoler à l'intérieur d'apostrophes doubles. De même, les apostrophes simples ne permettent pas d'empêcher l'évaluation de variables à l'intérieur d'apostrophes doubles. L'interpolation est très puissante mais d'un contrôle strict en Perl. Elle ne se produit que dans les guillemets et certaines opérations que nous décrirons dans la prochaine section.

```
print "\n";              # Ok, imprime un saut de ligne.
print  \n ;              # MAUVAIS, pas de contexte d'interpolation.
```

Choisissez vos délimiteurs

Alors que l'on ne considère souvent les délimiteurs que comme des constantes, ils fonctionnent plus comme des opérateurs avec Perl, permettant plusieurs formes d'interpolation et de correspondance de motifs. Perl fournit des délimiteurs spécifiques pour ces fonctions, mais offre également le moyen de choisir un délimiteur pour chacune. Au *tableau 2-3*, on peut utiliser tout caractère autre qu'alphanumérique ou d'espace comme délimiteur à la place de `/`. (Les caractères saut de ligne et d'espace ne sont plus admis dans les nouvelles versions de Perl.)

Tableau 2-3. Constructions avec guillemets

Habituelle	Générique	Signification	Interpolation
''	q//	chaîne littérale	non
""	qq//	chaîne littérale	oui
``	qx//	exécution de commande	oui
()	qw//	liste de mots	non
//	m//	recherche de motif	oui
s///	s///	substitution de motif	oui
y///	tr///	traduction	non
""	qr//	expression rationnelle	oui

Certains, ci-dessus, sont uniquement des formes de « sucre syntaxique » ; ils évitent d'alourdir une chaîne avec trop d'antislashs, en particulier dans les expressions régulières où slash et antislash se côtoient.

Si vous utilisez les guillemets simples comme délimiteurs, aucune interpolation de variable ne sera faite (même si le tableau ci-dessus indique « oui »). Si le délimiteur ouvrant est un crochet, une parenthèse, une accolade ou le signe inférieur, le délimiteur fermant doit lui correspondre (les occurrences de délimiteurs doivent correspondre par paires). Par exemple :

```
$simple = q!Je dis "Tu dis, 'Elle l'a dit.'"!;
$double = qq(On ne peut pas avoir une "bonne" $variable?);
```

```
$bout_de_code = q {
    if ($condition) {
        print "Pris!";
    }
};
```

Le dernier exemple montre qu'il est possible d'utiliser le caractère espace entre l'opérateur de protection et le délimiteur. Pour les constructions à deux éléments, comme `s///` et `tr///`, si la première paire de délimiteurs est une accolade, la seconde partie peut en utiliser un autre : on peut écrire `s<foo>(bar)` ou `tr(a-f)[A-F]`. Les espaces sont autorisés entre les deux paires de délimiteurs, ainsi le dernier exemple peut s'écrire :

```
tr [a-z]
   [A-Z];
```

Les caractères espace sont toutefois interdits lorsque le délimiteur est `#`. `q#foo#` est lu comme `'foo'`, alors que q #foo# est lu comme l'opérateur `q` suivi d'un commentaire. Le délimiteur sera pris sur la ligne suivante. Les commentaires peuvent être placés au milieu d'une construction à deux éléments, ce qui permet d'écrire :

```
s {foo}   # Remplacer foo
  {bar};  #    par bar.

tr [a-f]  # Translittération de minuscule hexa
   [A-F]; #            vers majuscule hexa
```

Ou n'en mettez pas du tout

Un mot qui n'a pas d'interprétation dans la grammaire sera traité comme s'il était dans une chaîne protégée. Ce sont les *mots simples*.[13] Comme pour les handles de fichier et les labels, un mot simple entièrement en minuscules risque d'entrer en conflit avec un futur mot réservé. Si vous utilisez l'option **`-w`**, Perl vous avertira de ce risque. Par exemple :

```
@jours = (Lun,Mar,Mer,Jeu,Ven);
print STDOUT Salut, ' ', monde, "\n";
```

stocke dans le tableau `@jours` les abréviations des jours de la semaine et affiche « `Salut tout le monde` » suivi d'un saut de ligne sur `STDOUT`. En ôtant le handle de fichier, Perl essaiera d'interpréter `Salut` comme un handle de fichier avec à la clef une erreur de syntaxe. C'est tellement source d'erreur que certains voudront déclarer hors la loi les mots simples. Les opérateurs de protection listés ci-avant ont d'autres formes y compris l'opérateur « quote words » `qw//` qui opère sur une liste de mots séparés par des espaces :

```
@jours = qw(Lun Mar Mer Jeu Ven);
print STDOUT "Salut, tout le monde\n";
```

Vous pouvez aussi bannir les mots simples de votre code. Si on écrit :

---

13. Les noms de variables, descripteurs de fichier, labels et autres, ne sont pas considérés comme des mots simples car ils ont un sens qui dépend du mot précédent ou suivant (ou les deux). Les noms prédéfinis comme les noms de sous-programme n'en sont pas non plus. Un mot simple le reste tant qu'il n'y a pas la preuve du contraire.

```
use strict 'subs';
```

alors tout mot simple qui n'est *pas* interprété comme un appel à un sous-programme produit à la place une erreur de compilation. Cette restriction dure jusqu'à la fin du bloc qui la contient. Un autre bloc peut l'annuler par

```
no strict 'subs';
```

Il faut remarquer que les identifiants dans des constructions telles que :

```
"${verb}able"
$jours{Fév}
```

ne sont pas considérés comme des mots simples car ils sont autorisés par une règle explicite plutôt que par le fait de « ne pas avoir d'interprétation dans la grammaire ».

Un nom simple avec `::` en fin, tel que `main::` ou bien `Chameau::` est toujours considéré comme un nom de paquetage. Perl transforme `Chameau::` en chaîne « `Chameau` » au moment de la compilation, et ne sera pas rejeté par la directive `use strict`.

Interpolation des tableaux

Les variables tableau sont interpolées dans une chaîne entre doubles apostrophes en réunissant tous leurs éléments avec le délimiteur spécifié dans la variable `$"`[14] l'espace sera pris par défaut. Les exemples suivants sont équivalents :

```
$temp = join($",@ARGV);
print $temp;

print "@ARGV";
```

À l'intérieur des motifs de recherche (qui comprend aussi une interpolation identique à celle des doubles guillemets) il existe une ambiguïté déplaisante : `/$machin[chose]/` est-il interprété comme `/${machin}[chose]}` (où `[chose]` est une classe de caractères pour l'expression) ou comme `/${machin[chose]}/` (où `[chose]` est l'indice du tableau `@machin`) ? Si `@machin` n'existe pas, c'est bien sûr une classe de caractères. Si `@machin` existe, Perl devinera `[chose]`, presque toujours à bon escient.[15] S'il décrypte mal, ou si vous êtes paranoïaque, des accolades permettent de forcer l'interprétation correcte comme ci-dessus. Et même si vous êtes simplement prudent, ce n'est pas une mauvaise idée.

Documents « ici-même »

Un format de séparation orienté ligne est basé sur la syntaxe des documents « ici-même » (*« here » documents*) du shell[16] Après les caractères `<<`, on saisit une chaîne destinée à terminer la partie à délimiter, toutes les lignes suivant la ligne courante jusqu'à

14. `$LIST_SEPARATOR` si vous utilisez le module English.

15. Le « devin » est trop ennuyeux à décrire complètement, mais, grosso modo, il fait une moyenne pondérée de ce qui ressemble à une classe de caractères (`a-z`, `\w`, les caractères initiaux `^`) par rapport à tout ce qui ressemble à des expressions (les variables et les mots réservés).

16. Il est orienté ligne dans le sens où les délimiteurs sont les lignes plutôt que des caractères. Le délimiteur initial est la ligne courante, et celui de la fin est une ligne contenant la chaîne spécifiée.

la chaîne de terminaison étant délimitées. La chaîne de terminaison peut être un identifiant (un mot) comme du texte délimité. Dans ce cas, le type de délimiteur utilisé détermine le traitement du texte, comme en délimitation standard. Un identifiant non délimité fonctionne comme des apostrophes doubles. Il ne doit pas y avoir d'espace entre les caractères << et l'identifiant (un espace est traité comme un identifiant nul, ce qui est valable mais pas recommandé, et correspond avec la première ligne vide ; voir le premier exemple `Hourra~!` plus loin). La chaîne de terminaison doit apparaître en tant que telle (sans délimiteurs et non entourée d'espaces) sur la dernière ligne.

```
print <<EOF;        # même exemple que précédemment
Le prix est $Prix.
EOF

print <<"EOF";      # même exemple qu'au dessus avec des apostrophes
                    # doubles explicites
Le prix est $Prix.
EOF

print <<'EOF';      # avec des apostrophes simples
Tout est possible (par exemple le passage d'un chameau par le chas
d'une aiguille), c'est vrai. Mais imaginez l'état du chameau,
déformé en un long fil sanglant de la tête aux pieds.
-- C.S. Lewis
EOF

print << x 10;      # affiche la ligne suivante 10 fois
Les chameaux débarquent! Ho u rra! Hourra!

print <<"" x 10;    # La même chose, en mieux
Les chameaux débarquent! Ho u rra! Hourra!

print <<`EOC`;      # exécute les commandes
echo comment vas-tu
echo yau de poêle?
EOC

print <<"dromadaire", <<"camelide";  # on peut les empiler
Je dis bactriane.
dromadaire
Elle dit lama.
camelide
```

N'oubliez quand même pas de mettre un point virgule à la fin pour finir la phrase, car Perl ne sait pas que vous n'allez pas essayer de faire ceci :

```
print <<ABC
179231
ABC
+ 20;   # affiche 179251
```

Si vous voulez indenter vos documents « ici-même » avec votre code, il faut enlever les espaces devant chaque ligne manuellement :

```
($quote = <<'QUOTE') =~ s/^\s+//gm;
    The Road goes ever on and on,
    down from the door where it began.
QUOTE
```

Il est aussi possible d'instancier un tableau avec les lignes d'un document « ici-même » comme suit :

```
@sauces = <<fin_lignes =~ m/(\S.*\S)/g;
   tomate normale
   tomate épicée
   chili vert
   pesto
   vin blanc
fin_lignes
```

Litéral V-chaîne

Un litéral qui commence avec un v suivi d'un ou plusieurs entiers séparés par des points est traduit en chaîne dont chaque atome a pour valeur ordinale chaque entier.

```
$crlf = v13.10;           # valeur ASCII: retour chariot, saut de ligne
```

Le terme *v-chaîne* est une contraction de « vecteur-chaîne » ou « version de chaîne ». Il fournit une alternative plus cohérente pour construire une chaîne à partir des valeurs numériques de chaque caractère. Par exemple, il est plus facile d'écrire `v1.20.300.4000` plutôt que :

```
"\x{1}\x{14}\x{12c}\x{fa0}"
pack("U*", 1, 20, 300, 4000)
chr(1) . chr(20) . chr(300) . chr(4000)
```

Si un tel littéral est composé de deux ou trois points (au moins trois entiers), le préfixe v peut être omis.

```
print v9786;              # affiche le caractère UTF-8 SMILEY, "\x{263a}"
print v102.111.111;       # affiche "foo"
print 102.111.111;        # idem

use 5.6.0;                # nécessite au moins cette version de Perl

$ipaddr = 204.148.40.9;   # l'addresse IP de oreilly.com
```

Les v-chaînes sont pratiques pour représenter les adresses IP ou les numéros de version. En particulier avec le codage de caractères sur plus d'un octet qui devient courant de nos jours, on peut comparer les numéros de version de toute taille avec les v-chaînes.

Les numéros de version et les adresses IP ne sont pas très lisibles sous la forme de v-chaînes. Pour obtenir une chaîne affichable utilisez l'option v dans un masque avec `printf`, comme par exemple dans `"%vd"`, tel que décrit au chapitre 29. Pour en savoir plus sur les chaînes Unicode, voir le chapitre 15, ainsi que la directive `use bytes` au chapitre 31 ; pour la comparaison des versions avec les opérateurs de comparaison de chaîne, voir `$^V` au chapitre 28 ; et pour la représentation des adresses IPv4, voir `gethostbyaddr` au chapitre 29.

Autres symboles littéraux

Tout symbole préfixé par deux caractères « souligné » et postfixé de même est réservé pour un usage spécial par Perl. Deux mots spéciaux, à savoir `__LINE__` et `__FILE__`, représentent le numéro de la ligne courante et le fichier à ce stade du programme. Ils ne peuvent être utilisés que comme des mots séparés ; ils ne sont pas interpolés dans les chaînes. De la même manière, `__PACKAGE__` est le nom du paquetage courant de compilation. S'il n'existe pas, ce qui est le cas lorsque la directive `package;` est vide, `__PACKAGE__` vaut undef. Le symbole `__END__`, ou les caractères Control-D ou Control-Z, peuvent être utilisés pour indiquer la fin logique du script courant avant la fin physique du fichier. Tout caractère suivant est ignoré par le lecteur Perl mais peut être lu *via* le descripteur spécial `DATA`.

`__DATA__` fonctionne de la même façon que `__END__`, mais ouvre le handle de fichier `DATA` à l'intérieur de l'espace de nom du paquetage courant, ce qui *fait* que si vous insérez plusieurs fichiers par `require`, chacun peut disposer de son handle `DATA` personnel et l'ouvrir sans conflit avec les autres. Pour en savoir plus, voir `DATA` au chapitre 28.

Contexte

Jusqu'à présent, nous avons vu un certain nombre de termes pouvant produire des valeurs scalaires. Avant d'en voir davantage, nous devons aborder la notion de *contexte*.

Contexte scalaire et contexte de liste

Chaque opération[17] d'un script Perl est évaluée dans un contexte spécifique, et la façon dont l'opération se comportera peut dépendre des contraintes de ce contexte. Il existe deux contextes majeurs : les *scalaires* et les *listes*. Par exemple, l'affectation d'une variable scalaire évalue la partie droite dans un contexte scalaire :

```
$x         = fonction();  # contexte scalaire
$x[1]      = fonction();  # contexte scalaire
$x{"ray"}  = fonction();  # contexte scalaire
```

Mais l'affectation d'un tableau ou d'un hachage (ou d'une tranche de ceux-ci) évalue la partie droite dans un contexte de liste. L'affectation d'une liste de scalaire se fera aussi par un contexte de liste pour la partie droite.

```
@x         = fonction();  # contexte de liste
@x[1]      = fonction();  # id
@x{"ray"}  = fonction();  # id
%x         = fonction();  # id
```

L'affectation à une liste de scalaires fournit aussi un contexte de liste en rvalue, même s'il n'y a qu'un seul élément dans la liste produite :

```
($x,$y,$z) = fonction();  # contexte de liste
($x)       = fonction();  # id
```

17. Nous utilisons ce terme à la fois pour opérateur et terme. Ces deux concepts se confondent l'un et l'autre lorsque l'on commence à parler de fonctions qui agissent comme des termes mais qui ressemblent à des opérateurs unaires.

Ces règles sont inchangées quand la variable est déclarée avec les termes `my` ou `our`, ainsi nous avons :

```
my $x      = fonction();  # contexte scalaire
my @x      = fonction();  # contexte de liste
my %x      = fonction();  # id
my ($x)    = fonction();  # id
```

Vous aurez beaucoup de problèmes jusqu'au jour où vous aurez compris la différence entre le contexte scalaire et le contexte de liste, car certains opérateurs (tel notre fonction imaginaire `fonction()`), connaissent leur contexte de retour et typent leur valeur de retour en conséquence. (Ce comportement sera toujours signalé quand c'est le cas.) En langage informatique, on dit que les opérateurs ont leur type de retour *surchargé* par le contexte d'affectation. Il s'agit de la surcharge la plus simple basée sur deux types simples, le scalaire et la liste.

Si certains opérateurs répondent suivant le contexte, c'est parce que quelque chose indique son type scalaire ou liste. L'affectation se comporte comme une fonction qui fournit à son opérande de droite le contexte. On a besoin de savoir *quel* opérateur fournit *quel* contexte pour ses opérandes. Toutes les fonctions avec un contexte de liste, ont le terme *LISTE* dans leur description. Celles qui ne l'ont pas ont un scalaire. C'est en général assez intuitif.[18] Au besoin, on peut forcer un contexte scalaire au milieu d'un contexte de *LISTE* en utilisant la pseudo-fonction `scalar`. (Perl ne fournit pas le moyen de forcer un contexte de liste en contexte scalaire, car partout où un contexte de liste est prévu, ceci est déjà indiqué par le contexte *LISTE* de certaines fonctions de contrôle.)

Le contexte scalaire peut ensuite être classé en contexte de chaînes, numériques ou contexte tolérant. À la différence de la distinction entre scalaire et liste que nous venons de faire, les opérations ne savent jamais *quel* type de contexte scalaire elles ont en entrée. Elles se contentent de renvoyer le type de valeur scalaire qu'elles désirent et laissent Perl traduire les numériques en chaîne dans un contexte de chaîne, et les chaînes en nombres dans un contexte numérique. Certains contextes scalaires se moquent de savoir si une chaîne ou un numérique est retourné, ce qui fait qu'aucune conversion ne sera effectuée. (Cela arrive, par exemple, lorsqu'on affecte une valeur à une autre variable. Celle-ci prend simplement le même sous-type que l'ancienne valeur.)

Contexte booléen

Le contexte booléen est un contexte scalaire particulier. Il correspond à l'endroit où une expression est évaluée pour savoir si elle est à vrai ou à faux. Parfois nous écrivons Vrai ou Faux à la place de la définition technique utilisée par Perl : une valeur scalaire est à vrai si ce n'est pas une chaîne nulle ou le nombre 0 (ou sa chaîne équivalente : `"0"`). Les références sont toujours vraies. Une référence est toujours vraie car c'est une adresse physique jamais nulle. Une valeur indéfinie, appelée `undef`, est toujours fausse car elle vaut, suivant le contexte, `""` ou 0. (Les listes n'ont pas de valeur booléenne car elles ne sont jamais produites dans un contexte scalaire !)

18. Remarquez cependant que le contexte *LISTE* peut se propager aux appels de sous-programmes ultérieurs et il n'est donc pas toujours facile de savoir d'un seul coup d'œil si une phrase va être évaluée dans un contexte scalaire ou de liste. Le programme peut retrouver son contexte dans un sous-programme par la fonction `wantarray`.

Puisque le contexte booléen est un contexte tolérant, il n'entraîne aucune conversion ; à noter que c'est un contexte scalaire, et tout opérande pour lequel c'est nécessaire sera converti en scalaire. Et pour tout opérande le nécessitant, le scalaire retourné dans un contexte scalaire est une valeur booléenne possible, ce qui signifie par exemple qu'un opérateur retournant une liste, peut être utilisé dans un test booléen : la fonction `unlink` qui opère dans un contexte de liste, peut utiliser un argument de type array :

```
unlink @liste_fichiers;      # détruit chaque fichier de la liste.
```

Maintenant, utilisons le tableau dans une condition (donc contexte booléen), alors le tableau retourne sa cardinalité car il sait que c'est un contexte scalaire ; cette valeur est vraie tant qu'il reste des éléments dans le tableau. Supposons alors que nous voulons vérifier que les fichiers ont été supprimés correctement, on écrirait :

```
while (@liste_fichiers) {
    my $fichier = shift @liste_fichiers;
    unlink $fichier or warn "Ne peut pas supprimer $fichier: $!\n";
}
```

Ici `@liste_fichier` est évaluée dans le contexte booléen induit par la boucle `while`, de telle sorte que Perl évalue le tableau pour voir si sa valeur est vraie ou fausse. Cette valeur est vraie tant que la liste contient des éléments et devient fausse dès que le dernier élément est enlevé (par l'opérateur `shift`). Notez que la liste n'est pas évaluée dans un contexte scalaire. Nous indiquons au tableau qu'il est scalaire et lui demandons ce qu'il vaut dans ce contexte.

N'essayez pas d'utiliser `defined @files` pour ce faire. La fonction `defined` regarde si le scalaire est égal à `undef`, et un tableau n'est pas un scalaire. Un simple test booléen suffit dans ce cas.

Contexte vide

Un autre type particulier des contextes scalaires est le contexte *vide*. Non seulement il se moque de la valeur à retourner, mais en plus il ne *veut* même pas de valeur de retour. Du point de vue du fonctionnement des fonctions, il n'est pas différent des autres contextes scalaires, mais par le switch **`-w`** de la ligne de commande, le compilateur Perl prévient de l'utilisation d'une expression sans effet secondaire à un endroit qui ne veut pas d'une valeur, comme dans une phrase qui ne renvoie pas de valeur. Par exemple, si une chaîne est employée comme phrase :

```
"Camel Lot";
```

on peut obtenir un avertissement tel que :

```
Useless use of a constant in void context in monprog line 123;
```

Contexte interpolatif

Nous avons déjà mentionné que les apostrophes doubles autour d'une chaîne permettent une interpolation des variables et une interprétation des antislashs, mais le contexte interpolatif (souvent appelé « contexte double quotes ») ne s'applique pas qu'aux chaînes entre doubles apostrophes. Les autres constructions à double quotes concernent l'opérateur général apostrophe inverse `qx//`, l'opérateur de recherche de motif `m//` et

l'opérateur de substitution `s///` et l'opérateur d'expression régulière `qr//`. En fait, l'opérateur de substitution fait une interpolation de sa partie gauche avant de faire la recherche de motif, puis à chaque motif trouvé fait une interpolation de sa partie droite.

Le contexte interpolatif n'intervient qu'entre apostrophes, ou pour des choses qui fonctionnent de la même façon, et il n'est donc peut être pas vraiment correct de l'appeler contexte au même titre que les contextes scalaires ou les contextes de liste (ou peut-être bien que si).

Valeur de liste et tableaux

Maintenant que nous avons parlé des contextes, nous pouvons parler des listes de valeurs et de leur comportement dans les différents contextes. Nous avons déjà vu les listes de littéraux, définies en séparant chaque valeur par une virgule (et en encadrant la liste par une paire de parenthèses). Comme il n'est pas génant (presque jamais) d'ajouter des parenthèses supplémentaires, la forme syntaxique d'une liste s'écrit :

(*LISTE*)

Nous venons de voir que la forme *LISTE* indique un terme qui retourne une liste à son argument, mais une liste littérale simple est l'exception à cette règle en ce qu'elle fournit un contexte de liste uniquement quand la liste elle-même est dans un contexte de liste. La valeur d'une liste litérale dans un contexte de liste est simplement les valeurs de chaque élément séparées par une virgule dans l'ordre spécifié. Comme un terme dans une expression, une liste litérale empile simplement des valeurs temporaires sur la pile Perl, elles seront utilisées ensuite par l'opérateur qui attend cette liste.

Dans un contexte scalaire, la valeur d'une liste est la valeur de son dernier élément, comme avec l'opérateur virgule du C qui ignore toujours la valeur à gauche et renvoie celle de droite (en faisant référence à ce que nous disions plus haut, la partie à gauche de la virgule fournit un contexte vide). L'opérateur virgule étant associatif à gauche, une liste de valeurs séparées par des virgules fournit toujours le dernier élément car la virgule oublie chaque valeur précédente. Par exemple :

```
@truc = ("un", "deux", "trois");
```

affecte toute la liste de valeurs au tableau `@truc`, mais :

```
$truc = ("un", "deux", "trois");
```

n'affecte que la valeur « `trois` » à la variable `$truc`. Comme pour le tableau `@liste_fichiers`, l'opérateur virgule sait s'il est dans un contexte scalaire ou de liste et adapte son comportement en conséquence.

C'est important de souligner qu'une liste est différente d'un tableau. Une variable tableau connaît aussi son contexte, et dans un contexte de liste retourne la liste de ses valeurs internes comme une liste littérale. Mais dans un contexte scalaire elle retourne simplement sa longueur. Le code suivant affecte la valeur 3 à la variable `$truc` :

```
@truc = ("un", "deux", "trois");
$truc = @truc;
```

Si vous pensiez obtenir la valeur « `trois` », vous avez généralisé un peu vite car en fait Perl a détecté le contexte scalaire et n'a empilé que la longueur du tableau. Aucun terme ou opérateur n'empilera des valeurs de liste dans un contexte scalaire. Il empilera une

valeur scalaire, qui ne sera certainement pas la dernière valeur de la liste qu'il retournerait dans un contexte de liste, valeur scalaire qui est ici la longueur du tableau. Il faut bien comprendre cet exemple (sinon, relisez le paragraphe car c'est important).

Retournons aux vraies *LISTE*, celles qui induisent un contexte de liste. Jusqu'à présent nous avons soutenu que le contexte *LISTE* ne contient que des littéraux. Mais en fait, toute expression qui retourne une valeur peut être utilisée à l'intérieur d'une liste. Les valeurs ainsi utilisées peuvent être des valeurs scalaires ou des listes de valeurs. Le contexte *LISTE* fait automatiquement une interpolation des sous-listes. C'est pourquoi, lorsqu'un contexte *LISTE* est évalué, chaque élément de la liste est évalué dans ce contexte et la valeur de la liste résultante est interpolée dans *LISTE* comme si chaque élément individuel était un membre de *LISTE*. Les tableaux perdent ainsi leur identité dans *LISTE*. La liste :

```
(@machin,@chose,&sousProgramme)
```

contient tous les éléments de `@machin`, suivis de tous les éléments de `@chose`, suivis de tous les éléments renvoyés par le sous-programme `sousProgramme` lorsqu'il est appelé dans un contexte de liste. Notez que si l'un quelconque des éléments est nul, son interpolation ne compte pas. La liste nulle est représentée par `()`. Son interpolation dans une liste n'a pas d'effet. Ainsi `((),(),())` est équivalent à `()`. De même, l'interpolation d'un tableau sans élément donne, à ce stade, la même chose que s'il n'avait pas été interpolé.

Une conséquence est que l'on peut mettre une virgule optionnelle à la fin de toute liste. Ceci facilitera ultérieurement les ajouts d'éléments.

```
@nombres = (
    1,
    2,
    3,
);
```

Le mot `qw`, que nous avons mentionné plus tôt, permet aussi d'entrer une liste littérale. Il construit quelque chose d'équivalent à une chaîne entre apostrophes éclatée sur plusieurs lignes. Par exemple :

```
@machin = qw(
    pomme       banane      carambole
    orange      goyave      kumquat
    mandarine   nectarine   pêche
    poire       persimmon   prune
);
```

Remarquez que ces parenthèses ne sont pas ordinaires et fonctionnent comme des apostrophes. On aurait pu aussi mettre des signes inférieur-supérieur, des accolades ou des slashs (mais les parenthèses c'est joli).

Une liste de valeurs peut aussi être indicée comme un tableau standard. Vous devez alors mettre la liste entre parenthèses (des vraies) pour éviter toute ambiguïté. Si cela sert souvent pour obtenir une valeur du tableau, c'est aussi une tranche de la liste et la forme syntaxique s'écrit :

```
(LISTE)[LISTE]
```

Exemples :

```
# Stat renvoie une valeur liste.
```

```
$modification_time = (stat($fichier))[8];

# ERREUR DE SYNTAXE ICI.
$modification_time = stat($fichier)[8];  # OUPS, PAS DE PARENTHESES

# Trouver un chiffre hexa.
$chiffrehexa = ('a','b','c','d','e','f')[$chiffre-10];

# Un "opérateur virgule inverse".
return (pop(@machin),pop(@machin))[0];

# Affectation de plusieurs valeurs à l'aide d'une tranche.
($jour, $mois, $annee) = (localtime)[3,4,5];
```

Affectation de liste

Une liste est affectée si chacun de ses éléments est affecté :

```
($a, $b, $c) = (1, 2, 3);

($map{rouge}, $map{vert}, $map{bleu}) = (0xff0000, 0x00ff00, 0x0000ff);
```

Il est possible d'affecter à `undef` dans une liste. C'est utile pour ignorer certaines valeurs de retour d'une fonction :

```
($dev, $ino, undef, undef, $uid, $gid) = stat($fichier);
```

L'élément final d'une liste peut être un tableau ou un hachage :

```
($a, $b, @reste) = split;
my ($a, $b, %reste) = @arg_list;
```

Vous pouvez mettre un tableau ou un hachage dans la liste affectée en sachant que le premier de ce type rencontré absorbera les valeurs restantes et les variables restantes seront initialisées à `undef`. Ceci peut être utile dans une initialisation de type `my` ou `local`, où on veut que les tableaux soient vides.

Il est aussi possible d'affecter une liste vide :

```
() = fonction_bidon();
```

De cette façon, la fonction est appelée dans un contexte de liste, mais on laisse tomber les valeurs renvoyées. Si vous aviez fait l'appel sans affectation, la fonction aurait été évaluée dans un contexte vide, qui est un contexte scalaire, et se serait comportée complètement différemment.

L'affectation de liste dans un contexte scalaire retourne le nombre d'éléments que fournit la partie *droite* de l'affectation :

```
$x = ( ($machin,$truc) = (7,7,7) );# $x contient 3, pas 2
$x = ( ($machin,$truc) = f() );     # $x contient le nombre d'éléments de f()
$x = ( () = f() );                  # idem
```

C'est pratique lorsqu'on veut faire l'affectation d'une liste dans un contexte booléen, car la plupart des fonctions liste retourne une liste nulle lorsqu'elle se termine, valeur qui affectée à un scalaire produit la valeur 0 qui vaut faux dans ce contexte. Voici un exem-

ple d'utilisation dans une boucle `while` :

```
while (($login, $password) = getpwent) {
    if (crypt($login, $password) eq $password) {
        print "$login a un mot de passe non sécurisé!\n";
    }
}
```

Longueur d'un tableau

Le nombre d'éléments du tableau `@jours` est donné par l'évaluation de `@jours` dans un contexte scalaire :

```
@jours + 0;      # force implicitement @jours dans un contexte scalaire
scalar(@jours)   # force explicitement @jours dans un contexte scalaire
```

Attention, cela ne fonctionne que pour les tableaux. Cela ne fonctionne pas pour les listes de valeurs en général. Une liste avec des virgules comme séparateur évaluée dans un contexte scalaire retournera la dernière valeur, comme l'opérateur virgule en C. Mais puisqu'il n'est presque jamais nécessaire de connaître la longueur d'une liste en Perl ce n'est pas un problème.

`$#jours` est très proche de l'évaluation scalaire de `@jours`. Cette dernière renvoie l'indice du dernier élément du tableau, ou un de moins que la longueur puisqu'il existe (habituellement) un élément d'indice zéro. L'affectation de `$#jours` change la longueur du tableau. Raccourcir un tableau de la sorte détruit les valeurs intermédiaires. Vous pouvez être plus efficace en surdimensionnant à l'avance un tableau destiné à s'agrandir (pour agrandir un tableau, on peut lui affecter un élément au-delà de sa fin). Un tableau peut être tronqué par l'affectation de la liste nulle `()`. Les deux instructions suivantes sont équivalentes :

```
@nimportequoi = ();
$#nimportequoi = -1;
```

Et celle qui suit est toujours vraie :

```
scalar(@nimportequoi) == $#nimportequoi + 1;
```

La troncation d'un tableau ne récupère pas la mémoire ainsi libérée. Il faut faire un `undef(@nimportequoi)` pour récupérer sa mémoire dans le processus courant. Vous ne pouvez vraisemblablement pas la retouner au système d'exploitation car peu de systèmes en sont capables.

Hachages

Comme nous l'avons déjà dit, un hachage n'est qu'un type spécial de tableau dans lequel on retrouve les valeurs par une chaîne-clef au lieu d'un indice numérique. En fait, on définit des associations entre les clefs et les valeurs, et c'est pourquoi les hachages sont souvent appelés tableaux associatifs par les personnes assez courageuses pour taper sur un clavier.

Il n'existe pas vraiment de syntaxe particulière aux hachages en Perl, mais si une liste ordinaire est affectée à un hachage, chaque paire de valeur de la liste sera prise comme

une association clef/valeur.

```
%map = ('rouge',0xff000,'vert',0x00ff00,'bleu',0x0000ff);
```

Ce qui a le même effet que :

```
%map = ();              # d'abord initialiser le hachage
$map{rouge} = 0xff0000;
$map{vert}  = 0x00ff00;
$map{bleu}  = 0x0000ff;
```

Il est souvent plus lisible d'utiliser l'opérateur => entre la clef et sa valeur. L'opérateur => est synonyme de la virgule, mais il est plus clair visuellement et permet aussi une séparation d' identifiant à gauche (comme les identifiants entre accolades ci-dessus) ; l'initialisation des variables hachages devient plus simple :

```
%map = (
    rouge => 0xff0000,
    vert  => 0x00fff0,
    bleu  => 0x0000ff,
);
```

ou pour initialiser des références banalisées de hachage utilisées comme des enregistrements :

```
$rec = {
    NOM       => 'John Smith'
    GRADE     => 'Captain'
    MATRICULE => '951413',
};
```

ou pour utiliser des appels complexes de fonctions par un nom :

```
$field = $query->radio_group(
                    NOM        => 'group_name',
                    VALEURS    => ['eenie','meenie','minie'],
                    DEFAUT      => 'meenie',
                    SAUTDELIGNE => 'true',
                    LABELS      => %labels,
                );
```

Mais n'allons pas trop vite. Revenons à nos hachages.

Une variable hachage (`%hachage`) peut être utilisée dans un contexte de liste, toutes ses paires clef/valeurs étant interpolées dans la liste. Mais ce n'est pas parce que le hachage a été initialisé dans un certain ordre que les valeurs sont retournées dans le même. Les hachages sont implémentés au niveau interne en utilisant des tables de hachage pour une recherche rapide, ce qui signifie que l'ordre dans lequel les données sont stockées dépend du type de fonction de hachage utilisée pour déterminer la place de la paire clef/valeur. Ainsi, les entrées sont apparemment retournées dans un ordre quelconque (au niveau d'une paire clef/valeur, les deux éléments sont bien sûr rendus dans le bon ordre). Pour des exemples de réordonnancement en sortie, voir la fonction `keys` au chapitre 29.

Si un hachage est évalué dans un contexte scalaire, une valeur vraie est renvoyée si et seulement si le hachage contient des paires clefs/valeurs. (S'il existe plusieurs paires, la valeur renvoyée est une chaîne contenant le nombre de cellules utilisées (*buckets*) et le nombre

de cellules allouées, séparés par un slash. Ceci n'est utile que pour savoir si l'algorithme résident de Perl traite correctement ou non l'ensemble de données. Par exemple, on entre 10 000 éléments dans un hachage, mais l'évaluation de `%HASH` dans un contexte scalaire révèle « `1/8` », cela signifie que seulement une des 8 cellules a été touchée, et qu'elle contient probablement les 10 000 éléments. Ce n'est pas censé se produire.

Pour trouver le nombre de clefs dans un hachage, utilisez la fonction `keys` dans un contexte scalaire : `scalar(keys(%HASH))`.

Il est possible d'émuler un tableau multidimensionnel en spécifiant plus d'une clef dans les accolades séparées par une virgule. Les clefs listées sont concaténées ensemble séparées par le caractère `$;` (`$SUBSCRIPT_SEPARATOR`) qui a la valeur `chr(28)` par défaut. La clef résultante devient une clef du hachage. Ces deux lignes font la même chose :

```
$habitants{ $pays, $département } = $résultat_recensement;
$habitants{ join $; =>; $pays, $département } = $résultat_recensement;
```

Cette fonctionalité fut à l'origine implémentée pour le traducteur *a2p* (awk vers perl). Aujourd'hui, vous utiliseriez plus justement un tableau multidimensionnel comme il est décrit au chapitre 9, *Structures de données*. Un usage encore pratique de cet ancien style est la liaison des hachages aux fichiers DBM (voir `DB_File` au chapitre 32, *Modules standards*), qui n'implémentent pas les clefs multidimensionnelles.

Ne confondez pas l'émulation multidimensionnelle des hachages avec les tranches de tableaux. L'un est une valeur scalaire, l'autre représente une liste :

```
$hachage{ $x, $y, $z }      # une valeur simple
@hachage{ $x, $y, $z }      # une tranche de trois valeurs
```

Typeglobs et handles de fichiers

Perl utilise un type spécial appelé *typeglob* qui contient la table des types Perl pour ce symbole. (La table des symboles `*foo` contient les valeurs de `$foo`, `@foo`, `%foo`, `&foo` et d'autres interprétations de `foo`.) Le préfixe de typage d'un typeglob est * car il représente tous les types.

Les typeglobs (ou leur références) sont toujours utilisés pour passer ou stocker des handles de fichier. Pour sauvegarder un handle de fichier il faut taper :

```
$fh = *STDOUT;
```

ou comme une vraie référence, de cette manière :

```
$fh = \*STDOUT;
```

C'est aussi le moyen de créer un handle de fichier local. Par exemple :

```
sub newopen {
    my $path = shift;
    local *FH;       # et non my ou our!
    open (FH, $path) || return undef;
    return *FH;      # et non \*FH!
}
$fh = newopen('/etc/passwd');
```

Voir la fonction `open` pour générer d'autres handles de fichiers.

Cependant, la principale utilisation des typeglobs est actuellement de servir d'alias entre deux entrées symboliques. C'est comme un surnom. Si on écrit

```
*machin = $truc;
```

tout ce qui s'appelle « machin », est synonyme de tout ce qui s'appelle « truc ». L'alias peut porter sur un seul type des variables du nom en affectant une référence :

```
*machin = $truc;
```

$machin devient alias de $truc, mais@machin n'est pas synonyme de @truc, ni %machin de %truc. Tout ceci ne concerne que les variables globales, c'est-à-dire définies dans un paquetage ; les variables lexicales ne peuvent être accédées à travers la table des symboles. Le mécanisme d'import/export est entièrement basé sur ce principe, l'alias ne présuppose en effet aucune appartenance à un module spécifique. Cette instruction :

```
local *Ici::bleu = \$Ailleurs::vert;
```

fait de $Ici::bleu un alias pour $Ailleurs::vert, mais ne fait pas de @Ici::bleu un alias pour @Ailleurs::vert, ni %Ici::bleu un alias pour %Ailleurs;::vert. Heureusement, toutes ces manipulations compliquées de typeglobs sont transparentes la plupart du temps. Voir le chapitre 8, et *Tables de symboles* au chapitre 10 et le chapitre 11, *Modules*, pour un développement sur ces sujets.

Opérateurs d'entrée

Nous allons maintenant présenter plusieurs opérateurs d'entrée qui sont analysés comme des termes. En fait, on les appelle parfois des pseudo-littéraux car ils agissent, dans bien des cas, comme des chaînes protégées (les opérateurs de sortie comme print fonctionnent comme des opérateurs de liste et sont traités au chapitre 29.)

Opérateur d'exécution de commande (Backtick)

Premièrement il y a l'opérateur de ligne de commande, aussi appellé opérateur *backtick*, et qui ressemble à ceci :

```
$info = `finger $user`;
```

Une chaîne protégée par des apostrophes inverses procède tout d'abord à une interpolation des variables comme dans une chaîne entre apostrophes doubles. Le résultat est alors interprété comme une commande par le shell, et la sortie de la commande devient la valeur de l'identifiant (ceci fonctionne comme certains opérateurs similaires des shells UNIX). Dans un contexte scalaire, le résultat est une chaîne simple contenant toutes les sorties de la commande. Dans un contexte de liste, une liste de valeurs est renvoyée, une pour chaque ligne sortie par la commande. (On peut utiliser $/ pour avoir un autre caractère de terminaison de ligne.)

La commande est interprétée chaque fois que l'identifiant est évalué. La valeur numérique du statut de la commande est stockée dans $? (voir le chapitre 28 pour l'interprétation de $?, connu aussi sous le nom de $CHILD_ERROR). À la différence de *csh*, aucune transformation n'est faite sur les données renvoyées ; les sauts de ligne restent des sauts de ligne. À l'inverse de tous les shells, les apostrophes simples n'empêchent pas l'interprétation des noms de variables. Pour passer un $ au shell il faut le cacher par un anti-

slash. Le `$user` de notre exemple ci-dessus est interpolé par Perl, mais non par le shell (la commande lançant un processus shell, consultez le chapitre 23, *Sécurité*, pour les problèmes de sécurité).

La forme généralisée pour les apostrophes inverses est `qx//` (pour *quoted execution*, exécution protégée), mais l'opérateur fonctionne exactement de la même façon que les apostrophes inverses ordinaires. Il suffit de choisir ses caractères de protection. Comme avec les pseudo-fonctions de protection, si vous choisissez un simple guillemet comme délimiteur, la chaîne de commande n'est pas interpolée :

```
$perl_info  = qx(ps $$);            # Variable $$ de Perl
$shell_info = qx'ps $$';            # Variable $$ du shell
```

Opérateur de lecture de ligne (Angle)

L'opérateur de lecture de ligne est l'opérateur le plus utilisé, aussi connu sous le nom d'opérateur inférieur-supérieur, ou fonction `readline`. L'évaluation d'un handle de fichier entre inférieur-supérieur (par exemple `<STDIN>`) donne la ligne suivante du fichier associé. Le saut de ligne est inclus, donc en accord avec les critères de véracité de Perl, chaque ligne lue a une valeur *vrai*, et lorsque le programme atteint la fin du fichier, l'opérateur *angle* retourne `undef`, c'est-à-dire *faux*. On affecte d'habitude la valeur lue à une variable, mais il existe un cas où une affectation automatique se produit. Si, et *seulement* si, l'opérateur de lecture de ligne est la seule chose présente dans le test de boucle d'un `while`, la valeur est automatiquement affectée à la variable spéciale `$_`. La valeur affectée est alors testée pour savoir si elle est définie (cela peut vous paraître idiot, mais vous utiliserez cette construction dans presque tous vos scripts Perl). Bref, les lignes suivantes sont toutes équivalentes :

```
while (defined($_ = <STDIN>)) { print $_; }  # méthode longue
while ($_ = <STDIN>) { print; }              # utilise $_ explicitement
while (<STDIN>) { print; }                   # méthode courte
for (;<STDIN>;) { print; }                   # boucle while déguisée
print $_ while defined($_ = <STDIN>);        # instruction modifiée longue
print while $_ = <STDIN>;                    # utilise $_
print while <STDIN>;                         # instruction modifiée courte
```

Souvenez-vous que cette astuce requiert une boucle `while`. Si vous utilisez l'opérateur ailleurs, vous devez affecter le résultat explicitement pour garder la valeur.

```
while (<FH1> && <FH2>) {...}            # mauvais : perd les deux entrées
if (<STDIN>)      { print; }            # mauvais, affiche l'ancienne
                                        # valeur de $_
if ($_ = <STDIN>) { print; }            # ne teste pas si $_ est défini
if (defined($_ = <STDIN>)) { print; }   # meilleur
```

Quand la variable `$_` est implicitement affectée dans une boucle, c'est la variable globale dont il s'agit. Vous pouvez protéger la valeur de `$_` de la manière suivante :

```
while (local $_ = <STDIN>) { print; }    # utilise local $_
```

L'ancienne valeur est rétablie à la sortie de la boucle. `$_` peut toujours être accédée depuis la boucle. Pour éviter toute confusion, il est préférable d'utiliser une variable lexicale :

```
while (my $ligne = <STDIN>) { print $ligne; }    # variable privée
```

(Ces deux boucles `while` testent si l'affectation est `defined`, car `my` et `local` ne changent pas son comportement habituel.) Les handles de fichiers `STDIN`, `STDOUT` et `STDERR` sont prédéfinis et ouverts. D'autres handles peuvent être créés avec les fonctions `open` ou `sysopen`. Voir la documentation de ces fonctions au chapitre 29 pour les détails.

Dans la boucle `while` ci-dessus, la ligne d'entrée est évaluée dans un contexte scalaire, chaque valeur est retournée séparément. Si on l'utilise dans un contexte de liste, une liste de toutes les lignes d'entrée restantes est retournée, chaque ligne étant un élément de liste. Un *important* espace de données peut ainsi être créé, et il faut donc l'utiliser avec précaution :

```
$une_ligne = <MYFILE>;      # Donne la première ligne.
@toutes_lignes = <MYFILE>;  # Donne le reste des lignes.
```

Il n'existe pas de magie particulière pour le `while` associé à la forme liste de l'opérateur d'entrée, car la condition d'une boucle `while` est toujours un contexte scalaire (comme toutes les conditions).

L'utilisation du handle de fichier nul à l'intérieur de l'opérateur inférieur-supérieur (ou opérateur angle) est particulière et peut être utilisée pour émuler la ligne de commande de programmes standards d'UNIX tels que *sed* et *awk*. Quand on lit les lignes depuis `<>`, toutes les lignes de tous les fichiers mentionnés sur la ligne de commande sont renvoyées. Si aucun fichier n'était spécifié, c'est l'entrée standard qui est renvoyée à la place et le programme peut ainsi être facilement inséré entre des processus pour former un pipe.

Voici comment cela fonctionne : la première fois que `<>` est évalué, le tableau `@ARGV` est contrôlé, et s'il est nul, `$ARGV[0]` est mis à « - », qui donne l'entrée standard quand on l'ouvre. Le tableau `@ARGV` est traité comme une liste de noms de fichiers. La boucle

```
while (<>) {
    ...                     # code pour chacune des lignes
}
```

est équivalente au pseudo-code Perl suivant :

```
while (@ARGV and $ARGV[0] =~ /^-/) {
    $_=shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-V/)     { $verbose++ }
    ...              # autres alternatives
}
while (<>) {
    ...              # du code dans chaque ligne
}
```

à part une meilleure lisibilité, le résultat reste identique. Le tableau `@ARGV` est décalé et le nom du fichier courant est stocké dans la variable `$ARGV`. Le handle de fichier `ARGV` est aussi utilisé en interne ; `<>` est simplement un synonyme de `<ARGV>`, qui est magique (le pseudo-code ci-dessus ne fonctionne pas parce qu'il traite `<ARGV>` comme non-magique).

Il est possible de modifier `@ARGV` avant l'instruction `<>` du moment que le tableau est construit avec les noms de fichier que vous attendez. Le nom de fichier « - » qui représente l'entrée standard, peut être ouvert avec la fonction `open`, ainsi que des flux plus

ésotériques comme « `gzip -dc < file.gz|` »). Les numéros de ligne (`$.`) s'incrémentent comme s'il n'existe qu'un fichier en entrée. (Mais observez l'exemple ci-dessous sur `eof` pour voir comment on réinitialise les numéros de ligne à chaque fichier.)

Pour affecter `@ARGV` à une liste de fichiers, c'est direct :

```
# par défaut: le fichier README si aucun argument n'est donné
@ARGV = ("README") unless @ARGV;
```

Pour passer des options dans un script, on peut utiliser un des modules Getopt ou mettre une boucle initiale telle que :

```
while ($_ = $ARGV[0], /^-/) {
    shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/)     { $verbose++  }
    ...             # autres options
}
while (<>) {
    ...             # code pour chacune des lignes
}
```

Le symbole `<>` ne renvoie faux qu'une seule fois. En cas d'appel ultérieur, une autre liste `@ARGV` est supposée être traitée, et si `@ARGV` n'est pas initialisé les lectures se font depuis `STDIN`.

Si la chaîne entre l'opérateur angle est une variable (par exemple, `<$machin>`), alors celle-ci contient le nom du fichier à lire ou une référence à celui-ci. Par exemple :

```
$fh = \*STDIN;
$ligne = <$fh>;
```

ou :

```
open($fh, "<donnees.txt>");
$line = "<$fh>";
```

Opérateur de globalisation des noms de fichier

Vous vous souciez peut-être de ce qui peut arriver à un opérateur de lecture de ligne si vous mettez quelque chose de bizarre entre l'inférieur et le supérieur. Il est alors transformé en un opérateur différent. Si la chaîne à l'intérieur de l'opérateur angle est différente du nom d'un handle de fichier ou d'une variable scalaire (même s'il agit d'espace), elle sera interprétée comme étant un motif de fichier à « globaliser ».[19] Le motif de recherche s'applique aux fichiers du répertoire courant (ou dans le répertoire spécifié dans le motif glob lui-même), et les fichiers correspondants sont retournés par l'opérateur. Comme pour les lectures de ligne, les noms sont renvoyés un à un dans un contex-

19. Cela n'a rien à voir avec les typeglobs déjà cités, si ce n'est qu'ils utilisent tous deux le caractère * à la façon d'un joker. Le caractère * prend le surnom « glob » lorsqu'il est utilisé de cette manière. Avec les typeglobs, on globalise les symboles homonymes d'une table de symboles. Avec un glob de nom de fichier (global), on effectue une correspondance par des jokers des fichiers d'un répertoire, comme le font de nombreux shells.

te scalaire, ou tous à la fois dans un contexte de liste. Ce dernier usage est en fait prédominant. Il est courant de voir :

```
my @fichiers = <*.xml>;
```

Comme pour d'autre types de pseudo-constantes, un premier niveau d'interpolation est effectué, mais on ne peut pas écrire `<$machin>` car il s'agit d'un handle de fichier indirect, comme nous l'avons déjà vu. Dans d'anciennes versions de Perl, les programmeurs inséraient des accolades pour forcer une interprétation de nom de fichier glob : `<${machin}>`. De nos jours, on considère qu'il est plus propre d'appeler la fonction interne directement par `glob($machin)`, ce qui aurait dû être le cas dès l'origine. On écrirait donc :

```
@fichiers = glob("*.xml");
```

si vous préférez.

Que l'on utilise la fonction `glob` ou sa forme ancienne avec l'opérateur d'angle, l'opérateur de globalisation fonctionne de la même façon dans une boucle que l'opérateur `while` en affectant le résultat à `$_`. (Ce fut la première raison de surcharger cet opérateur.) Par exemple, si vous vouliez changer les autorisations de fichiers de code C, vous écririez :

```
while (glob "*.c") {
    chmod 0644, $_;
}
```

ce qui est équivalent à :

```
while (<*.c>) {
    chmod 0644, $_;
}
```

La fonction `glob` était implémentée comme une commande shell dans les anciennes versions de Perl (aussi dans les anciennes versions d'Unix), était plus coûteuse en ressources et ne fonctionnait pas de la même façon sur tous les systèmes. C'est aujourd'hui une fonction prédéfinie plus fiable et plus rapide. Voyez la description du module `File::Glob` au chapitre 32 pour modifier son comportement par défaut pour, par exemple, traiter les espaces dans ces arguments, l'expansion de certains caractères (tilde ou accolade), l'insensiblité à la casse ou bien le classement des valeurs retournées, entre autres choses.

Bien sûr, la façon la plus courte et la moins lisible pour faire la commande `chmod` ci-dessus, est d'utiliser la globalisation comme un opérateur sur liste :

```
chmod 0644, <*.c>;
```

Un glob n'évalue son argument que quand il démarre une nouvelle liste. Toutes les valeurs doivent être lues avant son exécution. Dans un contexte de liste, ce n'est pas très important, puisqu'on les a toutes automatiquement. Mais dans un contexte scalaire, l'opérateur renvoie la prochaine valeur à chaque appel, ou une valeur fausse si on arrive au bout. Attention, faux n'est retourné qu'une seule fois. Ainsi, si l'on attend une valeur unique pour un glob, il vaut mieux écrire :

```
($fichier) = <blurch*>;  # contexte de liste
```

au lieu de :

```
$fichier = <blurch*>;    # contexte scalaire
```

car la première forme avale tous les noms de fichier correspondants et réinitialise l'opérateur, alors que ce que renvoie la seconde alterne entre un nom de fichier et faux.

Pour interpoler des variables, il faut absolument utiliser l'opérateur `glob`, car l'ancienne syntaxe peut être confondue avec la notation indirecte des handles de fichiers. Mais à ce niveau, il devient évident que la frontière entre les termes et les opérateurs est un peu floue.

```
@files = <$dir/*.[ch]>;          # à éviter
@files = glob("$dir/*.[ch]");    # appelle glob en tant que fonction
@files = glob $un_motif;         # appelle glob en tant qu'opérateur
```

Nous ne mettons pas les parenthèses dans le second exemple pour montrer que `glob` peut être utilisé comme un opérateur *unaire*, c'est-à-dire un opérateur préfixe avec un seul argument. L'opérateur `glob` est un exemple d'*opérateur unaire défini*, qui n'est qu'un des types d'opérateurs dont nous parlerons dans le prochain chapitre. Ensuite nous parlerons des opérations de recherche de motif, qui ressemblent elles aussi à des termes, mais fonctionnent comme des opérateurs.

3

Opérateurs unaires et binaires

Dans le chapitre précédent, nous avons parlé des différents types de termes que vous pouvez utiliser dans une expression. Mais pour être honnête, les termes isolés sont un peu ennuyeux. Beaucoup de termes sont de joyeux fêtards ; ils aiment comuniquer les uns avec les autres. Un terme typique ressent une forte envie de s'identifier avec d'autres termes, ou de les influencer de différentes manières. Cependant, il existe de nombreuses sortes d'interactions sociales et de nombreux niveaux d'implication. En Perl, ces relations sont exprimées à l'aide d'opérateurs.

Il faut bien que la sociologie serve à quelque chose.

D'un point de vue mathématique, les opérateurs sont des fonctions ordinaires avec une syntaxe spéciale. D'un point de vue linguistique, les opérateurs sont juste des verbes du troisième groupe. Mais comme tout linguiste vous le dira, les verbes du troisième groupe sont généralement ceux que l'on utilise le plus souvent. C'est important du point de vue de la théorie de l'information, car les verbes du troisième groupe sont généralement plus courts et plus efficaces tant pour la production que pour la reconnaissance.

D'un point de vue pratique, les opérateurs sont maniables.

On distingue plusieurs variétés d'opérateurs, en fonction de leur *arité* (le nombre d'opérandes qu'ils prennent), leur *précédence* (leur capacité à prendre leurs opérandes aux opérateurs voisins), et leur *associativité* (agissent-ils de gauche à droite ou de droite à gauche quand ils sont associés à des opérateurs de même précédence).

Les opérateurs Perl se présentent sous trois arités : *unaire*, *binaire* ou *ternaire*. Les opérateurs unaires sont toujours des opérateurs préfixes (exceptés les opérateurs de post-incrémentation et de post-décrémentation).[1] Tous les autres sont des opérateurs infixes — à moins que vous ne comptiez les opérateurs de liste, qui peuvent précéder autant d'arguments que vous voulez. Cependant la plupart des gens préfèrent voir les opérateurs de liste comme des fonctions normales autour desquelles on peut oublier de mettre des parenthèses. Voici quelques exemples :

1. Bien que l'on puisse également voir les différents types de guillemets et de parenthèses (crochets, accolades) comme des opérateurs « circonfixes » qui englobent et délimitent des termes.

```
! $x                # un opérateur unaire
$x * $y             # un opérateur binaire
$x ? $y : $z        # un opérateur ternaire
print $x, $y, $z    # un opérateur de liste
```

La précédence d'un opérateur détermine sa force d'attraction. Les opérateurs de plus haute précédence capturent les arguments qui les entourent avant les opérateurs de moindre précédence. L'exemple typique sort tout droit des mathématiques élémentaires, où la multiplication est prioritaire sur l'addition :

```
2 + 3 * 4           # vaut 14, et non 20
```

L'ordre dans lequel sont exécutés deux opérateurs de même précédence dépend de leur associativité. Les règles suivent en partie les conventions des mathématiques :

```
2 * 3 * 4           # signifie (2 * 3) * 4, associatif à gauche
2 ** 3 ** 4         # signifie 2 ** (3 ** 4), associatif à droite
2 != 3 != 4         # illégal, car non associatif
```

Le *tableau 3-1* liste l'associativité et l'arité des opérateurs Perl par ordre de précédence décroissante.

Tableau 3-1. Précédence des opérateurs

Associativité	Arité	Classe de précédence
Non associatifs	0	Termes et opérateurs de liste (vers la gauche)
Gauche	2	`->`
Non associatifs	1	`++ --`
Droite	2	`**`
Droite	1	`! ~ \+` unaire et `-` unaire
Gauche	2	`=~ !~`
Gauche	2	`* / % x`
Gauche	2	`+ - .`
Gauche	2	`<< >>`
Droite	0,1	Opérateurs unaires nommés
Non associatifs	2	`< > <= >= lt gt le ge`
Non associatifs	2	`== != <=> eq ne cmp`
Gauche	2	`&`
Gauche	2	`| ^`
Gauche	2	`&&`
Gauche	2	`||`
Non associatifs	2	`.....`
Droite	3	`?:`
Droite	2	`= += -= *=` et suivants
Gauche	2	`, =>`
Droite	0+	Opérateurs de liste (vers la droite)
Droite	1	`not`

Tableau 3-1. Précédence des opérateurs (suite)

Associativité	Arité	Classe de précédence
Gauche	2	`and`
Gauche	2	`or xor`

Vous pourriez penser qu'il y a bien trop de niveaux de précédence à se souvenir. Et bien vous avez raison. Heureusement, deux choses jouent en votre faveur. D'abord les niveaux de précédence tels qu'ils sont définis sont en général assez intuitifs, en supposant que vous n'êtes pas psychotique. Ensuite, si vous êtes seulement névrosé, vous pouvez toujours ajouter des parenthèses supplémentaires pour calmer votre angoisse.

Notez également que tous les opérateurs empruntés à C conservent les mêmes relations de précédence entre eux, même quand les règles de précédence de C sont légèrement biscornues. (Cela rend Perl d'autant plus facile à apprendre pour les personnes qui connaissent C ou C++. Peut-être même pour celles qui connaissent Java.)

Les sections qui suivent couvrent ces opérateurs dans leur ordre de précédence. À de très rares exceptions, ils opèrent tous uniquement sur des valeurs scalaires et non sur des listes. Nous indiquerons les exceptions quand elles se présenteront.

Bien que les références soient des valeurs scalaires, utiliser la plupart de ces opérateurs sur des références n'a pas beaucoup de sens, car la valeur numérique d'une référence n'a de signification que dans les profondeurs de Perl. Néanmoins, si une référence pointe sur un objet d'une classe qui autorise la surcharge d'opérateurs, vous pouvez utiliser ces opérateurs sur cet objet ; si la classe a défini une surcharge pour tel ou tel opérateur, cela décrit comment l'objet sera traité par cet opérateur. C'est ainsi par exemple que les nombres complexes sont implémentés en Perl. Pour plus d'informations sur la surcharge d'opérateurs, voir le chapitre 13, *Surcharge*.

Termes et opérateurs de listes (vers la gauche)

En Perl, tout terme est de précédence maximale. Les termes comprennent les variables, les apostrophes et les opérateurs de type quote, la plupart des expressions entre parenthèses, crochets ou accolades, et toute fonction dont les arguments sont entre parenthèses. En fait, si on regarde les choses ainsi, il n'y a pas vraiment de fonctions, mais seulement des opérateurs de liste et des opérateurs unaires qui se comportent comme des fonctions parce que vous avez mis des parenthèses autour de leurs arguments. Tout cela n'empêche pas le chapitre 29 de s'appeler *Fonctions*.

Maintenant lisez attentivement. Voici quelques règles très importantes qui simplifient grandement les choses, mais qui peuvent à l'occasion produire des résultats contraires à l'intuition pour les imprudents. Si un opérateur de liste (comme `print`) ou un opérateur unaire nommé (comme `chdir`) est suivi d'une parenthèse ouvrante comme token suivant (sans tenir compte des blancs), l'opérateur et ses arguments entre parenthèses ont la précédence la plus haute, comme s'il s'agissait d'un appel de fonction normal. La règle est la suivante : si cela *ressemble* à un appel de fonction, alors c'est un appel de fonction. Vous pouvez le faire ressembler à une non-fonction en préfixant les parenthèses avec un plus unaire, qui ne fait absolument rien sémantiquement parlant ; il ne convertit même pas les arguments en numérique.

Par exemple, puisque || a une précédence plus faible que chdir, nous aurons :

```
chdir $toto    || die;       # (chdir $toto) || die
chdir($toto)   || die;       # (chdir $toto) || die
chdir ($toto)  || die;       # (chdir $toto) || die
chdir +($toto) || die;       # (chdir $toto) || die
```

mais comme * a une précédence plus grande que chdir, nous aurons :

```
chdir $toto * 20;            # chdir ($toto * 20)
chdir($toto) * 20;           # (chdir $toto) * 20
chdir ($toto) * 20;          # (chdir $toto) * 20
chdir +($toto) * 20;         # chdir ($toto * 20)
```

De même pour n'importe quel opérateur numérique qui est également un opérateur unaire nommé, comme rand :

```
rand 10 * 20;                # rand (10 * 20)
rand(10) * 20;               # (rand 10) * 20
rand (10) * 20;              # (rand 10) * 20
rand +(10) * 20;             # rand (10 * 20)
```

En l'absence de parenthèses, la précédence d'opérateurs de liste comme print, sort, ou chmod est soit très haute soit très basse selon que vous regardez à gauche ou à droite de l'opérateur (c'est le sens du « vers la gauche » dans le titre de cette section). Par exemple, dans :

```
@tab = (1, 3, sort 4, 2);
print @tab;          # imprime 1324
```

Les virgules à droite de sort sont évaluées avant le sort, mais les virgules à sa gauche sont évaluées après. En d'autres termes, un opérateur de liste tend à avaler tous les arguments qui le suivent, puis à se comporter comme un simple terme pour l'expression qui le précède. Il vous faut encore être prudent avec les parenthèses :

```
# Ces arguments sont évalués avant de faire le print :
print($toto, exit);  # Évidemment pas ce que vous voulez.
print $toto, exit;   # Ici non plus.

# Ces lignes font le print avant d'évaluer l'exit :
(print $toto), exit; # C'est ce que vous voulez.
print($toto), exit;  # Ici aussi.
print ($toto), exit; # Et même ceci.
```

Le cas le plus simple pour se faire piéger, c'est quand vous utilisez des parenthèses pour grouper des arguments mathématiques, en oubliant que les parenthèses servent aussi à regrouper les arguments de fonctions :

```
print ($toto & 255) + 1, "\n";   # affiche ($toto & 255)
```

Cela ne fait probablement pas ce à quoi vous vous attendiez au premier coup d'œil. Heureusement, les erreurs de cette nature provoquent des avertissements comme « Useless use of addition in a void context » quand les avertissements sont activés (avec l'option de ligne de commande -w).

Les constructions do {} et eval {} sont aussi analysées comme des termes, de même que les appels de sous-programmes et de méthodes, les créateurs de tableaux et de hachages anonymes, [ ] et {}, et le créateur de sous-programme anonymes sub {}.

L'opérateur flèche

Tout comme en C et en C++, l'opérateur binaire `->`est un opérateur infixe de déréférencement. Si à droite on trouve un indice de tableau entre `[...]`, un indice de hachage entre `{...}` ou une liste d'arguments de sous-programme entre `(...)`, alors la partie à gauche de la flèche doit être respectivement une référence (symbolique ou en dur) de tableau, de hachage ou de sous-programme. Dans un contexte de *lvalue* (où l'on peut affecter une valeur), si la partie à gauche n'est pas une référence, elle doit être capable de contenir une référence, auquel cas cette référence sera *autovivifiée* pour vous. Pour en savoir plus à ce sujet (et pour quelques avertissements sur l'autovivification accidentelle), voir le chapitre 8, *Références*.

```
$aref->[42]              # un déréférencement de tableau
$href->{"corned beef"}   # un déréférencement de hachage
$sref->(1,2,3)           # un déréférencement de sous-programme
```

Sinon, c'est une sorte d'appel de méthode. La partie à droite doit être un nom de méthode (ou une simple variable scalaire contenant le nom de la méthode) et la partie à gauche doit s'évaluer soit comme un objet (une référence consacrée), soit comme un nom de classe (c'est-à-dire un nom de paquetage) :

```
$yogi = Ours->new("Yogi");    # un appel de méthode de classe
$yogi->pique($piquenique);    # un appel de méthode sur une instance
```

Le nom de méthode peut être qualifiée avec un nom de paquetage pour indiquer dans quelle classe commencer à chercher la méthode, ou avec le nom de paquetage spécial `SUPER::`, pour indiquer que la recherche doit commencer dans la classe parente. Voir le chapitre 12, *Objets*.

Auto-incrémentation et autodécrémentation

Les opérateurs `++` et `--` fonctionnent comme en C. C'est-à-dire que, placés avant une variable, ils l'incrémentent ou la décrémentent avant d'en renvoyer la valeur. Quand ils sont placés après, ils l'incrémentent ou la décrémentent après en avoir renvoyé la valeur. Par exemple, `$A++`[2] incrémente la valeur de la variable `$A` et en retourne la valeur *avant* de réaliser l'incrémentation. De même, `--$b{(/(\w+)/)[0]}` décrémente l'élément du hachage `%b` indexé par le premier « mot » de la variable de recherche par défaut (`$_`) et renvoie la valeur *après* la décrémentation.[3]

---

2. N.d.T. : Pour d'intéressantes variations sur `$A++`, vous pouvez consulter *http://paris.monguers.net/aplusplus.html*.

3. Bon d'accord, ce n'était pas très sympa. Nous voulions juste être sûrs que vous suiviez. Voici comment cette expression fonctionne. D'abord, la recherche de motif trouve le premier « mot » de `$_` en utilisant l'expression régulière `\w+`. Les parenthèses qui l'entourent permettent de retourner le mot comme valeur dans une liste à un seul élément car la détection se fait dans un contexte de liste. Le contexte de liste est fourni par l'opérateur de tranche de liste, `(...)[0]`, qui retourne le premier (et unique) élément de la liste. Cette valeur est utilisée comme clé pour le hachage et l'entrée du hachage (sa valeur) est décrémentée et retournée. En général, quand vous êtes confronté à une expression complexe, analysez-la de l'intérieur vers l'extérieur pour voir dans quel ordre les événements se produisent.

L'opérateur d'auto-incrémentation comporte en plus un peu de magie. Si vous incrémentez une variable qui est numérique, ou qui a été utilisée à un moment donné dans un contexte numérique, vous avez une incrémentation normale. Si au contraire la variable n'a été utilisée que dans des contextes de chaîne depuis qu'elle a été affectée, que sa valeur est non nulle et correspond au motif `/^[a-zA-Z]*[0-9]*$/`, l'incrémentation est faite sur la chaîne, en préservant chaque caractère dans son domaine, avec une retenue :

```
print ++($toto = '99');      # prints '100'
print ++($toto = 'a9');      # prints 'b0'
print ++($toto = 'Az');      # prints 'Ba'
print ++($toto = 'zz');      # prints 'aaa'
```

À l'heure où nous écrivons ces lignes, l'auto-incrémentation magique n'a pas été étendue aux lettres et chiffres Unicode, mais pourrait l'être dans le futur.

L'opérateur d'autodécrémentation n'est, lui, pas magique, et il n'est pas prévu que cela change.

Exponentiation

L'opérateur ** binaire est l'opérateur d'exponentiation. Notez qu'il lie encore plus fortement que le moins unaire ; c'est pourquoi `-2**4` vaut `-(2**4)`, et non `(-2)**4`. L'opérateur est implémenté grâce à la fonction *pow*(3) de C, qui fonctionne en interne avec des nombres en virgule flottante. Les calculs sont faits grâce à des logarithmes, ce qui signifie qu'il fonctionne avec des puissances fractionnaires, mais que vous obtiendrez parfois des résultats moins précis qu'avec une simple multiplication.

Opérateurs unaires idéographiques

La plupart des opérateurs portent simplement des noms (voir *Opérateurs unaires nommés et de test de fichier* plus loin dans ce chapitre), mais certains opérateurs sont jugés suffisament importants pour mériter leur propre représentation symbolique spéciale. Ces opérateurs semblent tous avoir un rapport avec la négation. Plaignez-vous auprès des mathématiciens.

Le ! unaire réalise une négation logique, c'est-à-dire un « non ». Voyez `not` pour une version de moindre précédence du non logique. La valeur d'un opérande nié est vraie (1) si l'opérande est faux (0 numérique, chaîne `"0"`, chaîne vide ou valeur indéfinie) et fausse (`""`) si l'opérande est vrai.

Le - unaire effectue une négation arithmétique si son opérande est numérique. Si l'opérande est un identificateur, une chaîne composée d'un signe moins concaténé à l'identificateur est renvoyée. Sinon, si la chaîne commence avec un plus ou un moins, une chaîne commençant avec le signe opposé est renvoyée. Un des effets de ces règles est que `-motsimple` est équivalent à `"-motsimple"`. C'est particulièrement utile aux programmeurs Tk.

Le ~ unaire réalise une négation sur les bits, c'est-à-dire un complément à 1. Par définition, ce n'est pas vraiment portable quand c'est limité par la taille du mot-machine de votre ordinateur. Par exemple, sur une machine 32 bits `~123` vaut `4294967172`, tandis

que sur une machine 64 bits, cela vaut `18446744073709551492`. Mais vous le saviez déjà.

Ce que vous ne saviez peut-être pas, c'est que si l'argument de ~ est une chaîne au lieu d'un nombre, une chaîne de même longueur est renvoyée avec tous ses bits complémentés. C'est une manière rapide de basculer un grand nombre de bits d'un coup, et de façon portable puisque cela ne dépend pas de la taille de votre mot-machine. Plus tard nous verrons aussi les opérateurs logiques sur les bits, qui ont eux des variantes pour les chaînes.

Le + n'a aucun effet sémantique, même sur les chaînes. Son utilité est surtout syntaxique, pour séparer le nom d'une fonction d'une expression entre parenthèses qui serait sinon interprétée comme la liste complète des arguments de la fonction. (Voir les exemples dans la section *Termes et opérateurs de listes (vers la gauche)*.) Si vous y réfléchissez d'une manière un peu détournée, le + inverse l'effet qu'ont les parenthèses de changer les opérateurs préfixes en fonctions.

Le \ crée une référence sur ce qui le suit. Utilisé sur une liste, il crée une liste de références. Voir la section *L'opérateur antislash* au chapitre 8 pour plus de détails. Attention à ne pas confondre ce comportement avec celui de l'antislash dans une chaîne, bien que les deux formes comportent la notion vaguement négative de protection de leurs arguments contre l'interprétation. Cette ressemblance n'est pas purement fortuite.

Opérateurs de liaison

Le `=~` binaire lie une expression scalaire à un opérateur de recherche de motif, de substitution ou de translittération (négligemment appelée traduction). Ces opérations chercheraient ou modifieraient sinon la chaîne contenue dans `$_`(la variable par défaut). La chaîne à lier est placée à gauche, tandis que l'opérateur lui-même est placé à droite. La valeur de retour indique le succès ou l'échec de l'opération effectuée à droite, puisque l'opérateur de lien ne fait rien par lui-même.

Si l'argument de droite est une expression plutôt qu'une recherche de motif, une substitution ou une translittération, il sera interprété comme une recherche de motif à l'exécution. C'est-à-dire que `$_ =~ $pat` est équivalent à `$_ =~ /$pat/`. C'est moins efficace qu'une recherche explicite, puisque le motif doit être vérifié et potentiellement recompilé à chaque fois que l'expression est évaluée. Vous pouvez éviter cette recompilation en précompilant le motif original avec l'opérateur `qr//` de citation d'expression régulière (« quote regex » en anglais).

Le `!~` binaire est identique à `=~`sauf que la valeur de retour est inversée logiquement. Les expressions suivantes sont fonctionnellement équivalentes :

```
$chaine !~ /pattern/
not $chaine =~ /pattern/
```

Nous avons dit que la valeur de retour indique le succès, mais il existe plusieurs sortes de succès. La substitution renvoie le nombre de substitutions réussies, comme la translittération. (En fait, la translittération sert souvent à compter les caractères.) Puisque n'importe quel résultat non nul est vrai, tout fonctionne. La forme de valeur vraie la plus spectaculaire est une valeur de liste : en contexte de liste, les recherches de motif peuvent renvoyer les sous-chaînes capturées par les parenthèses dans le motif. Conformément aux règles de l'affectation de liste, cette affectation retournera vrai si quelque

chose a été détecté et affecté, et faux dans le cas contraire. C'est pourquoi vous voyez parfois des choses comme :

```
if ( ($c,$v) = $chaine =~ m/(\w+)=(\w*)/ ) {
    print "CLE $c VALEUR $v\n";
}
```

Décomposons tout cela. Le `=~` a la précédence sur `=`, il se produit donc en premier. Le `=~`lie `$chaine` à la recherche de motif à droite. Celle-ci recherche ce qui ressemble à *`CLE=VALEUR`* dans votre chaîne. Il se trouve en contexte de liste, car du côté droit d'une affectation de liste. Si le motif est trouvé, il renvoie une liste à affecter à `$c` et `$v`. L'affectation de liste elle-même se trouve dans un contexte scalaire et renvoie donc `2`, le nombre de valeurs à droite de l'affectation. Et il se trouve que `2` est vrai, puisque notre contexte scalaire est aussi un contexte booléen. Quand la recherche échoue, aucune valeur n'est affectée, ce qui renvoie 0, qui est faux.

Voir le chapitre 5, *Recherche de motif*, pour en savoir plus sur le fonctionnement des motifs.

Opérateurs multiplicatifs

Perl fournit des opérateurs proches de ceux de C : * (multiplication), `/` (division) et `%` (modulo). * et `/` fonctionnent exactement comme vous vous y attendez, en multipliant ou divisant leurs deux opérandes. La division se fait en virgule flottante, à moins que vous n'ayez spécifié le pragma `integer`.

L'opérateur `%` convertit ses opérandes en entiers avant de calculer le reste de la division entière. (Néanmoins cette division est faite en virgule flottante donc vos opérandes peuvent faire 15 chiffres sur la plupart des machines 32 bits.) Supposons que nos deux opérandes s'appellent `$a` et `$b`. Si `$b` est positif, alors le résultat de `$a % $b` est `$a` moins le plus grand multiple de `$b` inférieur à `$a` (ce qui veut dire que le résultat sera toujours dans l'intervalle `0 .. $b-1`). Si `$b` est négatif, alors le résultat de `$a % $b` est `$a` moins le plus petit multiple de `$b` supérieur à `$a` (ce qui veut dire que le résultat sera toujours dans l'intervalle `$b+1 .. 0`).

Quand on est à portée de `use integer`, `%` vous donne directement accès à l'opérateur modulo tel qu'il est implémenté par votre compilateur C. Cet opérateur n'est pas bien défini pour les opérandes négatifs, mais s'exécutera plus rapidement.

Le `x` est l'opérateur de répétition. En réalité, il s'agit de deux opérateurs. En contexte scalaire, il retourne une chaîne concaténée composée de l'opérande de gauche répété le nombre de fois spécifié par l'opérande de droite. (Pour assurer la compatibilité avec les versions précédentes, il le fait également en contexte de liste si l'argument de gauche n'est pas entre parenthèses.)

```
print '-' x 80;                          # imprime une rangée de tirets
print "\t" x ($tab/8), ' ' x ($tab%8);   # tabule à la colonne $tab
```

En contexte de liste, si l'opérande de gauche est une liste entre parenthèses, `x` fonctionne comme un réplicateur de liste plutôt que comme un réplicateur de chaîne. Cela peut servir à initialiser à la même valeur tous les éléments d'un tableau de longueur indéterminée.

```
@ones = (1) x 80;           # une liste de 80 1
@ones = (5) x @ones;        # met tous les éléments à 5
```

De la même manière, vous vous pouvez également utiliser `x` pour initialiser des tranches de tableaux et de hachages.

```
@cles = qw(du Perl aux cochons);
@hash{@cles} = ("") x @cles;
```

Si cela vous laisse perplexe, remarquez que `@cles` est utilisé à la fois comme une liste à gauche de l'affectation et comme une valeur scalaire (qui renvoie la longueur du tableau) à droite. L'exemple précédent a le même effet sur `%hash` que :

```
$hash{du}      = "";
$hash{Perl}    = "";
$hash{aux}     = "";
$hash{cochons} = "";
```

Opérateurs additifs

Étonnament, Perl dispose des opérateurs usuels `+` (addition) et `-` (soustraction). Ces deux opérateurs convertissent leurs arguments de chaîne en valeurs numériques si nécessaire, et renvoient un résultat numérique.

Perl fournit également l'opérateur `.` qui réalise la concaténation de chaînes. Par exemple :

```
$presque = "Fred" . "Pierrafeu";    # renvoie FredPierrafeu
```

Remarquez que Perl n'ajoute pas d'espace entre les chaînes concaténées. Si vous voulez l'espace ou si vous avez plus de deux chaînes à concaténer, vous pouvez utiliser l'opérateur `join`, décrit au chapitre 29, *Fonctions*. Cependant, la plupart du temps on concatène implicitement dans une chaîne entre doubles apostrophes :

```
$nomcomplet = "$prenom $nom";
```

Opérateurs de décalage

Les opérateurs de décalage de bits (`<<` et `>>`) renvoient la valeur de l'argument de gauche décalé à gauche (`<<`) ou à droite (`>>`) du nombre de bits spécifié par l'argument de droite. Les arguments doivent être des entiers. Par exemple :

```
1 << 4;     # renvoie 16
32 >> 4;    # renvoie 2
```

Faites attention cependant, car les résultats sur de grands nombres (ou sur des nombres négatifs) peuvent dépendre du nombre de bits utilisés par votre machine pour représenter les entiers.

Opérateurs unaires nommés et de test de fichier

Certaines des « fonctions » décrites au chapitre 29 sont en fait des opérateurs unaires. Le *tableau 3-2* liste tous les opérateurs unaires nommés.

Les opérateurs unaires nommés ont une précédence supérieure à celle de certains opérateurs binaires. Par exemple :

```
sleep 4 | 3;
```

Tableau 3-2. Opérateurs unaires nommés

-X (tests de fichiers)	gethostbyname	localtime	return
alarm	getnetbyname	lock	rmdir
caller	getpgrp	log	scalar
chdir	getprotobyname	lstat	sin
chroot	glob	my	sleep
cos	gmtime	oct	sqrt
defined	goto	ord	srand
delete	hex	quotemeta	stat
do	int	rand	uc
eval	lc	readlink	ucfirst
exists	lcfirst	ref	umask
exit	length	require	undef

ne dort pas pendant 7 secondes ; il dort pendant 4 secondes puis prend la valeur de retour de `sleep` (typiquement zéro) et la combine par OU avec 3, comme si l'expression avait des parenthèses comme suit :

```
(sleep 4) | 3;
```

À comparer avec :

```
print 4 | 3;
```

qui prend `effectivement` la valeur de 4 combiné par OU avec 3 avant de l'afficher (7 dans ce cas), comme s'il était écrit :

```
print (4 | 3);
```

En effet, `print` est un opérateur de liste, et non un simple opérateur unaire. Une fois que vous saurez quels opérateurs sont des opérateurs de liste, vous n'aurez plus de problème pour distinguer les opérateurs de liste des opérateurs unaires. En cas de doute, vous pouvez toujours utiliser des parenthèses pour transformer un opérateur unaire en fonction. Souvenez-vous, si ça ressemble à une fonction, c'est une fonction.

Une autre caractéristique curieuse des opérateurs unaires nommés est que beaucoup d'entre eux prennent `$_` comme argument par défaut, si vous ne leur en fournissez pas d'autre. Cependant, si vous omettez l'argument et que ce qui suit ressemble au début d'un argument, Perl va se tromper, car il attend un terme. Quand le caractère suivant du programme est l'un de ceux listés au *tableau 3-3*, le *tokeniseur* de Perl renvoie différents types de token selon qu'il attend un terme ou un opérateur.

Une erreur typique est donc :

```
next if length < 80;
```

où pour l'analyseur le `<` ressemble au début du symbole `<>` (un terme) au lieu du « inférieur à » (un opérateur) auquel vous pensiez. Il n'existe pas vraiment de moyen d'arranger les choses tout en gardant Perl pathologiquement éclectique. Si vous êtes paresseux au point de ne pouvoir vous résoudre à taper les deux caractères `$_`, l'une de ces instructions devrait faire l'affaire :

Tableau 3-3. Caractères ambigus

Caractère	Opérateur	Terme
`+`	Addition	Plus unaire
`-`	Soustraction	Moins unaire
`*`	Multiplication	`*typeglob`
`/`	Division	`/motif/`
`<`	Inférieur à, décalage à gauche	`<HANDLE>, <<END`
`.`	Concaténation	`.3333`
`?`	`?:`	`?motif?`
`%`	Modulo	`%assoc`
`&`	`&, &&`	`&sousprogramme`

```
next if length() < 80;
next if (length) < 80;
next if 80 > length;
next unless length >= 80;
```

Quand un terme est attendu, un signe moins suivi d'une lettre seule sera toujours interprété comme un `test de fichier`. Un opérateur de test de fichier est un opérateur unaire qui prend commme argument un nom ou un handle de fichier, et teste le fichier associé pour savoir si une certaine propriété est vraie à son sujet. Si l'argument est omis, il teste `$_`, sauf pour `-t`, qui teste `STDIN`. Sauf si c'est indiqué autrement dans la documentation, il retourne `1` pour vrai et `""` pour faux, ou la valeur indéfinie si le fichier n'existe pas ou n'est pas accessible. Les opérateurs de test de fichier actuellement implémentés sont listés au *tableau 3-4*.

Tableau 3-4. Opérateurs de test de fichier

Opérateur	Signification
`-r`	Fichier lisible par l'UID/GID effectif.
`-w`	Fichier en écriture pour l'UID/GID effectif.
`-x`	Fichier exécutable par l'UID/GID effectif.
`-o`	Fichier possédé par l'UID/GID effectif.
`-R`	Fichier lisible par l'UID/GID réel.
`-W`	Fichier en écriture pour l'UID/GID réel.
`-X`	Fichier exécutable par l'UID/GID réel.
`-O`	Fichier possédé par l'UID/GID réel.
`-e`	Le fichier existe.
`-z`	Fichier de taille nulle.
`-s`	Fichier de taille non nulle (renvoie la taille)
`-f`	Fichier simple.
`-d`	Le fichier est un répertoire.
`-l`	Le fichier est un lien symbolique.

Tableau 3-4. Opérateurs de test de fichier (suite)

Opérateur	Signification
`-p`	Le fichier est un tube nommé (FIFO).
`-S`	Le fichier est une socket.
`-b`	Fichier spécial de type bloc
`-c`	Fichier spécial de type caractère.
`-t`	Handle de fichier ouvert sur un tty.
`-u`	Fichier avec bit setuid.
`-g`	Fichier avec bit setgid.
`-k`	Fichier avec sticky bit.
`-T`	Fichier texte.
`-B`	Fichier binaire (le contraire de `-T`).
`-M`	Âge du fichier (au démarrage) en jours depuis sa modification.
`-A`	Âge du fichier (au démarrage) en jours depuis le dernier accès.
`-C`	Âge du fichier (au démarrage) en jours depuis le changement d'inode.

Remarquez que `-s/a/b/` ne fait pas une substitution négative. `-exp($foo)` fonctionne cependant comme prévu ; seules les lettres isolées qui suivent un signe moins sont interprétées comme des tests de fichier.

L'interprétation des opérateurs de test de permissions `-r`, `-R`, `-w`, `-W`, `-x` et `-X` est fondée uniquement sur le mode du fichier et les ID d'utilisateur et de groupe de l'utilisateur. Il peut y avoir d'autres raisons pour lesquelles vous ne pouvez effectivement pas lire, écrire ou exécuter le fichier, comme les listes de contrôle d'accès d'AFS (*Andrew File System*).[4] Notez également que pour le super utilisateur `-r`, `-R`, `-w` et `-W` renvoient toujours 1, et que `-x` et `-X` renvoient 1 si l'un des bits d'exécution est à 1 dans le mode. C'est pourquoi les scripts lancés par le super utilisateur peuvent nécessiter un `stat` afin de connaître le véritable mode du fichier, ou bien remplacer temporairement l'UID par autre chose.

Les autres opérateurs de test de fichier se moquent de savoir qui vous êtes. N'importe qui peut utiliser le test pour les fichiers « simples » :

```
while (<>) {
    chomp;
    next unless -f $_;      # ignore les fichiers "spéciaux"
    ...
}
```

Les options `-T` et `-B` fonctionnent comme suit. Le premier bloc (plus ou moins) du fichier est examiné à la recherche de caractères inhabituels comme des code de contrôle ou des octets donc le bit de poids fort est à 1 (et qui n'ont pas l'air d'être de l'UTF-8). Si on trouve plus d'un tiers d'octets inhabituels, c'est un fichier binaire ; sinon c'est un fi-

4. Vous pouvez cependant ignorer la sémantique intégrée avec le pragma `use filetest`. Voir le chapitre 31, *Modules de pragmas*.

chier texte. De même, tout fichier dont le premier bloc contient le caractère ASCII NUL (`\0`) est considéré comme binaire. Si `-T` ou `-B` est utilisé sur un handle de fichier, le tampon d'entrée standard (standard I/O ou « stdio ») en cours est examiné, au lieu du premier bloc du fichier. `-T` et `-B` renvoient vrai sur un fichier vide, ou sur un fichier à EOF (*end-of-file* : fin de fichier) quand on teste un handle. Comme Perl doit lire le fichier pour faire le test `-T`, vous devrez éviter de vous en servir sur des fichiers qui risquent de bloquer ou de vous causer des ennuis. C'est pourquoi la plupart du temps, vous devrez tester avec un `-f` d'abord, comme dans :

```
next unless -f $fichier && -T $fichier;
```

Si l'on donne à l'un des tests de fichier (ou à l'un des opérateurs `stat` ou `lstat`) le handle spécial constitué d'un souligné unique, c'est la structure *stat* du dernier test de fichier (ou de l'opérateur `stat`) qui est utilisée, économisant ainsi un appel système. (Cela ne marche pas avec `-t`, et il faudra vous souvenir que `lstat` et `-l` laissent dans la structure *stat* les valeurs pour le lien symbolique et non pour le fichier réel. De même, `-l _` sera toujours faux après un `stat` normal.)

Voici quelques exemples :

```
print "Fait l'affaire.\n"    if -r $a || -w _ || -x _;

stat($fichier);
print "En lecture\n"         if -r _;
print "En écriture\n"        if -w _;
print "Exécutable\n"         if -x _;
print "Setuid\n"             if -u _;
print "Setgid\n"             if -g _;
print "Sticky\n"             if -k _;
print "Texte\n"              if -T _;
print "Binaire\n"            if -B _;
```

Les âges des fichiers donnés par `-M`, `-A` et `-C` sont en jours (avec partie fractionnaire) depuis le moment où le script a commencé à tourner. Cette date est stockée dans la variable spéciale `$^T` (`$BASETIME`). Donc, si le fichier a été modifié depuis que le script a démarré, vous obtiendrez une durée négative. Remarquez que comme la plupart des durées (86 399 sur 86 400 en moyenne) sont fractionnaires, il est généralement illusoire de tester l'égalité avec un entier sans passer par la fonction `int`. Exemples :

```
next unless -M $fichier > .5;       # fichier vieux de plus de 12 heures
&nouveaufichier if -M $fichier < 0; # fichier plus récent que le processus
&avertissemnt if int(-A) == 90;     # fichier ($_) accédé il y a 90 jours
                                    # aujourd'hui
```

Pour remettre à la date courante la date de démarrage du script, écrivez :

```
$^T = time;
```

Opérateurs relationnels

Perl possède deux classes d'opérateurs relationnels. L'une d'elles opère sur les valeurs numériques et l'autre sur les valeurs de chaîne, comme indiqué au *tableau 3-5*.

Tableau 3-5. Opérateurs relationnels

Numérique	Chaîne	Signification
`>`	`gt`	Supérieur à.
`>=`	`ge`	Supérieur ou égal à.
`<`	`lt`	Inférieur à.
`<=`	`le`	Inférieur ou égal à.

Ces opérateurs renvoient `1` pour vrai et `""` pour faux. Notez que les opérateurs relationnels sont non-associatifs, ce qui signifie que `$a < $b < $c` provoquera une erreur de syntaxe.

En l'absence de déclaration de locales, les comparaisons de chaînes s'appuient sur l'ordre lexicographique ASCII/Unicode et, contrairement à certains autres langages informatiques, les espaces finaux comptent pour la comparaison. Avec une déclaration de locale, l'ordre lexicographique spécifié par la locale est utilisé. (Les mécanismes de tri fondés sur les locales peuvent plus ou moins bien interagir avec les mécanismes Unicode actuellement en cours de développement.)

Opérateurs d'égalité

Les opérateurs d'égalité listés au *tableau 3-6* ressemblent beaucoup aux opérateurs relationnels.

Tableau 3-6. Opérateurs d'égalité

Numérique	Chaîne	Signification
`==`	`eq`	Égal à.
`!=`	`ne`	Différent de.
`<=>`	`cmp`	Comparaison, avec résultat signé.

Les opérateurs égal et différent renvoient `1` pour vrai et `""` pour faux (exactement comme les opérateurs relationnels). Les opérateurs `<=>` et `cmp` renvoient `-1` si l'opérande de gauche est inférieur à celui de droite, `0` s'il sont égaux et `+1` si l'opérande de gauche est supérieur à celui de droite. Bien que ces opérateurs semblent très proches des opérateurs relationnels, ils ont un niveau de précédence plus faible, et la syntaxe de `$a < $b <=> $c < $d` est donc valide.

Pour des raisons évidentes pour quiconque a vu *Star Wars*, l'opérateur `<=>` est aussi connu sous le nom d'opérateur « vaisseau spatial » (*spaceship operator*).

Opérateurs sur les bits

Comme C, Perl dispose des opérateurs ET, OU et OU-exclusif sur les bits : `&`, `|` et `^`. Vous avez bien sûr remarqué, après votre étude attentive du tableau au début de ce chapitre, que le ET sur les bits possède une précédence supérieure aux autres ; nous avons triché pour tous les inclure dans cette discussion.

Ces opérateurs fonctionnent différemment sur les valeurs numériques et sur les chaînes. (C'est l'un des rares cas où Perl fait une différence.) Si l'un des opérandes est un nombre (ou a été utilisé comme un nombre), les deux opérandes sont convertis en entiers, puis l'opération sur les bits est effectuée. Ces entiers font au moins 32 bits de long, mais peuvent faire 64 bits sur certaines machines. L'important est qu'il existe une limite arbitraire imposée par l'architecture de la machine.

Si les deux opérandes sont des chaînes (et n'ont pas été utilisés comme des nombres depuis leur dernière mise à jour), les opérateurs font les opérations sur les bits sur les bits correspondants des deux chaînes. Dans ce cas, il n'y a aucune limite arbitraire, puisque les chaînes ne sont pas limitées en taille. Si l'une des chaînes est plus longue que l'autre, on considère que la plus courte dispose d'un nombre suffisant de bits à 0 pour compléter la différence.

Par exemple, si vous faites un ET entre deux chaînes :

```
"123.45" & "234.56"
```

vous obtenez une autre chaîne :

```
"020.44"
```

Mais si vous faites un ET sur une chaîne et un nombre :

```
"123.45" & 234.56
```

La chaîne est d'abord convertie en nombre, ce qui donne :

```
123.45 & 234.56
```

Les nombres sont ensuite convertis en entiers :

```
123 & 234
```

ce qui donne 106. Remarquez que toutes les chaînes de bits sont vraies (à moins de donner la chaîne « 0 »). Cela signifie que si vous voulez vérifier si l'un des octets résultant est non nul, vous ne devrez pas écrire ceci :

```
if ( "fred" & "\1\2\3\4" ) { ... }
```

mais cela :

```
if ( ("fred" & "\1\2\3\4") =~ /[^\0]/ ) { ... }
```

Opérateurs logique de type C (à court-circuit)

Comme C, Perl propose également les opérateurs && (ET logique) et || (OU logique). Ils sont évalués de gauche à droite (&& ayant une précédence légérement supérieure à celle de ||) pendant le test de vérité de l'instruction. Ces opérateurs sont appelés opérateurs court-circuit car ils déterminent la vérité de l'instruction en évaluant le plus petit nombre d'opérandes possible. Par exemple, si l'opérande à gauche d'un opérateur && est faux, l'opérande de droite ne sera jamais évalué car le résultat de l'opérateur sera faux quelle que soit la valeur de l'opérande de droite.

Exemple	Nom	Résultat
`$a && $b`	Et	`$a` si `$a` est faux, sinon `$b`.
`$a || $b`	Ou	`$a` si `$a` est vrai, sinon `$b`.

Non seulement de tels courts-circuits font gagner du temps, mais ils sont fréquemment utilisés pour contrôler le flux de l'évaluation. Par exemple, un idiome courant en Perl est :

```
open(FILE, "monfichier") || die "Impossible d'ouvrir monfichier: $!\n";
```

Dans ce cas, Perl évalue d'abord la fonction `open`. Si sa valeur est vraie (car *monfichier* a été ouvert avec succès), l'exécution de la fonction `die` n'est pas nécessaire et est donc omise. Vous pouvez littéralement lire « Ouvre mon fichier ou meurs ! ».

Les opérateurs `&&` et `||` diffèrent de ceux de C en ceci qu'au lieu de retourner 0 ou 1, ils renvoient la dernière valeur évaluée. Dans le cas de `||`, cela a le délicieux effet de vous permettre de choisir la première valeur vraie d'une série. Une manière raisonnablement portable de trouver le répertoire personnel d'un utilisateur pourrait donc être :

```
$home = $ENV{HOME}
     || $ENV{LOGDIR}
     || (getpwuid($<))[7]
     || die "Vous êtes SDF !\n";
```

D'un autre côté, comme l'argument de gauche toujours évalué en contexte scalaire, vous ne pouvez pas vous servir de `||` afin choisir entre deux aggrégats pour une affectation :

```
@a = @b || @c;            # Ceci ne fait pas ce que vous voulez
@a = scalar(@b) || @c;    # car voici ce que cela veut dire.
@a = @b ? @b : @c;        # En revanche cela marche bien.
```

Perl fournit également des opérateurs `and` et `or` de moindre précédence, que certains trouvent plus lisibles et qui ne vous forcent pas à utiliser des parenthèses sur les opérateurs de liste. Ils sont aussi à court-circuit. Voir le *tableau 1-1* pour la liste complète.

Opérateur d'intervalle

L'opérateur d'intervalle `..` comprend en fait deux opérateurs différents selon le contexte.

L'opérateur est bi-stable, comme un flip-flop électronique, et émule l'opérateur d'intervalle de ligne (virgule) de *sed*, *awk* et d'autres éditeurs. Chaque opérateur `..` scalaire mémorise son propre état booléen. Il est faux tant que son opérande de gauche est faux. Une fois que l'opérande de gauche est vrai, l'opérateur reste vrai jusqu'à ce que l'opérande de droite soit vrai, après quoi l'opérateur d'intervalle redevient faux. L'opérateur ne devient pas faux jusqu'à sa prochaine évaluation. Il peut tester l'opérande de droite et devenir faux pendant la même évaluation que celle sur laquelle il est devenu vrai (comme l'opérateur de *awk*), mais renvoie vrai au moins une fois. Si vous ne voulez pas qu'il teste l'opérande de droite avant la prochaine évaluation (comme l'opérateur de *sed*) utilisez simplement trois points (`...`) au lieu de deux. Pour les deux opérateurs `..` et `...`, l'opérande de droite n'est pas évalué tant que l'opérateur se trouve dans l'état faux, et l'opérande de gauche n'est pas évalué tant que l'opérateur est dans l'état vrai.

La valeur renvoyée est soit la chaîne vide pour faux soit un numéro de séquence (commençant à `1`) pour vrai. Le numéro de séquence est remis à zéro pour chaque opérateur d'intervalle rencontré. La chaîne « `E0` » est accolée à la fin du numéro de séquence final dans un intervalle, ce qui n'affecte pas sa valeur numérique, mais vous donne quelque

chose à chercher si vous voulez exclure l'élément final. Vous pouvez exclure le point de départ en attendant que le numéro de séquence soit plus grand que 1. Si l'un des opérandes de `..` est un littéral numérique, cet opérande est implicitement comparé à la variable `$.`, qui contient le numéro de ligne courant de votre fichier d'entrée. Exemples :

```
if (101 .. 200) { print; }  # imprime la deuxième centaine de lignes
next line if (1 .. /^$/);   # passe les entêtes d'un message
s/^/> / if (/^$/ .. eof()); # cite le corps d'un message
```

En contexte de liste, `..` renvoie une liste de valeurs comptées à partir de la valeur de gauche jusqu'à la valeur de droite (par incréments de un). Cela sert à écrire des boucles `for (1..10)` et pour faire des tranches de tableaux :

```
for (101 .. 200) { print; }          # affiche 101102...199200
@foo = @foo[0 .. $#foo];             # un no-op coûteux
@foo = @foo[ -5 .. -1];              # la tranche des 5 derniers éléments
```

Si la valeur de gauche est supérieure à celle de droite, une liste vide est renvoyée. (Pour construire une liste en ordre inverse, voir l'opérateur `reverse`.)

Si ses opérandes sont des chaînes, l'opérateur d'intervalle emploie l'algorithme d'auto-incrémentation magique vu précédemment.[5] Vous pouvez donc écrire :

```
@alphabet = ('A' .. 'Z');
```

pour obtenir toutes les lettres de l'alphabet (latin), ou :

```
$hexdigit = (0 .. 9, 'a' .. 'f')[$nombre & 15];
```

pour obtenir un chiffre hexadécimal, ou :

```
@z2 = ('01' .. '31');  print $z2[$jour];
```

pour obtenir des dates avec un zéro en tête. Vous pouvez également écrire :

```
@combis = ('aa' .. 'zz');
```

pour obtenir toutes les combinaisons de deux lettres minuscules. Attention cependant à quelque chose comme :

```
@grossescombis = ('aaaaaa' .. 'zzzzzz');
```

car cela va nécessiter beaucoup de mémoire. Pour être précis, il faudra de l'espace pour stocker 308 915 776 scalaires. Espérons que vous avez une grosse partition de swap. Vous devriez vous intéresser à une approche itérative.

Opérateur conditionnel

L'opérateur `?:` est le seul opérateur ternaire, comme en C. On l'appelle souvent l'opérateur conditionnel, car il fonctionne comme un *if-then-else*, excepté qu'il peut être inclus sans problème dans d'autres expressions et fonctions, car c'est une expression et non une instruction. En tant qu'opérateur ternaire, ses deux éléments séparent trois expressions :

```
COND ? EXPR_SI_VRAI : EXPR_SI_FAUX
```

5. Si la valeur finale spécifiée n'est pas dans la séquence que l'incrémentation magique produirait, la séquence continue jusqu'à ce que le prochain élément soit plus long que la valeur finale.

Si la condition *COND* est vrai, seule l'expression *EXPR_SI_VRAI* est évaluée, et la valeur de cette expression devient la valeur de toute l'expression. Sinon, seule l'expression *EXPR_SI_FAUX* est évaluée, et sa valeur devient celle de toute l'expression.

Le contexte scalaire ou de liste se propage vers le deuxième ou le troisième argument, selon celui qui est sélectionné. (Le premier argument est toujours en contexte scalaire, puisque c'est une condition.)

```
$a = $ok ? $b : $c;  # donne un scalaire
@a = $ok ? @b : @c;  # donne un tableau
$a = $ok ? @b : @c;  # donne le nombre d'éléments d'un tableau
```

Vous verrez souvent l'opérateur conditionnel inclus dans des listes de valeurs à formater avec `printf`, car personne n'a envie de dupliquer une instruction complète juste pour basculer entre deux valeurs proches.

```
printf "J'ai %d chameau%s.\n",
            $n,       $n <= 1 ? "" : "x";
```

La précédence de `?:` est opportunément plus grande que celle de la virgule, mais inférieure à celle de la plupart des opérateurs que vous utiliserez à l'intérieur (comme `==` dans cet exemple) ; vous n'aurez donc généralement pas à utiliser de parenthèses. Mais vous pouvez ajouter des parenthèses pour clarifier, si vous voulez. Pour les opérateurs conditionnels emboîtés à l'intérieur de la partie *EXPR_SI_VRAI* d'autres opérateurs conditionnels, nous vous suggérons de de faire des sauts de ligne et d'indenter comme s'il s'agisait d'instruction `if` ordinaires :

```
$bissextile =
     $annee % 4 == 0
          ? $annee % 100 == 0
              ? $annee % 400 == 0
                  ? 1
                  : 0
              : 1
          : 0;
```

Pour les conditions imbriquées dans des parties *EXPR_SI_FAUX* d'opérateurs conditionnels précédents, vous pouvez faire quelque chose d'équivalent :

```
$bissextile =
     $annee % 4
          ? 0
          : $annee % 100
            ? 1
            : $annee % 400
              ? 0
              : 1;
```

mais il est habituellement préférable d'aligner verticalement toutes les *COND* et les *EXPR_SI_VRAI* :

```
$bissextile =
     $annee %   4 ? 0 :
     $annee % 100 ? 1 :
     $annee % 400 ? 0 : 1;
```

Même des structures assez encombrées peuvent s'éclaircir en alignant les points d'interrogation et les deux-points :

```
printf "Oui, j'aime mon livre du %s!\n",
    $i18n eq "anglais"  ? "camel"           :
    $i18n eq "allemand" ? "Kamel"           :
    $i18n eq "japonais" ? "\x{99F1}\x{99DD}" :
                          "chameau"
```

Vous pouvez affecter à l'opérateur conditionnel[6] si le deuxième et le troisième arguments sont tous les deux des *lvalues*légales (c'est-à-dire qu'on peut leur affecter une valeur), et que tous deux sont soit des scalaires, soit des listes (sinon Perl ne saura pas quel contexte fournir à la partie droite de l'affectation) :

```
($a_ou_b ? $a : $b) = $c;  # donne la valeur de $c soit à $a, soit à $b
```

Souvenez-vous que l'opérateur conditionnel lie encore plus fort que les multiples opérateurs d'affectation. C'est habituellement ce que vous voulez (voir l'exemple `$bissextile` plus haut, par exemple), mais vous ne pourrez pas obtenir l'effet inverse sans parenthèses. L'utilisation d'affectations dans un opérateur conditionnel vous attirera des ennuis, et vous risquez même de ne pas avoir d'erreur à l'analyse car l'opérateur conditionnel peut être analysé comme une *lvalue*. Par exemple, vous pourriez écrire ceci :

```
$a % 2 ? $a += 10 : $a += 2        # FAUX
```

Mais ce serait analysé comme cela :

```
(($a % 2) ? ($a += 10) : $a) += 2
```

Opérateurs d'affectation

Perl reconnaît les opérateurs d'affectation de C, et fournit également les siens. Il y en a un certain nombre :

```
=    **=    +=    *=    &=    <<=    &&=
            -=    /=    |=    >>=    ||=
            .=    %=    ^=
                  x=
```

Chaque opérateur nécessite une *lvalue* cible (typiquement une variable ou un élément de tableau) à sa gauche et une expression à sa droite. Pour l'opérateur d'affectation simple :

```
CIBLE = EXPR
```

La valeur de *`EXPR`* est stockée dans la variable ou à l'endroit désigné par *`CIBLE`*. Pour les autres opérateurs, Perl évalue l'expression :

```
CIBLE OP= EXPR
```

comme s'il était écrit :

```
CIBLE = CIBLE OP EXPR
```

6. Cela ne garantit pas une amélioration de la lisibilité de votre programme. Mais ça peut servir à construire des participations sympas pour un concours de Perl assombri (*Obfuscated Perl Contest*).

C'est un bon moyen mnémotechnique, mais il est trompeur, et de deux manières. D'abord les opérateurs d'affectation sont analysés au niveau de précédence des affectations ordinaires, quelle que soit la précédence que l'opérateur *OP* aurait eu par lui-même. Ensuite, *CIBLE* n'est évalué qu'une seule fois. Habituellement cela n'a aucune importance, sauf en cas d'effets secondaires, comme pour une auto-incrémentation.

```
$var[$a++] += $valeur;              # $a is incrementé une fois
$var[$a++] = $var[$a++] + $valeur;  # $a is incrementé deux fois
```

À la différence de C, l'opérateur d'affectation produit une *lvalue* valide. Modifier une affectation revient à faire l'affectation puis à modifier la variable à laquelle on vient d'affecter une valeur. Cela peut servir à modifier une copie de quelque chose, comme ceci :

```
($tmp = $global) += $constante;
```

qui est équivalent à :

```
$tmp = $global + $constante;
```

De même :

```
($a += 2) *= 3;
```

est équivalent à :

```
$a += 2;
$a *= 3;
```

Ce n'est pas tellement utile, mais voici un idiome fréquent :

```
($nouveau = $ancien) =~ s/toto/titi/g;
```

Dans tous les cas, la valeur de l'affectation est la nouvelle valeur de la variable. Comme les opérateurs d'affectations sont associatifs de droite à gauche, cela peut servir à affecter la même valeur à plusieurs variables, comme dans :

```
$a = $b = $c = 0;
```

qui affecte `0` à `$c` et le résultat de cette opération (toujours `0`) à `$b`, puis le résultat de (*toujours* `0`) à `$a`.

Les affectations de listes ne se font qu'avec l'opérateur d'affectation simple, `=`. Dans un contexte de liste, une affectation de liste retourne la liste des nouvelles valeurs, tout comme l'affectation scalaire. En contexte scalaire, une affectation de liste renvoie le nombre de valeurs qui étaient disponibles à droite de l'affectation, comme indiqué au chapitre 2, *Composants de Perl*. Cela sert pour tester les fontions qui renvoient une liste vide quand elles échouent (ou finissent par échouer), comme dans :

```
while (($cle, $valeur) = each %gloss) { ... }

next unless ($dev, $ino, $mode) = stat $fichier;
```

Opérateurs virgule

Le « `,` » binaire est l'opérateur virgule. En contexte scalaire, il évalue son argument de gauche en contexte vide, jette le résultat, puis évalue son argument de droite et retourne cette valeur. Exactement comme l'opérateur virgule de C. Par exemple :

```
$a = (1, 3);
```

affecte `3` à `$a`. Attention à ne pas confondre son utilisation en contexte scalaire avec son utilisation en contexte de liste. Dans un contexte de liste, une virgule est juste le séparateur des arguments de la liste et il insère ses deux arguments dans la *`LISTE`*. Il ne jette aucune valeur.

Par exemple, si vous modifiez l'exemple précédent en :

```
@a = (1, 3);
```

vous construisez une liste de deux éléments, tandis que :

```
atan2(1, 3);
```

appelle la fonction `atan2` avec deux arguments.

La plupart du temps, le digramme `=>` est juste un synonyme pour l'opérateur virgule. Il sert à documenter les arguments appariés. Il force également l'interprétation comme chaîne de tout identificateur placé à sa gauche.

Opérateurs de liste (vers la droite)

La partie droite d'un opérateur de liste commande tous les arguments de l'opérateur de liste, qui sont séparés par des virgules ; donc la précédence d'un opérateur de liste est plus faible que celle de la virgule, si vous regardez vers la droite. Une fois qu'un opérateur de liste commence à avaler des arguments séparés par des virgules, les seules choses qui l'arrêtent sont les tokens qui terminent l'expression toute entière (comme les points-virgules ou les modificateurs d'instructions) ou les tokens qui terminent la sous-expression en cours (comme les parenthèses ou les crochets fermants) ou les opérateurs logiques de faible précédence dont nous aller parler tout de suite.

And, or, not et xor logiques

Perl fournit les opérateurs `and`, `or`, et `not` comme alternatives de moindre précédence à `&&`, `||`, et `!`. Le comportement de ces opérateurs est identique — en particulier, `and` et `or` court-circuitent comme leurs alter-egos, ce qui leur donne leur utilité non seulement pour les expressions logiques, mais aussi pour le contrôle du flux d'évaluation.

Comme la précédence de ces opérateurs est beaucoup plus basse que celle des opérateurs empruntés à C, vous pouvez les utiliser en toute sécurité après un opérateur de liste, sans qu'il soit besoin de parenthèses.

```
unlink "alpha", "beta", "gamma"
        or enrage(), next LIGNE;
```

Avec les opérateurs issus de C, vous auriez dû l'écrire comme ceci :

```
unlink("alpha", "beta", "gamma")
        || (enrage(), next LIGNE);
```

Mais vous ne pouvez pas simplement remplacer toutes les occurences de `||` par `or`. Supposons que vous changiez ceci :

```
$xyz = $x || $y || $z;
```

en cela :

```
$xyz = $x or $y or $z;    # FAUX
```

Cela ne ferait pas du tout la même chose ! La précédence de l'affectation est supérieure à celle de `or` mais plus basse que celle de `||`, donc on affecterait toujours `$x` à `$xyz` pour ensuite seulement faire les `or`. Pour obtenir le même résultat qu'avec `||`, vous devrez écrire :

```
$xyz = ( $x or $y or $z );
```

La morale de cette histoire, c'est qu'il faut toujours apprendre les règles de précédence (ou utiliser les parenthèses) quels que soient les opérateurs logiques que vous utilisez.

Il existe également un `xor` logique qui n'a pas d'équivalent exact en C ou en Perl, puisque le seul autre opérateur OU-exclusif (`^`) travaille sur les bits. L'opérateur `xor` ne peut pas court-circuiter, car les deux côtés doivent être évalués. Le meilleur équivalent de `$a xor $b` est peut-être `!$a != !$b`. On pourrait également écrire `!$a ^ !$b` ou même `$a ? !$b : !!$b`, bien sûr. L'essentiel est que `$a` et `$b` doivent tous les deux être évalués comme vrais ou faux dans un contexte booléen, et l'opérateur sur les bits existant n'en fournit pas sans qu'on l'y aide.

Opérateurs C manquant en Perl

Voici ce que C a et que Perl n'a pas :

`&` unaire

L'opérateur adresse-de. L'opérateur `\` de Perl (qui fournit une référence) occupe la même niche écologique :

```
$ref_var = \$var;
```

Mais les références de Perl sont plus sûres que les pointeurs de C.

`*` unaire

L'opérateur de déréférencement d'adresse. Comme Perl ne connaît pas les adresses, il n'a pas besoin de les déréférencer. En revanche, Perl a des références et ce sont donc les caractères de préfixe variable qui servent d'opérateurs de déréférence, tout en indiquant le type : `$`, `@`, `%` et `&`. Étonnament, il existe bien un opérateur `*` de déréférencement, mais comme `*` est le drôle de caractère indiquant un typeglob, vous ne l'utiliserez pas ainsi.

`(TYPE)`

L'opérateur de transtypage. De toute façon, personne n'aime se faire transtyper.

4

Instructions et déclarations

Un programme Perl consiste en une séquence de déclarations et d'instuctions. Une déclaration peut-être placée partout où une instruction peut l'être, mais son effet premier se produit à la compilation. Certaines déclarations jouent un double rôle en tant qu'instructions, mais la plupart sont totalement transparentes à l'exécution. Après la compilation, la séquence d'instructions principale est exécutée une seule fois.

Contrairement à beaucoup de langages de programmation, Perl n'impose pas de déclarer explicitement les variables ; elles se mettent à exister à leur première utilisation, que vous les ayez déclarées ou non. Si vous essayez d'utiliser la valeur d'une variable à laquelle aucune valeur n'a jamais été affectée, elle est traitée silencieusement comme si elle contenait `0` si vous vouliez un nombre, comme `""` si vous vouliez une chaîne ou simplement comme une valeur fausse si vous vouliez une valeur logique. Si vous préférez être averti de l'utilisation de valeurs non définies comme si elles étaient de véritables chaînes ou nombres, ou même traiter cette utilisation comme une erreur, la déclaration `use warnings` s'en chargera ; voir la section *Pragmas* à la fin de ce chapitre.

Cependant, si vous préférez, vous pouvez *éventuellement* déclarer vos variables, en utilisant soit `my` ou `our` avant le nom de variable. Vous pouvez même faire qu'utiliser une variable non déclarée soit une erreur. C'est bien de vouloir de la discipline, encore faut-il la demander. Normalement, Perl ne s'intéresse pas à vos habitudes de programmation ; mais avec la déclaration `use strict`, l'utilisation de variables non déclarées est détectée à la compilation. De nouveau, voir la section *Pragmas*.

Instructions simples

Une instruction simple est une expression évaluée pour ses effets secondaires. Toute instruction simple doit se terminer par un point-virgule, sauf si c'est la dernière instruction d'un bloc. Dans ce cas, le point-virgule est optionnel — Perl sait que vous en avez fini avec cette instruction, puisque vous avez fini le bloc. Mais mettez quand même le point-virgule s'il s'agit d'un bloc multiligne, car vous pourriez bien ajouter une autre ligne par la suite.

Bien que des opérateurs comme `eval {}`, `do {}` et `sub {}` ressemblent à des instructions composées, en fait ce n'en sont pas. Certes, ils permettent d'avoir plusieurs instructions à l'intérieur, mais cela ne compte pas. Vus de l'extérieur, ces opérateurs sont juste des termes dans une expression ; c'est pourquoi ils nécessitent un point-virgule explicite quand ils sont utilisés comme dernier élément d'une instruction.

Toute instruction simple peut être optionnellement suivi d'un modificateur unique, juste avant le point-virgule final (ou la fin de bloc). Les modificateurs possibles sont :

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach LISTE
```

Les modificateurs `if` et `unless` fonctionnent comme peuvent s'y attendre les anglophones :

```
$trash->take('out') if $you_love_me;    # sors la poubelle si tu m'aimes
shutup() unless $you_want_me_to_leave;  # tais-toi, sauf si tu veux que
                                        # je m'en aille
```

Les modificateurs `while` et `until` sont évalués de façon répétée. Comme notre lectorat anglophone pouvait s'y attendre, un modificateur `while` exécute l'expression tant que sa propre expression reste vraie, tandis qu'un modificateur `until` continue de s'exécuter tant qu'elle reste fausse :

```
$expression++ while -e "$file$expression";
kiss('me') until $I_die;                # embrasse-moi jusqu'à la mort
```

Le modificateur `foreach` (aussi orthographié `for`) est évalué une fois par élément de sa *LISTE*, `$_` étant un alias de l'élément courant :

```
s/java/perl/ for @curriculum;
print "champ: $_\n" foreach split /:/, $ligne;
```

Les modificateurs `while` et `until` ont la sémantique usuelle des boucles while (la condition est évaluée en premier), sauf quand ils s'appliquent à un `do`*BLOC* (ou à l'instruction maintenant dépréciée `do`*SUBROUTINE*), auquel cas le bloc s'exécute une fois avant que la condition soit évaluée. Cela vous permet d'écrire des boucles comme :

```
do {
    $ligne = <STDIN>;
    ...
} until $ligne eq ".\n";
```

Voyez aussi les trois différentes entrées de `do` au chapitre 29, *Fonctions*. Remarquez également que les opérateurs de contrôle de boucle décrits plus loin ne fonctionnent pas dans cette construction, car les modificateurs de boucles ne prennent pas d'étiquette. Vous pouvez toujours placer un bloc supplémentaire autour pour terminer plus tôt, ou à l'intérieur pour itérer avant la fin de la boucle, comme cela est décrit dans la section *Blocs simples*. Ou bien vous pourriez écrire une vraie boucle avec plusieurs commandes de contrôle de boucle à l'intérieur. À propos de vraies boucles, nous allons maintenant parler des instructions composées.

Instructions composées

On appelle *bloc* une séquence d'instructions définie dans une portée[1]. Parfois la portée s'étend sur un fichier tout entier, comme par exemple un fichier appelé avec `require` ou le fichier contenant votre programme principal. D'autres fois la portée a l'étendue d'une chaîne évaluée avec `eval`. Mais en général, un bloc est entouré d'accolades ({}). Quand nous parlons de portée, cela signifie n'importe laquelle de ces trois possibilités. Quand nous voudrons dire un bloc entouré d'accolades, nous emploierons le terme *BLOC*.

Les instructions composées sont construites à partir d'expressions et de *BLOC*. Les expressions sont construites à partir de termes et d'opérateurs. Dans nos descriptions syntaxiques, nous utiliserons le mot *EXPR* pour indiquer un emplacement où vous pouvez utiliser n'importe quelle expression scalaire. Pour indiquer une expression évaluée en contexte de liste, nous dirons *LISTE*.

Les constructions suivantes peuvent être utilisées pour contrôler l'exécution de *BLOC* de façon conditionnelle ou répétée. (L'étiquette *LABEL* est optionnelle.)

```
if (EXPR) BLOC
if (EXPR) BLOC else BLOC
if (EXPR) BLOC elsif (EXPR) BLOC ...
if (EXPR) BLOC elsif (EXPR) BLOC ... else BLOC

unless (EXPR) BLOC
unless (EXPR) BLOC else BLOC
unless (EXPR) BLOC elsif (EXPR) BLOC ...
unless (EXPR) BLOC elsif (EXPR) BLOC ... else BLOC

LABEL while (EXPR) BLOC
LABEL while (EXPR) BLOC continue BLOC

LABEL until (EXPR) BLOC
LABEL until (EXPR) BLOC continue BLOC

LABEL for (EXPR; EXPR; EXPR) BLOC

LABEL foreach (LISTE) BLOC
LABEL foreach VAR (LISTE) BLOC
LABEL foreach VAR (LISTE) BLOC continue BLOC

LABEL BLOC
LABEL BLOC continue BLOC
```

Remarquez qu'à l'inverse de C et de Java, elles sont définies en termes de *BLOC*, et non d'instructions. Cela signifie que les accolades sont obligatoires ; les instructions isolées ne sont pas autorisées. Si vous voulez écrire des conditions sans accolades, il y a plusieurs manières de le faire. Les lignes suivantes font toutes la même chose :

1. Les portées et les espaces de nommage sont décrits au chapitre 2, *Composants de Perl*, à la section *Noms*.

```
unless (open(TOTO, $toto))    { die "Impossible d'ouvrir $toto: $!" }
if (!open(TOTO, $toto))       { die "Impossible d'ouvrir $toto: $!" }

die "Impossible d'ouvrir $toto: $!"   unless open(TOTO, $toto);
die "Impossible d'ouvrir $toto: $!"   if !open(TOTO, $toto);

open(TOTO, $toto)             || die "Impossible d'ouvrir $toto: $!";
open TOTO, $toto              or die "Impossible d'ouvrir $toto: $!";
```

Nous avons tendance à préférer les deux dernières dans la plupart des cas. Elles sont plus lisibles que les autres, en particulier la version « `or die` ». Avec `||` vous devez vous habituer à utiliser les parenthèses religieusement, tandis qu'avec la version `or`, ce n'est pas grave de les oublier.

Mais la principale raison pour laquelle nous préférons les dernières versions est qu'elles mettent la partie la plus importante de l'instruction au début de la ligne, là où vous la verrez le mieux. La gestion d'erreur est repoussée vers la droite, où vous n'avez pas besoin d'y faire attention, sauf si vous le voulez[2]. Et si vous tabulez tous vos tests « `or die` » à la même position à droite de chaque ligne, c'est encore plus facile à lire :

```
chdir $dir                  or die "chdir $dir: $!";
open TOTO, $fichier         or die "open $fichier: $!";
@lines = <TOTO>             or die "$fichier est vide ?";
close TOTO                  or die "close $fichier: $!";
```

Instructions if et unless

L'instruction `if` est simple. Comme les *`BLOC`*s sont délimités par des accolades, il n'y jamais d'ambiguïté pour savoir à quel `if` en particulier un `else` ou un `elsif` est lié. Dans une séquence donnée de *`BLOC`*s `if`/`elsif`/`else`, seul le premier dont la condition est vraie est exécuté. Si aucune des conditions n'est vraie, alors le *`BLOC`* `else`, s'il existe, est exécuté. C'est en général une bonne idée de mettre un `else` à la fin d'une chaîne de `elsif`, afin de se prémunir contre un cas oublié.

Si vous utilisez `unless` à la place de `if`, le sens du test est inversé. C'est-à-dire que :

```
unless ($x == 1) ...
```

est équivalent à :

```
if ($x != 1) ...
```

ou au plus laid :

```
if (!($x == 1)) ...
```

La portée d'une variable déclarée dans la condition de contrôle s'étend de sa déclaration jusqu'à la fin de l'instruction conditionnelle, y compris tous les `elsif` et l'éventuelle clause `else` finale, mais pas plus loin :

```
if ((my $couleur = <STDIN>) =~ /rouge/i) {
    $valeur = 0xff0000;
}
```

2. Tout comme pour cette note.

```
elsif ($couleur =~ /vert/i) {
    $valeur = 0x00ff00;
}
elsif ($couleur =~ /bleu/i) {
    $valeur = 0x0000ff;
}
else {
    warn "`$couleur' : composante RGB inconnue, le noir est sélectionné\n";
    $valeur = 0x000000;
}
```

Après le `else`, la variable `$couleur` est hors de portée. Si vous voulez que sa portée s'étende plus loin, déclarez la variable plus tôt.

Instructions de boucle

Dans leur syntaxe formelle, toutes les instructions de boucle comportent un *`LABEL`* (ou étiquette) facultatif. (Vous pouvez mettre une telle étiquette sur n'importe quelle instruction, mais cela a une signification spéciale pour les boucles.) S'il est présent, le label consiste en un identificateur suivi de deux-points. Il est courant d'écrire l'étiquette en majuscules pour éviter tout conflit avec des mots réservés et les faire mieux ressortir. Bien que Perl ne se pose pas de problème si vous utilisez un label qui a déjà une signification comme `if` ou `open`, vos lecteurs risquent de se tromper.

Instructions while et until

L'instruction `while` exécute le bloc tant que *`EXPR`* est vraie. Si le mot `while` est remplacé par `until`, le sens du test est inversé ; c'est-à-dire qu'il exécute le bloc tant que *`EXPR`* est fausse. La condition est cependant toujours testée avant la première itération.

Les instructions `while` ou `until` comportent un bloc supplémentaire facultatif : le bloc `continue`. Ce bloc est exécuté à chaque fois que l'on achève l'itération, soit en sortant à la fin du premier bloc, soit par un `next` explicite (`next` est un opérateur de contrôle de boucle qui passe à l'itération suivante). En pratique, le bloc `continue` n'est pas très utilisé, mais il est présent afin de pouvoir définir rigoureusement la boucle `for` dans la section qui suit.

Contrairement à la boucle `for` que nous allons voir dans un moment, une boucle `while` ne localise jamais implicitement de variables dans sa condition de test. Cela peut avoir d'« intéressantes » conséquences quand des boucles `while` utilisent des variables globales comme variables de boucle. En particulier, consultez la section *Opérateur de lecture de ligne (Angle)* au chapitre 2 pour voir comment une affectation implicite à la variable globale `$_` peut se produire dans certaines boucles `while`, ainsi qu'un exemple de gestion de ce problème en localisant explicitement `$_`. Il est cependant préférable de déclarer les autres variables de boucle avec `my`, comme dans l'exemple suivant.

Une variable déclarée dans la condition de test d'une instruction `while` ou d'un `until` est visible seulement dans le ou les blocs pilotés par ce test. Sa portée ne s'étend pas au-delà. Par exemple :

```
while (my $ligne = <STDIN>) {
```

```
    $ligne = lc $ligne;
}
continue {
    print $ligne;   # toujours visible
}
# $ligne maintenant hors de portée
```

Ici, la portée `$ligne` s'étend de sa déclaration dans l'expression de contrôle à l'ensemble de la boucle, y compris le bloc `continue`, mais pas au-delà. Si vous voulez que sa portée s'étende plus loin, déclarez la variable avant la boucle.

Boucles for

La boucle `for` en trois parties comporte trois expressions séparées par des points-virgules entre ses parenthèses. Ces expressions sont respectivement l'initialisation, la condition et la ré-initialisation de la boucle. Ces trois expressions sont optionnelles (mais pas les points-virgules) ; si la condition est omise, elle est toujours vraie. La boucle `for` peut être définie dans les termes de la boucle `while` correspondante. Ce qui suit :

```
LABEL:
  for (my $i = 1; $i <= 10; $i++) {
      ...
  }
```

est donc identique à :

```
{
    my $i = 1;
  LABEL:
    while ($i <= 10) {
        ...
    }
    continue {
        $i++;
    }
}
```

sinon qu'il n'y a pas vraiment de bloc extérieur. (Nous l'avons juste mis là pour montrer les limites du `my`.)

Si vous voulez itérer deux variables simultanément, il vous suffit de séparer les expressions parallèles par des virgules :

```
for ($i = 0, $bit = 0; $i < 32; $i++, $bit <<= 1) {
    print "Le bit $i is est à 1\n" if $mask & $bit;
}
# les valeurs de $i et $bit persistent après la boucle
```

Ou bien déclarer ces variables comme visibles seulement à l'intérieur de la boucle `for` :

```
for (my ($i, $bit) = (0, 1); $i < 32; $i++, $bit <<= 1) {
    print "Le bit $i is est à 1\n" if $mask & $bit;
}
# les $i et $bit de la boucle sont maintenant hors de portée
```

En plus des boucles habituelles sur les indices de tableaux, `for` a d'autres applications intéressantes. Il n'a même pas besoin d'une variable de boucle explicite. Voici un exemple permettant d'éviter le problème que vous rencontrez en testant explicitement la fin de fichier sur un descripteur de fichier interactif, provoquant ainsi le blocage du programme.

```
$sur_un_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $sur_un_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # fait quelque chose
}
```

Une autre application traditionnelle pour le `for` en trois parties vient du fait que les trois expressions sont optionnelles et que la condition par défaut est vraie. Si vous omettez les trois expressions, vous obtenez une boucle infinie :

```
for (;;) {
    ...
}
```

Ce qui est exactement équivalent à :

```
while (1) {
    ...
}
```

Si la notion de boucle infinie vous inquiète, nous devrions souligner que vous pouvez toujours sortir de la boucle quand vous voulez avec un opérateur de contrôle explicite de boucle comme `last`. Bien sûr, si vous écrivez le code de contrôle d'un missile de croisière, vous n'aurez jamais vraiment besoin de sortir de la boucle. Elle se terminera automatiquement au moment opportun.[3]

Boucles foreach

La boucle `foreach` parcourt une liste de valeurs en affectant tour à tour chaque élément de la liste à la variable de contrôle (*VAR*) :

```
foreach VAR (LISTE) {
    ...
}
```

Le mot-clé `foreach` est juste un synonyme du mot-clé `for`, vous pouvez donc utiliser au choix `for` et `foreach`, selon lequel vous trouvez le plus lisible dans une situation donnée. Si *VAR* est omis, la variable globale `$_` est utilisée. (Ne vous inquiétez pas — Perl distingue facilement `for (@ARGV)` de `for ($i=0; $i<$#ARGV; $i++)`, car ce dernier contient des points-virgules.) Voici quelques exemples :

```
$somme = 0; foreach $valeur (@tableau) { $somme += $valeur }

for $decompte (10,9,8,7,6,5,4,3,2,1,'BOUM') { # compte à rebours
    print "$decompte\n"; sleep(1);
}
```

3. C'est-à-dire que les retombées de la boucle ont tendance à se produire automatiquement.

```
for (reverse 'BOUM', 1 .. 10) {              # pareil
    print "$_\n"; sleep(1);
}

for $champ (split /:/, $data) {              # toute expression de LISTE
    print "Le champ contient : '$champ'\n";
}

foreach $cle (sort keys %hash) {
    print "$cle => $hash{$cle}\n";
}
```

Le dernier représente la manière classique d'afficher les valeurs d'un hachage dans l'ordre des clés. Consultez les entrées `keys` et `sort` au chapitre 29 pour des exemples plus élaborés.

Il n'existe aucune manière de savoir où vous en êtes dans une liste avec `foreach`. Vous pouvez comparer des éléments adjacents en vous souvenant du précédent dans une variable, mais il faudra parfois vous contenter d'écrire une boucle `for` en trois parties avec des indices. Après tout, cet autre `for` est justement là pour ça.

Si *LISTE* est entièrement constituée d'éléments auxquels on peut affecter une valeur (c'est-à-dire de variables, pas d'une énumération de constantes), vous pouvez modifier chacune de ces variables en modifiant *VAR* à l'intérieur de la boucle. Cela provient du fait que la variable d'indice de la boucle `foreach` est un alias implicite de chaque élément de la liste sur laquelle vous bouclez. Vous pouvez non seulement modifier un tableau d'un coup, mais également plusieurs tableaux ou hachages dans une seule liste :

```
foreach $paye (@salaires) {             # une augmentation de 8%
    $paye *= 1.08;
}

for (@noel, @paques) {                  # change le menu
    s/pâté/foie gras/;
}
s/pâté/foie gras/ for @noel, @paques;   # idem

for ($scalaire, @tableau, values %hash) {
    s/^\s+//;                           # retire les espaces initiaux
    s/\s+$//;                           # retire les espaces finaux
}
```

La variable de boucle est valide uniquement dans la portée dynamique ou lexicale de la boucle et sera implicitement lexicale si la variable a été précédemment déclarée avec `my`. Cela la rend invisible à toute fonction définie en dehors de la portée lexicale de la variable, même si elle est appelée depuis cette boucle. Cependant si aucune déclaration lexicale n'est à portée, la variable de boucle sera une variable globale localisée (à portée dynamique) ; cela permet aux fonctions appelées depuis la boucle d'accéder à cette variable. Dans tous les cas, la valeur qu'avait la variable localisée avant la boucle sera restaurée à la sortie de la boucle.

Si vous préférez, vous pouvez déclarer explicitement quel type de variable (lexicale ou globale) utiliser. Ceux qui maintiennent votre code sauront plus facilement ce qui se

passe ; sinon, ils devront remonter la chaîne des portées successives à la recherche d'une déclaration pour deviner de quelle variable il s'agit :

```
for my  $i    (1 .. 10) { ... }         # $i toujours lexicale
for our $Tick (1 .. 10) { ... }         # $Tick toujours globale
```

Quand une déclaration accompagne la variable de boucle, l'écriture courte `for` est toujours préférable à `foreach`, car cela se lit mieux en anglais.[4]

Voici comment un programmeur C ou Java pourrait d'abord penser à écrire un algorithme donné en Perl :

```
for ($i = 0; $i < @tab1; $i++) {
    for ($j = 0; $j < @tab2; $j++) {
        if ($tab1[$i] > $tab2[$j]) {
            last;   # Impossible d'aller à la boucle externe. :-(
        }
        $tab1[$i] += $tab2[$j];
    }
    # voici où ce last m'emmène
}
```

Mais voici comment un programmeur Perl expérimenté pourrait l'écrire :

```
WID: foreach $ceci (@tab1) {
    JET: foreach $cela (@tab2) {
        next WID if $ceci > $cela;
        $ceci += $cela;
    }
}
```

Vous voyez combien c'était simple en Perl idiomatique ? C'est plus propre, plus sûr et plus rapide. C'est plus propre car il y a moins de bruit. C'est plus sûr car si du code est ajouté par la suite entre les boucles interne et externe, ce code ne sera pas accidentellement exécuté, car `next` (expliqué plus loin) reboucle sur la boucle externe

Mais vous codez comme vous préférez. TMTOWTDI.

Comme l'instruction `while`, l'instruction `foreach` peut aussi avoir un bloc `continue`. Cela vous permet d'exécuter un bout de code à la fin de chaque itération de boucle, que vous en soyez arrivé là par le cours normal des événements ou par un `next`.

Contrôle de boucle

Nous avons déjà mentionné que vous pouviez mettre un *`LABEL`* sur une boucle pour lui donner un nom. L'étiquette identifie la boucle pour les opérateurs de contrôle de boucle `next`, `last` et `redo`. Le *`LABEL`* désigne la boucle toute entière, pas seulement le début de celle-ci. Une commande de contrôle de boucle ne « va » pas au *`LABEL`* lui-même. Pour l'ordinateur l'étiquette aurait aussi bien pu être placée à la fin de la boucle. Mais il semble que les gens préfèrent les étiquettes au début.

4. NdT : D'une manière générale, vous avez maintenant remarqué qu'il est utile d'avoir des notions d'anglais si l'on veut profiter des subtilités de Perl (et de CPAN).

Les boucles sont typiquement nommées d'après les éléments qu'elles manipulent à chaque itération. Cela se combine bien avec les opérateurs de contrôle de boucle, qui sont conçus pour se lire comme de l'anglais quand ils sont utilisés avec une étiquette appropriée et un modificateur d'instruction. La boucle typique traite des lignes, donc l'étiquette de boucle typique est `LINE:` ou `LIGNE:` et l'opérateur de contrôle de ligne typique ressemble à ceci :

```
next LIGNE if /^#/;      # supprime les commentaires
```

La syntaxe des opérateurs de contrôle de boucle est :

```
last LABEL
next LABEL
redo LABEL
```

Le *LABEL* est optionnel ; s'il est omis, l'opérateur se réfère à la boucle englobante la plus interne. Mais si vous voulez sauter plus d'un niveau, vous devez utiliser un *LABEL* pour désigner la boucle sur laquelle agir. Ce *LABEL* n'a pas besoin d'être dans la même portée lexicale que l'opérateur de contrôle, mais c'est probablement préférable. En fait, le *LABEL* peut être n'importe où dans la portée dynamique. Si cela vous force à sortir d'un `eval` ou d'un sous-programme, Perl émet un avertissement (sur demande).

Tout comme vous pouvez avoir autant de `return` que vous voulez dans une fonction, vous pouvez avoir autant d'opérateurs de contrôle de boucle que vous voulez dans une boucle. Cela n'a pas à être considéré comme mauvais ou pas cool. Aux débuts de la programmation structurée, certaines personnes insistaient sur le fait que les boucles et les sous-programmes ne devaient avoir qu'une entrée et qu'une sortie. La notion d'entrée unique est toujours une bonne idée, mais la notion de sortie unique a conduit à l'écriture de beaucoup de code artificiel. La programmation consiste principalement à parcourir des arbres de décision. Un arbre de décision commence naturellement par une racine unique mais se termine par de nombreuses feuilles. Écrivez votre code avec le nombre de sorties de boucle (et de retours de fonction) qui est naturel pour le problème que vous essayez de résoudre. Si vous avez déclaré vos variables dans des portées raisonnables, tout sera automatiquement nettoyé le moment venu, quelle que soit la manière dont vous quittez le bloc.

L'opérateur `last` sort immédiatement de la boucle en question. Le bloc `continue`, s'il existe, n'est pas exécuté. L'exemple suivant s'éjecte de la boucle à la première ligne blanche :

```
LIGNE: while (<STDIN>) {
    last LIGNE if /^$/;      # sort quand l'entête de mail est fini
    ...
}
```

L'opérateur `next` passe le reste de l'itération courante de la boucle et commence la suivante. S'il existe une clause `continue` sur la boucle, elle est exécutée juste avant que la condition soit ré-évaluée, exactement comme la troisième partie d'une boucle `for`. Elle peut donc servir à incrémenter une variable de boucle, même si une itération particulière de la boucle a été interrompue par un `next` :

```
LIGNE: while (<STDIN>) {
    next LIGNE if /^#/;      # ignore les commentaires
    next LIGNE if /^$/;      # ignore les lignes blanches
    ...
```

```
} continue {
    $compte++;
}
```

L'opérateur `redo` redémarre le bloc de boucle sans réévaluer la condition. Le bloc `continue`, s'il existe, n'est pas exécuté. Cet opérateur s'utilise souvent pour des programmes qui veulent se cacher à eux-mêmes ce qui vient d'être entré. Supposons que vous traitez un fichier dont les lignes sont parfois terminées par un antislash pour indiquer qu'elles continuent sur la ligne suivante. Voici un exemple d'utilisation de `redo`dans ce cas :

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;  # ne pas lire au-delà de la fin de chaque fichier
    }
    # traitement de $_
}
```

qui est la version Perl usuelle du plus explicite (et laborieux) :

```
LIGNE: while (defined($ligne = <ARGV>)) {
    chomp($ligne);
    if ($ligne =~ s/\\$//) {
        $ligne .= <ARGV>;
        redo LIGNE unless eof(ARGV);
    }
    # traitement de $ligne
}
```

Voici un exemple tiré d'un programme réel qui utilise les trois opérateurs de contrôle de boucle. Bien que cette méthode soit moins courante maintenant que nous disposons des modules `Getopt::*` dans la distribution de Perl standard, c'est toujours une illustration intéressante de l'utilisation des opérateurs de contrôle de boucle sur des boucles nommées et imbriquées :

```
ARG: while (@ARGV && $ARGV[0] =~ s/^-(?=.)//) {
    OPT: for (shift @ARGV) {
         m/^$/       && do {                              next ARG; };
         m/^-$/      && do {                              last ARG; };
         s/^d//      && do { $Niveau_Debug++;             redo OPT; };
         s/^l//      && do { $Genere_Listing++;           redo OPT; };
         s/^i(.*)//  && do { $Sur_Place = $1 || ".bak";  next ARG; };
         say_usage("Option inconnue : $_");
    }
}
```

Encore un mot au sujet des opérateurs de contrôle de boucle. Vous avez peut-être remarqué que nous ne les appelons pas « instructions ». Ce ne sont en effet pas des instructions — bien que comme toute expression, on puisse les utiliser comme des instructions. Vous pouvez presque les considérer comme des opérateurs unaires qui modifient le déroulement du programme. En fait, vous pouvez même vous en servir là où cela n'a aucun sens. On voit parfois cette erreur de codage :

```
open FICHIER, $fichier
     or warn "Impossible d'ouvrir $fichier : $!\n", next FICHIER; # FAUX
```

L'intention est bonne, mais `next FICHIER` est analysé comme l'un des paramètres de `warn`, qui est un opérateur de liste. Donc le `next` s'exécute avant que le `warn` ait la moindre chance d'émettre son avertissement. Dans ce cas, cela se corrige facilement en changeant l'opérateur de liste `warn` en la fonction `warn` à l'aide de parenthèses bien placées :

```
open FICHIER, $fichier
     or warn("Impossible d'ouvrir $fichier : $!\n"), next FICHIER; # Correct
```

Néanmoins, vous trouverez peut-être ceci plus facile à lire :

```
unless (open FICHIER, $fichier) {
     warn "Impossible d'ouvrir $fichier: $!\n";
     next FICHIER;
}
```

Blocs simples

Un *BLOC* (étiqueté ou non) est en soi l'équivalent sémantique d'une boucle qui s'exécute une seule fois. C'est pourquoi vous pouvez utiliser `last` pour quitter un bloc ou `redo` pour relancer le bloc.[5] Remarquez que ceci n'est pas le cas des blocs à l'intérieur d'`eval {}`, de `sub {}` ou, ce qui en étonne beaucoup, de `do {}`. En effet, ce ne sont pas des blocs de boucle, car ce ne sont pas des blocs par eux-mêmes. Le mot-clé qui les précède fait d'eux les termes d'expressions qui se trouvent contenir un bloc de code. N'étant pas des blocs de boucle, ils ne peuvent être étiquetés et les contrôles de boucle ne s'y appliquent pas. Les contrôles de boucle ne s'appliquent qu'aux véritables boucles, tout comme `return` ne s'utilise que dans un sous-programme (ou un `eval`).

Les contrôles de boucle ne marchent pas non plus avec un `if` ou un `unless`, puisque ce ne sont pas des boucles. Mais vous pouvez toujours ajouter une paire d'accolades supplémentaires pour construire un bloc simple qui *lui* est une boucle :

```
if (/pattern/) {{
    last if /alpha/;
    last if /beta/;
    last if /gamma/;
    # ne fait quelque chose que si on se trouve encore dans le if()
}}
```

Voici comment utiliser un bloc pour faire fonctionner les opérateurs de contrôle de boucle avec une construction `do {}`. Pour faire un `next` ou un `redo` sur un `do`, ajouter un bloc simple à l'intérieur :

```
do {{
    next if $x == $y;
    # faire quelque chose ici
}} until $x++ > $z;
```

5. Pour des raisons qui peuvent (ou non) paraître évidentes à la réflexion, un `next` permet également de sortir d'un bloc. Il y a cependant une petite différence en ce qu'un `next` exécute un bloc `continue`, ce qui n'est pas le cas d'un `last`.

Vous devez être plus subtil pour `last` :

```
{
    do {
        last if $x = $y ** 2;
        # faire quelque chose ici
    } while $x++ <= $z;
}
```

Et si vous voulez vous servir des deux contrôles de boucle disponibles, vous allez devoir distinguer ces blocs avec des étiquettes :

```
DO_LAST: {
            do {
DO_NEXT:          {
                    next DO_NEXT if $x == $y;
                    last DO_LAST if $x =  $y ** 2;
                    # faire quelque chose ici
                  }
            } while $x++ <= $z;
         }
```

Mais quand vous en arrivez à ce point (ou même avant), vous feriez mieux d'utiliser une simple boucle infinie avec un `last` à la fin :

```
for (;;) {
    next if $x == $y;
    last if $x =  $y ** 2;
    # faire quelque chose ici
    last unless $x++ <= $z;
}
```

Structures de cas

Contrairement à d'autres langages de programmation, Perl n'a pas d'instruction `switch` ou `case` officielle. Perl n'en a pas besoin, puisqu'il y a plus d'une manière de faire la même chose. Un bloc simple est particulièrement commode pour construire des structures à choix multiples. En voici une :

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $rien = 1;
}
```

et une autre :

```
SWITCH: {
    /^abc/      && do { $abc = 1; last SWITCH; };
    /^def/      && do { $def = 1; last SWITCH; };
    /^xyz/      && do { $xyz = 1; last SWITCH; };
    $rien = 1;
}
```

ou, formaté pour que chaque cas ressorte mieux :

```
SWITCH: {
    /^abc/      && do {
                        $abc = 1;
                        last SWITCH;
                   };
    /^def/      && do {
                        $def = 1;
                        last SWITCH;
                   };
    /^xyz/      && do {
                        $xyz = 1;
                        last SWITCH;
                    };
    $rien = 1;
}
```

ou même, horreur :

```
if    (/^abc/) { $abc = 1 }
elsif (/^def/) { $def = 1 }
elsif (/^xyz/) { $xyz = 1 }
else           { $rien = 1 }
```

Remarquez comme dans cet exemple l'opérateur `last` ignore les blocs `do`, qui ne sont pas des boucles, et sort de la boucle `for` :

```
for ($nom_de_variable_tres_long[$i++][$j++]->methode()) {
    /ce motif/              and do { push @flags, '-e'; last; };
    /celui-là/              and do { push @flags, '-h'; last; };
    /quelque chose d'autre/ and do {                    last; };
    die "valeur inconnue: `$_'";
}
```

Vous pouvez trouver bizarre de boucler sur une seule valeur, puisque vous n'allez traverser qu'une fois la boucle. mais il est commode de pouvoir utiliser les capacités d'alias de `for`/`foreach` pour affecter `$_` temporairement et localement. Cela rend les comparaisons multiples à la même valeur beaucoup plus faciles à taper et il est donc plus difficile de se tromper. On échappe aux effets secondaires d'une nouvelle évaluation de l'expression. Et pour rester en rapport avec cette section, c'est l'un des idiomes les plus répandus pour implémenter une structure de cas.

Pour des cas simples, une cascade d'opérateurs `?:` peut également marcher. Ici encore, nous utilisons les capacités d'alias de `for` afin de rendre les comparaisons multiples plus lisibles :

```
for ($couleur_utilisateur) {
    $value = /rouge/    ?   0xFF0000  :
             /vert/     ?   0x00FF00  :
             /bleu/     ?   0x0000FF  :
                            0x000000  ;  # noir s'il n'y a plus d'espoir
}
```

Dans des situations comme celle-ci, il vaut parfois mieux vous construire un hachage et

le classer rapidement pour en tirer la réponse. Contrairement aux conditions en cascade que nous avons vues, un hachage peut croître à un nombre illimité d'éléments sans prendre plus de temps pour trouver le premier ou le dernier élément. L'inconvénient est que vous ne pourrez faire que des comparaisons exactes et pas des recherches de motif. Si vous avez un hachage comme celui-ci :

```
%couleur = (
    azur        => 0xF0FFFF,
    chartreuse  => 0x7FFF00,
    lavande     => 0xE6E6FA,
    magenta     => 0xFF00FF,
    turquoise   => 0x40E0D0,
);
```

alors une recherche exacte de chaîne tourne vite :

```
$valeur = $couleur{ lc $couleur_utilisateur } || 0x000000;
```

Même les instructions compliquées de branchement multiple (où chaque cas implique l'exécution de plusieurs instructions différentes) peuvent se transformer en une rapide consultation. Vous avez juste besoin de vous servir d'un hachage de références à des functions. Pour savoir comment les manipuler, voyez la section *Hachages de fonctions* au chapitre 9, *Structures de données*.

goto

Perl supporte aussi un opérateur `goto`, mais il n'est pas destiné aux cœurs sensibles. Il se présente sous trois formes : `goto` *`LABEL`*, `goto` *`EXPR`* et `goto &`*`NOM`*.

La forme `goto` *`LABEL`* trouve l'instruction étiquetée avec *`LABEL`* et reprend l'exécution à partir de là. Elle ne peut pas servir à sauter à l'intérieur d'une structure qui nécessite une initialisation, comme un sous-programme ou une boucle `foreach`. Elle ne peut pas non plus servir pour entrer dans une structure qui a été éliminée lors de l'optimisation (voir le chapitre 18, *Compilation*). Elle peut servir pour aller à peu près n'importe où dans le bloc courant ou dans un bloc dans votre portée dynamique (c'est-à-dire un bloc duquel vous avez été appelé). Vous pouvez même sortir d'un sous-programme par `goto`, mais il vaut en général mieux utiliser une autre construction. L'auteur de Perl n'a jamais ressenti le besoin d'utiliser cette forme de `goto` (en Perl ; en C, c'est une autre affaire).

La forme `goto` *`EXPR`* n'est qu'une généralisation de `goto` *`LABEL`*. Elle attend de l'expression qu'elle produise un nom d'étiquette, dont la position doit évidemment être résolue dynamiquement par l'interprêteur. Cela permet des `goto` calculés à la FORTRAN, mais n'est pas nécessairement recommandé si vous cherchez à optimiser la maintenabilité du code :

```
goto(("TOTO", "TITI", "TUTU")[$i]);         # en espérant que 0 <= i < 3

@loop_label = qw/TOTO TITI TUTU/;
goto $loop_label[rand @loop_label];         # téléportation au hasard
```

Dans presque tous les cas de ce genre, il est très, très largement préférable d'utiliser les méchanismes structurés de contrôle de flux de `next`, `last` ou `redo` au lieu d'avoir recours à un `goto`. Pour certaines applications, un hachage de références à des fonctions

ou le méchanisme de capture et de gestion d'exceptions s'appuyant sur `eval` et `die` sont des approches prudentes.

La forme `goto &NAME` est fortement magique et suffisamment éloignée du `goto`usuel pour exempter ceux qui l'utilisent de l'opprobre qui couvre habituellement les utilisateurs de `goto`. Elle substitue à la routine en cours d'exécution un appel au sous-programme nommé. Ce fonctionnement est utilisé par les routines `AUTOLOAD` pour charger un autre sous-programme et faire croire que c'est cet autre sous-programme qui était appelé. Après le `goto`, même `caller` ne pourra dire que cette routine a été appelée en premier. Les modules `autouse`, `AutoLoader` et `SelfLoader` utilisent tous cette stratégie pour définir les fonctions lorsqu'elles sont appelées pour la première fois puis les lancer sans que personne ne puisse jamais savoir que ces fonctions n'ont pas toujours été là.

Déclarations globales

Les déclarations de sous-programmes et de formats sont des déclarations globales. Où que vous les placiez, ce qu'elles déclarent est global (c'est local au paquetage, mais comme les paquetages sont globaux au programme, tout ce qui est dans un paquetage est visible de partout). Une déclaration globale peut être placée partout où l'on peut mettre une instruction, mais n'a aucun effet sur l'exécution de la séquence primaire d'instructions, les déclarations prennent effet à la compilation.

Cela signifie que vous ne pouvez pas faire de déclaration conditionnelle de sous-programmes ou de formats et les cacher du compilateur au moyen d'une condition qui ne sera prise en compte qu'à l'exécution. Le compilateur voit les déclarations de sous-programmes et de formats (ainsi que les déclarations `use` et `no`) où qu'elles se produisent.

Les déclarations globales sont généralement mises au début ou à la fin de votre programme, ou dans un autre fichier. Cependant si vous déclarez des variables à portée lexicale (voir la section suivante), vous devrez vous assurer que vos déclarations de formats et de sous-programmes sont à portée de vos déclarations de variables si vous espérez accéder à ces variables privées.

Vous avez remarqué que nous sommes sournoisement passés des déclarations aux définitions. Séparer la *définition* de la *déclaration* a parfois un intérêt. La seule différence syntaxique entre les deux est que la définition fournit un *BLOC* contenant le code à exécuter, et pas la déclaration. (Une définition de sous-programme agit comme une déclaration si aucune déclaration n'a été vue.) Séparer la définition de la déclaration vous permet de mettre la déclaration au début du fichier et la définition à la fin (avec vos variables lexicales joyeusement au milieu) :

```
sub compte (@);       # Le compilateur sait maintenant comment appeler
                      # compte().
my $x;                # Le compilateur connaît maintenant la variable
                      # lexicale.
$x = compte(3,2,1);   # Le compilateur peut valider l'appel de fonction.
sub compte (@) { @_ } # Le compilateur sait maintenant ce que fait compte().
```

Comme le montre cet exemple, les sous-programmes n'ont pas besoin d'être définis avant que les appels vers eux soient compilés (en fait, leur définition peut même être repoussée jusqu'à leur première utilisation, si vous utilisez l'autochargement), mais la

déclaration des sous-programmes aide le compilateur de différentes manières et vous donne plus de choix dans votre façon de les appeler.

La déclaration d'un sous-programme lui permet d'être utilisé sans parenthèses, comme s'il s'agissait d'un opérateur intégré depuis ce point de la compilation. (Nous avons utilisé des parenthèses pour appeler `compte` dans l'exemple précédent, mais en fait nous n'en avions pas besoin.) Vous pouvez déclarer un sous-programme sans le définir juste en disant :

```
sub monnom;
$me = monnom $0        or die "Impossible de trouver mon nom";
```

Une déclaration simple comme celle-ci déclare la fonction comme un opérateur de liste et non comme un opérateur unaire, aussi faites attention à utiliser `or` et non `||` dans ce cas. L'opérateur `||` lie trop fortement pour être utilisé après des opérateurs de liste, même si vous pouvez toujours mettre des parenthèses autour des arguments de l'opérateur de liste pour le changer en appel de fonction. Autrement, vous pouvez utiliser le prototype `($)` pour transformer la routine en opérateur unaire :

```
sub monnom ($);
$me = monnom $0        || die "Impossible de trouver mon nom";
```

C'est maintenant analysé comme vous vous y attendez, mais vous devriez tout de même garder l'habitude d'utiliser `or` dans cette situation. Pour en savoir plus sur les prototypes, voir le chapitre 6, *Sous-programmes*.

Vous *devez* définir le sous-programme à un moment donné, sinon vous obtiendrez une erreur à l'exécution indiquant que vous avez appelé un sous-programme indéfini. À moins de définir le sous-programme vous-même, vous pouvez récupérer les définitions depuis d'autres endroits de plusieurs façons.

Vous pouvez charger les définitions depuis d'autres fichiers avec une simple instruction `require` ; c'était la meilleure manière de charger des fichiers en Perl 4, mais elle pose deux problèmes. Premièrement, l'autre fichier va typiquement insérer des noms de sous-programmes dans un paquetage (une table de symboles) de son choix, et non vos propres paquetages. Deuxièmement, un `require` se produit à l'exécution et donc arrive trop tard pour servir de déclaration dans le fichier invoquant le `require`. Il arrive cependant que vous cherchiez justement à retarder le chargement.

Une manière plus utile de charger les déclarations et les définitions est la déclaration `use`, qui fait un `require` du module à la compilation (car `use` compte comme un bloc `BEGIN`) et vous laisse importer une partie des déclarations du module dans votre propre programme. On peut donc considérer `use` comme une déclaration globale en ce qu'elle importe les noms à la compilation dans votre propre paquetage (global) comme si vous les aviez déclarés vous-même. Voir la section *Tables de symboles*, au chapitre 10, *Paquetages*, au sujet du fonctionnement bas niveau de l'importation entre paquetages, le chapitre 11, *Modules*, pour savoir comment configurer les paramètres d'importation et d'exportation des modules, et le chapitre 18 pour une explication de `BEGIN` et de ses cousins `CHECK`, `INIT` et `END`, qui sont aussi des sortes de déclarations globales puisqu'elles sont traitées à la compilation et peuvent avoir un effet global.

Déclarations avec portée

Comme les déclarations globales, les déclarations à portée lexicale ont un effet au moment de la compilation. Contrairement aux déclarations globales, les déclarations à portée lexicale ne s'appliquent que du point de déclaration jusqu'à la fin du bloc encadrant le plus interne (le premier bloc, fichier ou `eval` trouvé). C'est pourquoi nous les appelons à portée lexicale, bien que le terme « portée textuelle » soit peut-être plus exact, puisque la portée lexicale n'a rien à voir avec les lexiques. Mais les informaticiens du monde entier savent ce que signifie « portée lexicale », aussi perpétuons-nous cet usage ici.

Perl supporte aussi les déclarations à portée dynamique. Une *portée dynamique* s'étend aussi jusqu'à la fin du bloc encadrant le plus interne, mais « encadrant » dans ce cas est défini dynamiquement à l'exécution, plutôt que textuellement à la compilation. Pour le dire autrement, les bloc s'emboîtent dynamiquement en invoquant d'autres blocs, pas en les incluant. Cet emboîtement de portées dynamiques peut être corrélé en quelque sorte à l'emboîtement des portées lexicales, mais les deux sont en général différents, particulièrement quand des sous-programmes ont été invoqués.

Nous avons mentionné que certains aspects de `use` pouvaient être considérés comme des déclarations globales, mais d'autres aspects de `use` sont à portée lexicale. En particulier, `use` n'importe pas seulement les symboles, mais implémente également diverses directives de compilation magiques, connues sous le nom de *pragmas* (ou si vous préférez la forme classique, *pragmata*). La plupart des pragmas sont à portée lexicale, y compris le pragma `use strict 'vars'` qui vous oblige à déclarer vos variables avant de vous en servir. Voir plus loin la section *Pragmas*.

Une déclaration `package`, curieusement, est elle-même à portée lexicale, malgré le fait qu'un paquetage est une entité globale. Mais une déclaration `package` se contente de déclarer l'identité du paquetage par défaut pour le reste du bloc l'encadrant. Les noms de variables non déclarés, non qualifiés[6] sont recherchés dans ce paquetage. En un sens, un paquetage n'est jamais déclaré du tout, mais il se met à exister quand vous faites référence à quelque chose qui appartient à ce paquetage. C'est très Perlien.

Déclarations de variables à portée limitée

Le reste de ce chapitre parle surtout de l'utilisation de variables globales. Ou plutôt, parle de la *non* utilisation de variables globales. Il existe plusieurs déclarations qui vous aident à ne pas utiliser de variables globales — ou au moins à ne pas les utiliser bêtement.

Nous avons déjà mentionné la déclaration `package`, qui a été introduite en Perl il y a bien longtemps pour pouvoir séparer les variables globales en paquetages séparés. Cela marche assez bien pour un certain type de variables. Les paquetages sont utilisés par des bibliothèques, des modules et des classes pour stocker leurs données d'interface (et certaines de leur données semi-privées) pour éviter les conflits avec des variables et des

6. Et aussi les noms de sous-programmes, handles de fichiers, handles de répertoires et formats non qualifiés.

fonctions de même nom dans votre programme principal ou dans d'autres modules. Si vous voyez quelqu'un écire `$Quelque::chose`,[7] il se sert de la variable scalaire `$chose` du paquetage `Quelque`. Voir le chapitre 10.

Si c'était tout ce qu'il existait en la matière, les programmes Perl deviendraient de plus en plus difficiles à manipuler en grossissant. Heureusement, les trois déclarations de portée de Perl permettent de créer facilement des variables complètement privées (avec `my`), de donner sélectivement l'accès aux globales (avec `our`) et de donner des valeurs temporaires à des variables globales (avec `local`).

```
my $nose;
our $House;
local $TV_channel;
```

Si plusieurs variables sont listées, la liste doit être placée entre parenthèses. Pour `my` et `our`, les éléments ne peuvent être que de simples scalaires, tableaux ou hachages. Pour `local`, les contraintes sont quelque peu relâchées : vous pouvez également localiser des typeglobs en entier ou de simples éléments, ou des tranches de tableaux ou de hachages :

```
my ($nose, @eyes, %teeth);
our ($House, @Autos, %Kids);
local (*Spouse, $phone{HOME});
```

Chacun de ces modificateurs propose une sorte d'isolation différente aux variables qu'il modifie. Pour simplifier légèrement : `our` confine les noms à une portée, `local` confine les valeurs à une portée, et `my` confine à la fois les noms et les valeurs à une portée.

Ces constructions peuvent se voir affecter des valeurs, mais diffèrent en ce qu'elles font réellement avec ces valeurs, puisqu'elles proposent des mécanismes différents de stockage des valeurs. Elles sont aussi quelque peu différentes quand vous ne leur affectez *aucune* valeur (comme dans notre exemple ci-dessus) : `my` et `local` font démarrer les variables en question à la valeur `undef` ou `()` appropriée selon le contexte ; `our` au contraire ne modifie pas la valeur de la variable globale associée.

Syntaxiquement, `my`, `our` et `local` sont simplement des modificateurs (comme des adjectifs) agissant sur une *lvalue*. Quand vous affectez à une *lvalue* modifiée, le modificateur ne change pas le fait que la *lvalue* soit vue comme un scalaire ou une liste. Pour savoir comment le modificateur va fonctionner, faites comme s'il n'était pas là. Ces deux constructions fournissent donc un contexte de liste du côté droit :

```
my ($toto)  = <STDIN>;
my @tableau = <STDIN>;
```

Tandis que celle-ci fournit un contexte scalaire :

```
my $toto = <STDIN>;
```

Les modificateurs lient plus fort (avec une précédence supérieure) que l'opérateur virgule. L'exemple suivant ne déclare qu'une variable au lieu de deux, car la liste qui suit le modificateur n'est pas entre parenthèses.

7. Ou l'archaïque `$Quelque'chose` qui ne devrait probablement pas être encouragé en dehors de la poésie Perl.

```
my $toto, $titi = 1;              # FAUX
```

Cela a le même effet que :

```
my $toto;
$titi = 1;
```

Vous serez averti de cette erreur si vous avez demandé les avertissements, avec les options de ligne de commande **-w** ou **-W**, ou de préférence avec la déclaration `use warnings` expliquée plus loin dans la section *Pragmas*.

En général, il vaut mieux déclarer une variable dans la plus petite portée nécessaire. Comme les variables déclarées dans des instructions de contrôle de flux ne sont visibles que dans le bloc concerné par cette instruction, leur visibilité est réduite. Cela se lit également mieux en anglais (ce qui a moins d'intérêt pour ceux qui codent en français).

```
sub verifie_stock {
    for my $machin (our @Inventaire) {
        print "J'ai un $machin en stock aujourd'hui.\n";
    }
}
```

La forme de déclaration la plus fréquente est `my`, qui déclare des variables à portée lexicale dont le nom et la valeur sont stockés dans le bloc-note temporaire de la portée en cours et ne peuvent être accédés de manière globale. La déclaration `our` est très proche de cela, et fait entrer un nom lexical dans la portée en cours, tout comme `my`, mais pointe en réalité vers une variable globale à laquelle n'importe qui pourrait accéder s'il le voulait. En d'autres termes, c'est une variable globale maquillée en variable lexicale.

L'autre forme de portée, la portée dynamique, s'applique aux variables déclarées avec `local`, qui malgré l'emploi du mot « local » sont en fait des variables globales qui n'ont rien à voir avec le bloc-notes local.

Variables à portée lexicale : my

Pour vous éviter le casse-tête du suivi des variables globales, Perl fournit des variables à portée lexicale, appelées parfois *lexicales* pour faire court. Contrairement aux globales, les lexicales vous garantissent le secret de vos variables. Tant que vous ne distribuez pas des références à ces variables privées qui permettraient de les tripatouiller indirectement, vous pouvez être sûr que tous les accès possibles à ces variables privées sont restreints au code contenu dans une section limitée et aisément identifiable de de votre programme. Après tout, c'est la raison pour laquelle nous avons choisi le mot-clé `my`.

Une séquence d'instructions peut contenir des déclarations de variables à portée lexicale. De telles déclarations sont habituellement placées au début de la séquence d'instructions, mais ce n'est pas une obligation. En plus de déclarer les noms de variables à la compilation, les déclarations fonctionnent comme des instructions normales à l'exécution : chacune d'entre elles est élaborée dans la séquence d'instructions comme s'il s'agissait d'une instruction usuelle sans le modificateur :

```
my $nom = "fred";
my @affaires = ("voiture", "maison", "marteau");
my ($vehicule, $domicile, $outil) = @affaires;
```

Ces variables lexicales sont totalement cachées du monde à l'extérieur de la portée les

contenant immédiatement. Contrairement aux effets de portée dynamique de `local` (voir la section suivante), les lexicales sont cachées à tout sous-programme appelé depuis leur portée. Cela reste vrai même si le même sous-programme est appelé depuis lui-même ou ailleurs — chaque instance du sous-programme possède son propre « bloc-notes » de variables lexicales.

Contrairement aux portées de blocs, les portées de fichiers ne s'emboîtent pas ; il ne se produit pas d'« inclusion », tout au moins pas textuellement. Si vous chargez du code d'un autre fichier avec `do`, `require` ou `use`, le code contenu dans ce fichier ne pourra pas accéder à vos variables lexicales, tout comme vous ne pourrez pas accéder aux siennes.

Cependant toute portée à l'intérieur d'un fichier (ou même le fichier lui-même) fait l'affaire. Il est souvent utile d'avoir des portées plus larges que la définition de sous-programme, car cela vous permet de de partager des variables privées avec un nombre limité de routines. C'est ainsi que vous créez des variables qu'un programmeur C appellerait « statiques » :

```
{
    my $etat = 0;

    sub on     { $etat = 1 }
    sub off    { $etat = 0 }
    sub change { $etat = !$etat }
}
```

L'opérateur `eval` *`CHAINE`* fonctionne aussi comme une portée emboîtée, puisque le code à l'intérieur de l'`eval` peut voir les variables lexicales de l'appelant (tant que leurs noms ne sont pas cachés par des déclarations identiques dans la portée définie par l'`eval` lui-même). Les routines anonymes peuvent de même accéder à n'importe quelle variable lexicale des portées les renfermant ; si elles le font, elles sont alors nommées *fermetures*.[8] En combinant ces deux notions, si un bloc `eval`ue une chaîne qui crée un sous-programme anonyme, ce sous-programme devient une fermeture avec accès complet aux lexicales de l'`eval`et du bloc, même après que le l'`eval` et le bloc se sont terminés. Voir la section *Fermetures* au chapitre 8.

La variable nouvellement déclarée (ou la valeur, dans le cas de `local`) n'apparaît pas avant la fin de l'instruction *suivant* l'instruction contenant la déclaration. Vous pourriez donc copier une variable de cette façon :

```
my $x = $x;
```

Cela initialise le nouveau `$x` interne avec la valeur courante de `$x`, que la signification de `$x` soit globale ou lexicale. (Si vous n'initialisez pas la nouvelle variable, elle démarre avec une valeur vide ou indéfinie.)

La déclaration d'une variable lexicale d'un nom donné cache toute variable lexicale du même nom déclarée précédemment. Elle cache aussi toute variable globale non qualifiée du même nom, mais celle-ci reste toujours accessible en la qualifiant explicitement avec le nom du paquetage la contenant, par exemple `$NomPaquetage::nomvar`.

8. La véritable définition de la fermeture vient d'une notion mathématique concernant la complétude d'ensembles de valeurs et des opérateurs sur ces valeurs.

Déclarations de globales à portée lexicale : our

La déclaration `our` est une meilleure manière d'accéder aux globales, en particulier pour les programmes et les modules tournant avec la déclaration `use strict`. Cette déclaration est à portée lexicale dans le sens où elle ne s'applique que jusqu'à la fin de la portée courante. Mais contrairement au `my` à portée lexicale ou au `local` à portée dynamique, `our` n'isole rien dans la portée lexicale ou dynamique en cours. En fait, il donne accès à une variable globale dans le paquetage en cours, en cachant les lexicales de même nom qui vous auraient sinon empêché de voir cette globale. À cet égard, nos variables `our` fonctionnent tout comme mes variables `my`.

Si vous placez une déclaration `our` en dehors de tout bloc délimité par des accolades, elle dure jusqu'à la fin de l'unité de compilation en cours. Souvent, on la place juste au début de la définition d'un sous-programme pour indiquer qu'il accède à une variable globale :

```
sub verifie_entrepot {
    our @Inventaire_Courant;
    my  $machin;
    foreach $machin (@Inventaire_Courant) {
        print "J'ai un $machin en stock aujourd'hui.\n";
    }
}
```

Comme les variables globales ont une durée de vie plus longue et une visibilité plus large que les variables privées, nous préférons utiliser pour elles des noms plus longs et plus voyants que pour les variables temporaires. Cette simple habitude, si elle est suivie consciencieusement, peut faire autant que `use strict` pour décourager l'emploi de variables globales, particulièrement chez ceux qui ne sont pas des virtuoses du clavier.

Des déclarations `our` répétées ne s'emboîtent pas clairement. Chaque `my` emboîté produit une nouvelle variable, et chaque `local` emboîté produit une nouvelle valeur. Mais à chaque fois que vous employez `our`, vous mentionnez *la même* variable globale, sans notion d'emboîtement. Quand vous affectez à une variable `our`, les effets de cette affectation persistent une fois hors de portée de la déclaration. C'est parce que `our` ne crée jamais de valeur ; il donne seulement une forme d'accès limité à la globale qui, elle, existe pour toujours :

```
our $NOM_PROGRAMME = "client";
{
    our $NOM_PROGRAMME = "serveur";
    # Le code appelé d'ici voit "serveur".
    ...
}
# Le code exécuté ici voit toujours "serveur".
```

Comparez ceci avec ce qui arrive avec `my` ou `local`, quand la variable ou la valeur redevient visible après le bloc :

```
my $i = 10;
{
    my $i = 99;
    ...
}
```

```
# Le code compilé ici voit la variable externe.

local $NOM_PROGRAMME = "client";
{
    local $NOM_PROGRAMME = "serveur";
    # Le code appelé d'ici voit "serveur".
    ...
}
# Le code exécuté ici voit "client" de nouveau.
```

Faire une déclaration `our` n'a en général de sens qu'une seule fois, probablement tout au début de votre programme ou module ou, plus rarement, quand vous préfixez le `our` de son propre `local` :

```
{
    local our @Inventaire_Courant = qw(bananes);
    verifie_entrepot();  # non, nous n'avons pas de bananes :-)
}
```

Variables à portée dynamique : local

L'utilisation de l'opérateur `local` sur une variable globale lui donne une valeur temporaire à chaque fois que `local` est exécuté, mais n'affecte pas la visibilité globale de cette variable. Quand le programme atteint la fin de cette portée dynamique, cette valeur temporaire est jetée et la valeur précédente restaurée. Mais pendant que ce bloc s'exécute, c'est toujours une variable globale qui se trouve juste contenir une valeur temporaire. Si vous appelez une autre fonction pendant que votre variable globale contient la valeur temporaire et que cette fonction accède à cette variable globale, elle verra la valeur temporaire et pas la valeur initiale. En d'autres termes, cette autre fonction est dans votre portée dynamique, même si elle n'est probablement pas dans votre portée lexicale.[9]

Si vous avez un `local` qui ressemble à ceci :

```
{
    local $var = $nouveau;
    ma_fonction();
    ...
}
```

vous pouvez le voir entièrement en termes d'affectations à l'exécution :

```
{
    $ancien = $var;
    local $var = $nouveau;
    ma_fonction();
    ...
}
```

9. C'est pourquoi on appelle parfois la portée lexicale une *portée statique* : pour contraster avec la portée dynamique et insister sur le fait qu'elle est déterminée à la compilation. Ne confondez pas cette utilisation avec l'emploi du mot-clé `static` en C ou en C++. Le terme est très chargé, c'est pourquoi nous l'évitons.

```
continue {
    $var = $ancien;
}
```

La différence est qu'avec `local`, la valeur est restaurée quelle que soit la manière dont vous quittez le bloc, même si vous sortez de cette portée en faisant un `return`prématuré. La variable est toujours la même variable globale, mais la valeur qui s'y trouve dépend de la portée d'où la fonction a été appelée. C'est pourquoi on l'appelle *portée dynamique* : parce qu'elle change au cours de l'exécution.

Comme avec `my`, vous pouvez initialiser un `local` avec une copie de la même variable globale. Toutes les modifications faites à cette variable pendant l'exécution du sous-programme (et de tous les autres appelés depuis celui-ci, qui peuvent bien sûr voir cette globale à portée dynamique) seront perdues quand la routine se terminera. Vous devriez sûrement commenter ce que vous faites :

```
# ATTENTION : les modifications sont temporaires
# pour cette portée dynamique
local $Ma_Globale = $Ma_Globale;
```

Une variable globale est donc toujours visible dans l'intégralité de votre programme, qu'elle ait été déclarée avec `our`, qu'elle ait été créée à la volée ou qu'elle contienne une valeur `local`e destinée à être jetée une fois hors de portée. Ce n'est pas compliqué pour de petits programmes. Mais vous allez rapidement ne plus retrouver où sont utilisées ces variables globales dans de plus gros programmes. Si vous voulez, vous pouvez interdire l'utilisation accidentelle de variables globales à l'aide du pragma `use strict 'vars'`, décrit à la section suivante.

Bien que `my` et `local` confèrent toutes deux un certain niveau de protection, vous devriez largement préférer `my` à `local`. Cependant, vous devrez de temps en temps utiliser `local`pour pouvoir modifier temporairement la valeur d'une variable globale existante, comme celles décrites au chapitre 28, *Noms spéciaux*. Seuls les identificateurs alphanumériques peuvent avoir une portée lexicale et beaucoup de ces variables spéciales ne sont pas strictement alphanumériques. Vous aurez aussi besoin de `local` pour faire des modifications temporaires à la table des symboles d'un paquetage, comme c'est décrit dans la section *Tables de symboles* au chapitre 10. Enfin, vous pouvez aussi utiliser `local` sur un élément isolé ou tout une tranche d'un tableau ou d'un hachage. Cela fonctionne même si le tableau ou le hachage est en fait une variable lexicale, en rajoutant la couche de comportement de portée dynamique de `local` au dessus de ces variables lexicales. Nous ne parlerons pas plus de la sémantique de `local` ici. Pour plus d'informations, voir `local` au chapitre 29.

Pragmas

De nombreux langages de programmation vous permettent de donner des directives au compilateur. En Perl ces directives sont passées au compilateur par la déclaration `use`. Certains de ces pragmas sont :

```
use warnings;
use strict;
use integer;
use bytes;
use constant pi => ( 4 * atan2(1,1) );
```

Les pragmas de Perl sont décrits au chapitre 31, *Modules de pragmas*, mais nous allons tout de suite parler des plus utiles par rapport au contenu de ce chapitre.

Bien que certains soient des déclarations qui affectent les variables globales ou le paquetage en cours, la plupart des pragmas sont des déclarations à portée lexicale dont les effets ne durent que jusqu'à la fin du bloc, du fichier ou de l'`eval` qui les contient (en fonction du premier qui se présente). Un pragma à portée lexicale peut être annulé dans une portée incluse avec une déclaration `no`, qui fonctionne exactement comme `use`, mais à l'envers.

Contrôle des avertissements

Pour vous montrer comment tout cela fonctionne, nous allons manipuler le pragma `warnings` pour dire à Perl s'il doit nous avertir de pratiques discutables :

```
use warnings;       # Permet les avertissements d'ici à la fin de fichier
...
{
    no warnings;    # Désactive les avertissements dans le bloc
    ...
}
# Les avertissements sont automatiquement réactivés ici
```

Une fois les avertissements demandés, Perl vous signalera les variables utilisées une seule fois, les déclarations qui cachent d'autres déclarations dans la même portée, les conversions incorrectes de chaînes en nombres, l'utilisation de valeurs indéfinies comme des nombres ou des chaînes normaux, les tentatives d'écriture sur des fichiers ouverts en lecture seule (ou pas ouverts du tout) et bien d'autres problèmes potentiels listés au chapitre 33, *Messages de diagnostic*.

La méthode de contrôle des avertissements recommandée est l'utilisation de `use warnings`. Les anciens programmes ne pouvaient utiliser que l'option de ligne de commande `-w` ou bien modifier la variable globale `$^W` :

```
{
    local $^W = 0;
    ...
}
```

Il vaut beaucoup mieux utiliser les pragmas `use warnings` et `no warnings`. Un pragma vaut mieux car il se produit à la compilation, parce que c'est une déclaration lexicale et ne peut donc pas affecter du code qu'il n'était pas censé affecter, et (bien que nous ne vous l'ayons pas montré dans ces simples exemples) qu'il permet un contrôle plus fin sur plusieurs ensembles de classes. Pour en savoir plus sur le pragma `warnings`, y compris comment transformer des avertissements qui font juste un peu de bruit en erreurs fatales et comment supplanter le pragma pour activer les avertissements même si le module ne veut pas, voir `use warnings` au chapitre 31.

Contrôle de l'utilisation des globales

Une autre déclaration communément rencontrée est le pragma `use strict`, qui a plusieurs fonctions dont celle de contrôler l'utilisation des variables globales. Normalement, Perl vous laisse créer de nouvelles variables (ou trop souvent, écraser d'anciennes

variables) simplement en les citant. Aucune déclaration de variable n'est nécessaire (par défaut). Sachant que l'utilisation débridée de globales peut rendre les gros programmes ou les gros modules pénibles à maintenir, vous pourrez vouloir décourager leur utilisation accidentelle. Pour prévenir de tels accidents, vous pouvez écrire :

```
use strict 'vars';
```

Cela signifie que toute variable mentionnée à partir d'ici jusqu'à la fin de la portée courante doit faire référence soit à une variable lexicale, soit à une variable globale explicitement autorisée. Si ce n'est pas le cas, une erreur de compilation se produit. Une variable globale est explicitement autorisée si l'une de ces propositions est vraie :

- C'est l'une des variables spéciales de Perl (voir le chapitre 28).
- Elle est complètement définie avec son nom de paquetage (voir le chapitre 10).
- Elle a été importée dans le paquetage courant (voir le chapitre 11).
- Elle se fait passer pour une variable lexicale à l'aide d'une déclaration `our`. (C'est la raison principale pour laquelle nous avons ajouté la déclaration `our` à Perl.)

Bien sûr, il reste toujours la cinquième possibilité. Si le pragma est trop exigeant, annulez-le simplement dans un bloc intéreur avec :

```
no strict 'vars'
```

Avec ce pragma vous pouvez aussi demander une vérification stricte des déréférencements symboliques et de l'utilisation des mot simples. Habituellement, les gens tapent juste :

```
use strict;
```

pour mettre en œuvre les trois restrictions. Pour plus d'informations, voir l'entrée `use strict`, au chapitre 31.

5

Recherche de motif

Le support intégré de Perl pour la recherche de motif vous permet de rechercher dans de grandes quantités de données simplement et efficacement. Que vous fassiez tourner un énorme site portail commercial balayant tous les groupes de news existants à la recherche de potins intéressants, un organisme public occupé à comprendre la démographie humaine (ou le génome humain), une institution scolaire désireuse de mettre des informations dynamiques sur votre site web, Perl est l'outil qu'il vous faut ; d'une part à cause de ses liens avec les bases de données, mais surtout à cause de ses capacités de recherche de motif. Si vous prenez le mot « texte » dans son sens le plus large, près de 90% de tout ce vous faites est du traitement de texte. C'est vraiment ce pour quoi Perl est fait et a toujours été fait — en fait, cela fait même partie de son nom : Practical *Extraction* and Report Language. Les motifs de Perl fournissent de puissants moyens pour ratisser des montagnes de données brutes et en extraire de l'information utile.

Vous spécifiez un motif en créant une *expression rationnelle* (ou expression régulière[1] ou regex) et le moteur d'expressions régulières de Perl (le « Moteur », pour le reste de ce chapitre) prend cette expression et détermine si (et comment) le motif correspond à vos données. Alors que la plupart de vos données seront probablement des chaînes de caractères, rien ne vous empêche de vous servir des regex pour rechercher et remplacer n'importe quelle séquence de bits, y compris ce que vous auriez cru être des données « binaires ». Pour Perl, les octets sont juste des caractères qui ont une valeur ordinale inférieure à 256. (Pour en savoir plus sur ce sujet, voir le chapitre 15, *Unicode*.)

Si vous connaissiez déjà les expressions régulières avec d'autres outils, nous devons vous prévenir que celles de Perl sont un peu différentes. Premièrement elles ne sont pas vraiment « régulières » (ou « rationnelles ») au sens théorique du mot, ce qui signifie qu'elles peuvent faire beaucoup plus que les expressions rationnelles qu'on apprend en

1. NdT : *Regular expression* a deux traductions en français : « expression rationnelle », qui est un terme plutôt académique, et « expression régulière », qui ressemble fort au terme utilisé par les anglophones. Dans ce livre nous utiliserons plutôt le second (et son abréviation anglaise « regex »), mais il est possible qu'un peu de rationalité se glisse ici ou là...

cours d'informatique. Deuxièmement, elles sont tellement utilisées en Perl, qu'elles ont leurs propres variables spéciales et leur propres conventions de citation qui sont étroitement intégrées au langage, pas vaguement liées comme n'importe quelle autre librairie. Les nouveaux programmeurs Perl cherchent souvent en vain des fonctions comme :

```
match( $chaine, $motif );
subst( $chaine, $motif, $remplacement );
```

Mais la recherche et la substitution sont deux tâches si fondamentales en Perl qu'elle méritent leurs propres opérateurs d'une lettre : `m/`*`MOTIF`*`/` et `s/`*`MOTIF`*`/`*`REMPLACEMENT`*`/`. Ils sont non seulement brefs syntaxiquement, mais ils sont aussi analysés comme des chaînes entre apostrophes doubles plutôt que comme des opérateurs ordinaires ; toutefois, ils fonctionnent comme des opérateurs, c'est pourquoi nous les appelons ainsi. Tout au long de ce chapitre, vous les verrez utilisés pour comparer des motifs à des chaînes. Si une partie de la chaîne correspond au motif, nous disons que la correspondance (ou recherche) est réussie. Il y a plein de trucs sympas à faire avec des correspondances réussies. En particulier, si vous utilisez `s///`, une correspondance réussie provoque le remplacement de la partie correspondante dans la chaîne parce que vous avez spécifié comme *`REMPLACEMENT`*.

Tout ce chapitre concerne la construction et l'utilisation de motifs. Les expressions régulières de Perl sont puissantes, et concentrent beaucoup de sens en peu de volume. Elles peuvent donc vous intimider si vous essayez de saisir le sens d'un long motif d'un seul coup. Mais si vous pouvez le découper en petits morceaux et que vous savez comment le Moteur interprète ces morceaux, vous pouvez comprendre n'importe quelle expression régulière. Il n'est pas rare de voir une centaine de lignes de code C ou Java exprimées en une expression régulière d'une ligne en Perl. Cette expression régulière est peut-être un peu plus difficile à comprendre que n'importe quelle ligne du gros programme ; d'un autre côté, la regex sera beaucoup plus facile à comprendre que le plus long programme pris dans son ensemble. Il vous faut juste garder tout cela en perspective.

Bestiaire des expressions régulières

Avant de nous plonger dans les règles d'interprétation des expressions régulières, regardons à quoi certains motifs ressemblent. La plupart des caractères d'une expression régulière se correspondent à eux-mêmes. Si vous enchaînez plusieurs caractères à la suite, ils se correspondent dans l'ordre, comme on s'y attend. Donc si vous écrivez la recherche suivante :

```
/Frodon/
```

vous pouvez être sûr que le motif ne correspondra que si la chaîne contient quelque part la sous-chaîne « `Frodon` ». (Une *sous-chaîne* est juste un morceau de chaîne.) La correspondance peut se faire n'importe où dans la chaîne, tant que ces six caractères apparaissent quelque part, l'un après l'autre et dans cet ordre.

D'autres caractères ne se correspondent pas à eux-mêmes, mais se comportent d'une étrange manière. Nous les appelons *métacaractères*. (Tous les métacaractères sont des coquins, mais certains sont si mauvais, qu'ils conduisent les caractères voisins à se comporter aussi mal qu'eux.)

Voici les scélérats :

```
\ | ( ) [ { ^ $ * + ? .
```

Les métacaractères sont en fait très utiles et ont une signification spéciale à l'intérieur des motifs. Nous vous expliquerons toutes ces significations au fur et à mesure de ce chapitre. Mais sachez d'abord que vous pourrez toujours détecter n'importe lequel de ces douze caractères en le faisant précéder d'un antislash. Par exemple, comme l'antislash est un métacaractère, pour détecter un antislash, vous devrez l'antislasher : `\\`.

Vous voyez, l'antislash est le genre de caractère qui pousse les autres caractères à mal se conduire. Finalement, quand vous faites mal se conduire un métacaractère indiscipliné, il se conduit bien — un peu comme une double négation. Antislasher un caractère pour le prendre littéralement marche, mais seulement sur les caractères de ponctuation ; antislasher un caractère alphanumérique (qui habituellement se comporte correctement) fait le contraire : cela transforme le caractère littéral en quelque chose de spécial. Dès que vous voyez une telle séquence de deux caractères :

```
\b \D \t \3 \s
```

vous saurez que la séquence est un *métasymbole* qui correspond à quelque chose d'étrange. Par exemple, `\b` correspond à une limite de mot, tandis que `\t` correspond à un caractère de tabulation ordinaire. Remarquez qu'une tabulation fait un caractère de large, alors que la limite de mot a une largeur de zéro caractère, puisque c'est un point entre deux caractères. Nous appellerons donc `\b` une assertion de *largeur nulle*. Cependant, `\t` et `\b` se ressemblent en ce qu'ils supposent quelque chose sur une caractéristique de la chaîne. À chaque fois que vous faites une *assertion* au sujet de quelque chose dans une expression régulière, vous demandez à ce que quelque chose en particulier soit vrai pour que le motif corresponde.

La plupart des éléments d'une expression régulière sont des assertions, y compris les caractères ordinaires qui veulent se correspondre à eux-mêmes. Pour être plus précis, ils supposent aussi que le *prochain* élément sera en correspondance un caractère plus loin dans la chaîne, c'est pourquoi nous disons que la tabulation est de « largeur un caractère ». Certaines assertions (comme `\t`) consomment un peu de la chaîne au fur et à mesure qu'ils entrent en correspondance et d'autres (comme `\b`) non. Mais nous réservons en général le terme « assertion » pour les assertions de largeur nulle. Pour éviter les confusions, nous appellerons celles qui ont une largeur des *atomes*. (Si vous êtes physicien, vous pouvez voir les atomes de largeur non nulle comme des particules massives, à la différence des assertions, qui n'ont pas de masse comme les photons.)

Vous allez voir aussi des métacaractères qui ne sont pas des assertions ; ils sont plutôt structurels (tout comme les accolades et les points-virgules définissent la structure du code Perl, mais ne font pas vraiment quelque chose). Ces métacaractères structurels sont d'une certaine manière les plus importants, car le premier pas dans votre apprentissage de la lecture des expressions régulières est d'habituer votre regard à reconnaître les métacaractères structurels. Une fois que vous avez appris cela, lire des expressions régulières est un jeu d'enfant[2].

2. D'enfant précoce parfois, mais pas besoin d'être un futur Prix Nobel.

L'un de ces métacaractères structurels est la barre verticale, qui indique un *choix* :

```
/Frodon|Pippin|Merry|Sam/
```

Cela signifie que n'importe laquelle de ces chaînes peut correspondre. Ceci est traité dans la section *Alternative* plus loin dans ce chapitre. Dans la partie *Capture et regroupement*, nous vous montrerons comment utiliser les parenthèses autour de morceaux de votre motif pour faire des *regroupements* :

```
/(Frodon|Drogon|Bilbon) Sacquet/
```

ou même :

```
/(Frod|Drog|Bilb)on Sacquet/
```

Une autre chose que vous verrez sont les *quantificateurs*, qui disent combien de fois ce qui précède doit être trouvé à la queue-leu-leu. Les quantificateurs ressemblent à ceci :

```
*  +  ?  *?  {3}  {2,5}
```

Vous ne les verrez cependant jamais isolément. Les quantificateurs n'ont de sens qu'attachés à des atomes — c'est-à-dire à des assertions qui ont une largeur non nulle.[3] Les quantificateurs s'attachent à l'atome précédent seulement. Ce qui signifie en termes clairs qu'ils ne quantifient normalement qu'un seul caractère. Si vous voulez une correspondance avec trois copies de « `bar` » à la suite, vous devez regrouper les caractères individuels de « `bar` » dans une seule « molécule » avec des parenthèses, comme ceci :

```
/(bar){3}/
```

Cela correspondra avec « `barbarbar` ». Si vous aviez écrit `/bar{3}/`, cela aurait correspondu avec « `barrr` » — qui aurait sonné écossais mais n'aurait pas constitué un barbarbarisme. Pour en savoir plus sur les quantificateurs, voir plus loin la section *Quantificateurs*.

Maintenant que vous avez rencontré quelques-unes des bestioles qui habitent les expressions régulières, vous êtes sûrement impatient de commencer à les apprivoiser. Cependant, avant que nous ne discutions sérieusement des expressions régulières, nous allons un petit peu rebrousser chemin et parler des opérateurs qui utilisent les expressions régulières. (Et si vous apercevez quelques nouvelles bestioles regex en chemin, n'oubliez pas le guide.)

Opérateurs de recherche de motifs

Zoologiquement parlant, les opérateurs de recherche de motif de Perl fonctionnent comme une sorte de cage à expressions régulières : ils les empêchent de sortir. Si nous laissions les regex errer en liberté dans le langage, Perl serait une jungle complète. Le monde a besoin de jungles bien sûr — après tout, ce sont les moteurs de la diversité biologique — mais les jungles devraient rester où elles sont. De même, bien qu'étant le

3. Les quantificateurs sont un peu comme les modificateurs d'instructions du chapitre 4, *Instructions et déclarations*, qui ne peuvent s'attacher qu'à une instruction simple. Attacher un quantificateur à une assertion de largeur nulle, cela reviendrait à essayer d'attacher un modificateur `while` à une déclaration — ces deux actions étant aussi sensées que de demander à un chimiste une livre de photons. Les chimistes ne travaillent qu'avec des atomes.

moteur de la diversité combinatoire, les expressions régulières devraient rester à l'intérieur des opérateurs de recherche de motifs où est leur place. C'est une jungle, là-dedans.

Comme si les expressions régulières n'étaient pas assez puissantes, les opérateurs `m//` et `s///` leur donnent le pouvoir (lui aussi confiné) de l'interpolation entre apostrophes doubles. Comme les motifs sont analysés comme des chaînes entre apostrophes doubles, toutes les conventions usuelles des apostrophes doubles fonctionnent, y compris l'interpolation de variables (à moins que vous n'utilisiez les apostrophes simples comme délimiteurs) et les caractères spéciaux indiqués par des séquences d'échappement avec antislash. (Voir *Caractères spécifiques* plus loin dans ce chapitre.) Celles-ci sont appliquées avant que la chaîne soit interprétée comme une expression régulière. (C'est l'un des quelques endroits de Perl où une chaîne subit un traitement en plusieurs passes.) La première passe n'est pas une interprétation entre apostrophes doubles tout à fait normale, en ce qu'elle sait ce qu'elle doit interpoler et ce qu'elle doit passer à l'analyseur d'expressions régulières. Donc par exemple, tout `$` immédiatement suivi d'une barre verticale, d'une parenthèse fermante ou de la fin de chaîne ne sera pas traité comme une interpolation de variable, mais comme la traditionnelle assertion de regex qui signifie fin-de-ligne. Donc si vous écrivez :

```
$toto = "titi";
/$toto$/;
```

la passe d'interpolation entre apostrophes doubles sait que ces deux `$` fonctionnent différemment. Elle fait l'interpolation de `$toto` puis envoie ceci à l'analyseur d'expressions régulières :

```
/titi$/;
```

Une autre conséquence de cette analyse en deux passes est que le tokener ordinaire de Perl trouve la fin de l'expression régulière en premier, tout comme s'il cherchait le délimiteur final d'une chaîne ordinaire. C'est seulement après avoir trouvé la fin de la chaîne (et fait les interpolations de variables) que le motif est traité comme une expression régulière. Entre autres choses, cela signifie que vous ne pouvez pas « cacher » le délimiteur final d'un motif à l'intérieur d'un élément de regex (comme une classe de caractères ou un commentaire de regex, que nous n'avons pas encore couvert). Perl verra le délimiteur où qu'il se trouve et terminera le motif à cet endroit.

Vous devez également savoir que l'interpolation de variables dans un motif ralentit l'analyseur de motif, car il se sent obligé de vérifier si la variable a été modifiée, au cas où il aurait à recompiler le motif (ce qui le ralentira encore plus). Voir la section *Interpolation de variables* plus loin dans ce chapitre.

L'opérateur de translittération `tr///` ne fait pas d'interpolation de variables ; il n'utilise même pas d'expressions régulières ! (En fait, il ne devrait probablement pas faire partie de ce chapitre, mais nous n'avons pas trouvé de meilleur endroit où le mettre.) Il a cependant une caractéristique commune avec `m//` et `s///` : il se lie aux variables grâce aux opérateurs `=~` et `!~`.

Les opérateurs `=~` et `!~`, décrits au chapitre 3, *Opérateurs unaires et binaires*, lient l'expression scalaire à leur gauche à l'un des trois opérateurs de type apostrophes (ou guillemets) à leur droite : `m//` pour rechercher un motif, `s///` pour substituer une chaîne à une sous-chaîne correspondant au motif et `tr///` (ou son synonyme, `y///`) pour traduire un ensemble de caractères en un autre ensemble de caractères. (Vous pouvez écrire `//` pour `m//` sans le `m` si les slash sont utilisés comme délimiteurs.) Si la partie à droite

de =~ ou !~ n'est aucune de ces trois là, elle compte toujours comme une opération m/// de recherche, mais il n'y a pas la place de mettre un seul modificateur final (voir plus loin la section *Modificateurs de motif*) et vous devrez gérer vos propres apostrophes :

```
print "correspond" if $unechaine =~ $unmotif;
```

Vraiment, il n'y a pas de raison de ne pas l'écrire explicitement :

```
print "correspond" if $unechaine =~ m/$unmotif/;
```

Quand ils sont utilisés pour une recherche de correspondance, =~ et !~ se prononcent parfois respectivement « correspond à » et « ne correspond pas à » (bien que « contient » et « ne contient pas » puissent être plus clair).

Hormis les opérateurs m// et s///, les expressions régulières apparaissent à deux autres endroits dans Perl. Le premier argument de la fonction splitest une forme spéciale de l'opérateur de correspondance spécifiant les parties à *ne pas* retourner quand on découpe une chaîne en plusieurs sous-chaînes. Voir la description de split et les exemples au chapitre 29, *Fonctions*. L'opérateur qr// (*quote regex*) spécifie également un motif à l'aide d'une regex, mais il n'essaie pas de faire une correspondance avec quoi que ce soit (contrairement à m//, qui le fait). Au lieu de cela, il retourne une forme compilée de la regex pour une utilisation ultérieure. Voir *Interpolation de variables* pour plus d'informations.

Vous appliquez l'un des opérateurs m//, s/// ou tr/// à une chaîne particulière avec l'opérateur de lien =~ (qui n'est pas un véritable opérateur, mais plutôt un indicateur du sujet de l'opération). Voici quelques exemples :

```
$meuledefoin =~ m/aiguille/          # cherche un motif simple
$meuledefoin =~  /aiguille/          # pareil

$italiano =~ s/beurre/huile d'olive/ # une substitution bonne pour la santé

$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/ # encryption simple (à casser)
```

Sans opérateur de lien, c'est $_ qui est implicitement utilisé comme "sujet" :

```
/nouvelles vies/ and        # explore $_ et (si on trouve quelque chose)
    /autres civilisations/  #     au mépris du danger, explore encore $_

s/sucre/aspartame/          # substitue un substitut dans $_

tr/ATCG/TAGC/               # complémente le brin d'ADN dans $_
```

Comme s/// et tr/// modifient le scalaire auquel ils sont appliqués, vous ne pouvez les utiliser qu'avec des *lvalues* valides :

```
"onshore" =~ s/on/off/;     # FAUX : erreur à la compilation
```

En revanche, m// fonctionne sur le résultat de n'importe quelle expression scalaire :

```
if ((lc $chapeau_magique->contenu->comme_chaine) =~ /lapin/) {
    print "Euh, quoi d'neuf, docteur ?\n";
}
else {
    print "Ce tour ne marche jamais !\n";
}
```

Mais vous devez être un petit peu prudent, car =~ et !~ ont une précédence assez élevée (dans notre exemple précédent les parenthèses sont nécessaires autour du terme de gau-

che).[4] L'opérateur de lien `!~` fonctionne comme `=~`, mais inverse la logique du résultat de l'opération :

```
if ($romance !~ /parole/) {
    print qq/"$romance" semble être une romance sans parole.\n/;
}
```

Comme `m//`, `s///` sont des opérateurs apostrophes (ou guillemets), vous pouvez choisir vos délimiteurs. Ils fonctionnent de la même façon que les opérateurs de citation `q//`, `qq//`, `qr//`, et `qw//` (voir la section *Choisissez vos délimiteurs*, chapitre 2, *Composants de Perl*).

```
$path =~ s#/tmp#/var/tmp/scratch#;

if ($dir =~ m[/bin]) {
    print "Pas de répertoires de binaires, s'il vous plait.\n";
}
```

Quand vous utilisez des délimiteurs appariés avec `s///` ou `tr///`, si la première partie utilise l'une des quatre paires usuelles d'encadrement (parenthèses, crochets, accolades ou chevrons), vous pouvez choisir des délimiteurs différents des premiers pour la seconde partie :

```
s(oeuf)<larve>;
s{larve}{chrysalide};
s[chrysalide]/imago/;
```

Les blancs sont autorisés avant les délimiteurs ouvrants :

```
s (oeuf)       <larve>;
s {larve}      {chrysalide};
s [chrysalide] /imago/;
```

À chaque fois qu'un motif correspond avec succès (y compris les motifs de substitution), il affecte aux variables `$`` , `$&` et `$'` le texte à gauche de la correspondance, le texte correspondant et le texte à droite de la correspondance. C'est utile pour séparer les chaînes en leurs éléments :

```
"pain au chocolat" =~ /au/;
print "Trouvé :         <$`> $& <$'>\n"; # Trouvé : <pain > au < chocolat>
print "Gauche :         <$`>\n";         # Gauche :         <pain >
print "Correspondance : <$&>\n";         # Correspondance : <au>
print "Droite :         <$'>\n";         # Droite :         < chocolat>
```

Pour plus d'efficacité et de contrôle, utilisez des parenthèses pour capturer les portions spécifiques que vous voulez conserver. Chaque paire de parenthèses capture la sous-chaîne correspondant au *sous-motif entre parenthèses*. Les paires de parenthèses sont numérotées de gauche à droite par la position de chaque parenthèse ouvrante. Une fois la correspondance établie, les sous-chaînes correspondantes sont disponibles dans les variables numérotées `$1`, `$2`, `$3` et ainsi de suite :[5]

4. Sans les parenthèses, le `lc` de moindre précédence se serait appliqué à toute la correspondance de motif plutôt qu'au seul appel de méthode sur le chapeau de magicien.

5. Pas `$0` cependant, qui contient le nom de votre programme.

```
$_ = "Bilbon Sacquet est né le 22 Septembre";
/(.*) est né le (.*)/;
print "Personne: $1\n";
print "Date: $2\n";
```

$`, $&, $' et les variables numérotées sont implicitement localisées dans la portée dynamique les contenant. Elles durent jusqu'à la prochaine recherche réussie ou jusqu'à la fin de la portée courante, selon ce qui arrive en premier. Nous en reparlerons plus tard, dans un cadre différent.

Une fois que Perl a détecté que vous aurez besoin de l'une des variables $`, $& ou $' quelque part dans votre programme, il les fournira pour chaque correspondance. Cela va ralentir un peu votre programme. Comme Perl utilise un mécanisme semblable pour produire $1, $2et les autres, pour chaque motif contenant des parenthèses de capture vous payez un peu en performance. (Voir *Regroupement* pour éviter le coût de la capture en conservant la possibilité de regrouper.) Si vous n'utilisez jamais $`, $& ou $', alors les motifs *sans* parenthèses ne seront pas pénalisés. Si vous pouvez, il vaut donc mieux en général éviter d'utiliser $`, $& et $', en particulier dans les modules de bibliothèque. Mais si vous devez les utiliser au moins une fois (et certains algorithmes apprécient vraiment leur commodité), alors utilisez-les autant que vous voulez, car vous avez déjà payé le prix. $& n'est pas aussi coûteux que les deux autres dans les versions récentes de Perl.

Modificateurs de motif

Nous allons discuter des différents opérateurs de recherche de motif dans un moment, mais nous aimerions d'abord parler d'un de leur points communs à tous : les *modificateurs*.

Vous pouvez placer un ou plusieurs modificateurs d'une lettre immédiatement après le délimiteur final, dans n'importe quel ordre. Par souci de clarté, les modificateurs sont souvent appelés « le modificateur /o » et prononcés « le modificateur slash oh », même si le modificateur final peut être autre chose qu'un slash. (Certaines personnes disent « drapeau » ou « option » pour « modificateur ». C'est bien aussi.)

Certains modificateurs changent le comportement d'un seul opérateur, aussi les décrirons-nous en détail plus loin. D'autres changent la façon dont la regex est interprétée. Les opérateurs m//, s/// et qr//[6] acceptent tous les modificateurs suivants après le délimiteur final :

Modificateur	Signification
/i	Ignore les distinctions de casse des caractères (insensible à la casse).
/s	Fait correspondre . avec le saut de ligne et ignore la variable obsolète $*.
/m	Fait correspondre ^ et $ à côté d'un \n intérieur.
/x	Ignore (la plupart) des blancs et autorise les commentaires dans le motif.
/o	Ne compile le motif qu'une seule fois.

6. L'opérateur tr/// ne prend pas de regex, aussi ces modificateurs ne s'y appliquent pas.

Le modificateur `/i` indique de faire la correspondance à la fois en majuscules et en minuscules (et en casse de titre en Unicode). Ainsi `/perl/i` correspondrait avec « `SUPERLATIF` » ou « `Perlimpinpin` » (entre autres choses). Le pragma `use locale` peut aussi avoir une influence sur ce qui est considéré comme équivalent. (Cela peut avoir une mauvaise influence sur les chaînes contenant de l'Unicode.)

Les modificateurs `/s` et `/m` n'impliquent rien de coquin. Ils affectent la manière dont Perl traite les correspondances avec des chaînes contenant des sauts de ligne. Ils n'indiquent pas si votre chaîne contient réellement des sauts de ligne ; mais plutôt si Perl doit *supposer* que votre chaîne contient une seule ligne (`/s`) ou plusieurs lignes (`/m`), car certains métacaractères fonctionnent différemment selon qu'ils sont censés se comporter d'une manière orientée ligne ou non.

D'ordinaire le métacaractère « `.` » correspond à tout caractère *sauf* un saut de ligne, parce que sa signification traditionnelle est de correspondre aux caractères à l'intérieur d'une ligne. Avec un `/s` cependant, le métacaractère « `.` » peut aussi correspondre à des sauts de ligne, car vous avez dit à Perl d'ignorer le fait que la chaîne contient plusieurs sauts de ligne. (Le modificateur `/s` fait aussi ignorer à Perl la variable obsolète `$*`, dont nous espérons que vous l'ignoriez aussi.) Le modificateur `/m`, de son côté, change l'interprétation des métacaractères `^` et `$` en leur permettant de correspondre à côté de sauts de lignes à l'intérieur de la chaîne au lieu de considérer seulement les extrémités de la chaîne. Voir les exemples dans la section *Positions* plus loin dans ce chapitre.

Le modificateur `/o` contrôle la recompilation des motifs. Sauf si les délimiteurs sont des apostrophes simples, (`m'MOTIF'`, `s'MOTIF'REMPLACEMENT'`, ou `qr'MOTIF'`), toute variable dans le motif sera interpolée (et pourra provoquer la recompilation du motif) à chaque fois que l'opérateur de motif sera évalué. Si vous voulez qu'un motif ne soit compilé qu'une fois et une seule, servez-vous du modificateur `/o`. Cela empêche une recompilation coûteuse à l'exécution ; cela peut servir quand la valeur interpolée ne change pas au cours de l'exécution. Cependant, utiliser `/o` revient à faire la promesse de ne pas modifier les variables dans le motif. Si vous les modifiez, Perl ne s'en rendra même pas compte. Pour avoir un meilleur contrôle de la recompilation des motifs, utilisez l'opérateur de citation de regex `qr//`. Pour plus de détails, voir la section *Interpolation de variables* plus loin dans ce chapitre.

Le modificateur `/x` est l'*ex*pressif : il vous permet d'*ex*ploiter les blancs et les commentaires *ex*plicatifs pour *ex*alter la lisibilité de votre motif, ou même lui permettre d'*ex*plorer au-delà des limites des lignes.

Euh, c'est-à-dire que `/x` modifie la signification des blancs (et du caractère `#`) : au lieu de les laisser se correspondre à eux-mêmes comme le font les caractères ordinaires, il les transforme en métacaractères qui, étrangement, se comportent comme les blancs (et les caractères de commentaires) devraient le faire. `/x` permet donc l'utilisation d'espaces, de tabulations et de sauts de lignes pour le formatage, exactement comme le code Perl habituel. Il permet également l'utilisation du caractère `#`, qui n'est normalement pas un caractère spécial dans un motif, pour introduire un commentaire qui s'étend jusqu'au bout de la ligne en cours à l'intérieur de la chaîne de motif.[7] Si vous voulez détecter un véritable espace (ou le caractère `#`), alors vous devrez le mettre dans une classe de caractères, le protéger avec un antislash ou bien l'encoder en octal ou en hexadécimal. (Mais les blancs sont normalement détectés avec une séquence `\s*` ou `\s+`, donc la situation ne se rencontre pas souvent en pratique.)

À elles toutes, ces fonctionnalités font beaucoup pour rendre les expressions régulières plus proches d'un langage lisible. Dans l'esprit de TMTOWTDI, il existe donc plus d'une manière d'écrire une même expression rationnelle. En fait, il y a plus de deux manières :

```
m/\w+:(\s+\w+)\s*\d+/;       # Mot, deux-points, espace, mot, espace,
chiffres.

m/\w+: (\s+ \w+) \s* \d+/x;  # Mot, deux-points, espace, mot, espace,
chiffres.

m{
    \w+:                     # Détecte un mot et un deux-points.
    (                        # (début de groupe)
         \s+                 # Détecte un ou plusieurs blancs.
         \w+                 # Détecte un autre mot.
    )                        # (fin de groupe)
    \s*                      # Détecte zéro ou plusieurs blancs.
    \d+                      # Détecte des chiffres.
}x;
```

Nous expliquerons ces nouveaux métasymboles plus loin dans ce chapitre. (Cette section était supposée décrire les modificateurs de motifs, mais nous l'avons laisser dériver dans notre *ex*citation au sujet de `/x`. Bref.) Voici une expression régulière qui trouve les mots doublés dans un paragraphe, emprunté à *Perl en action* (*The Perl Cookbook*). Il utilise les modificateurs `/x` et `/i`, ainsi que le modificateur `/g` décrit plus loin.

```
# Trouve les doublons dans les paragraphes, si possible malgré les
# enjambements.
# Utilise /x pour les espaces et les commentaires, / pour détecter les
# deux 'tout' dans "Tout tout va bien ?", et /g pour trouver tous les
# doublons.
$/ = "";        # mode "paragrep"
while (<>) {
    while ( m{
                \b            # commence à une limite de mot
                (\w\S+)       # trouve un morceau de "mot"
                (
                    \s+       # séparé par des blancs
                    \1        # et ce même morceau
                ) +           # ad lib
                \b            # jusqu'à une autre limite de mot
            }xig
          )
    {
        print "doublon '$1' au paragraphe $.\n";
    }
}
```

7. Faites attention cependant à ne pas inclure le délimiteur de motif dans le commentaire. À cause de sa règle "trouve la fin d'abord", Perl n'a aucun moyen de savoir que vous n'aviez pas l'intention de finir le motif à cet endroit.

Quand on l'exécute sur ce chapitre, il affiche des avertissements tels que :

```
doublon 'nous' au paragraphe 36
```

Bien sûr, nous nous sommes aperçus que dans ce cas particulier c'est normal.

L'opérateur m// (match)

```
EXPR =~ m/MOTIF/cgimosx
EXPR =~ /MOTIF/cgimosx
EXPR =~ ?MOTIF?cgimosx
m/MOTIF/cgimosx
/MOTIF/cgimosx
?MOTIF?cgimosx
```

L'opérateur `m//` recherche dans la chaîne contenue dans le scalaire *`EXPR`* le motif *`MOTIF`*. Si le délimiteur est `/`ou `?`, le `m` initial est facultatif. `?` et `'` ont tous deux une signification particulière en tant que délimiteurs : le premier fait une détection à usage unique, le second supprime l'interpolation de variables et les six séquences de traduction (`\U` et autres, décrits plus loin).

Si l'évaluation de *`MOTIF`* conduit à une chaîne vide, soit parce que vous l'avez spécifié ainsi en utilisant `//` ou parce qu'une variable interpolée a donné la chaîne vide, la dernière expression régulière exécutée avec succès sans être cachée à l'intérieur d'un bloc inclus (ou à l'intérieur d'un `split`, `grep` ou `map`) est exécutée à la place.

En contexte scalaire, l'opérateur renvoie vrai (`1`) en cas de succès, et faux (`""`) sinon. Cette forme se rencontre habituellement en contexte booléen :

```
if ($comte =~ m/Sacquet/) { ... }  # recherche Sacquet dans la $comté
if ($comte =~ /Sacquet/)  { ... }  # recherche Sacquet dans la $comté

if ( m#Sacquet# )         { ... }  # cherche ici dans $_
if ( /Sacquet/ )          { ... }  # cherche ici dans $_
```

Utilisé en contexte de liste, `m//` renvoie la liste des sous-chaînes détectées par les parenthèses de capture du motif (c'est-à-dire `$1`, `$2`, `$3` et ainsi de suite), comme cela est décrit plus loin dans la section *Capture et regroupement*. Les variables numérotées sont modifiées malgré tout, même si la liste est retournée. Si la recherche réussit en contexte de liste mais qu'il n'y avait pas de parenthèses de capture (ni de `/g`), une liste consistant en `(1)` est retournée. Comme elle renvoie la liste vide en cas d'échec, cette forme de `m//` peut aussi être utilisée en contexte booléen, mais seulement quand elle y participe indirectement *via* une affectation de liste :

```
if (($cle,$valeur) = /(\w+): (.*)/) { ... }
```

Les modificateurs valides pour `m//` (sous toutes ses formes) sont listés dans le *tableau 5-1*.

Les cinq premiers modificateurs s'appliquent à la regex et ont été décrits précédemment. Les deux derniers modifient le comportement de l'opérateur lui-même. Le modificateur `/g` indique une détection globale — c'est-à-dire que l'on recherche autant de fois qu'il est possible à l'intérieur de la chaîne. La façon dont cela se comporte dépend du contexte. En contexte de liste, `m//g` renvoie la liste de toutes les occurrences trouvées. Dans l'exemple suivant nous trouvons toutes les endroits où quelqu'un a mentionné « `perl` », « `Perl` », « `PERL` », et ainsi de suite :

Tableau 5-1. Modificateurs de m//

Modificateur	Signification
`/i`	Ignore la casse des caractères.
`/m`	Permet à `^` et `$` de fonctionner à côté d'un `\n`.
`/s`	Permet au `.` de détecter des sauts de lignes et ignore la variable obsolète `$*`.
`/x`	Ignore (presque tous) les blancs et permet les commentaires à l'intérieur du motif.
`/o`	Compile le motif une seule fois.
`/g`	Recherche globale, pour détecter toutes les occurrences.
`/cg`	Permet de continuer la recherche après un échec dans `/g`.

```
if (@perls = $paragraphe =~ /perl/gi) {
    printf "Perl a été mentionné %d fois.\n", scalar @perls;
}
```

S'il n'y a pas de parenthèses de capture dans le motif `/g`, alors les correspondances complètes sont renvoyées. S'il y a des parenthèses de capture, seules les chaînes capturées sont retournées. Imaginez une chaîne comme celle-ci :

```
$chaine = "password=xyzzy verbose=9 score=0";
```

Imaginez également que vous souhaitez initialiser un hachage comme ceci :

```
%hash = (password => "xyzzy", verbose => 9, score => 0);
```

Sauf évidemment que vous n'avez pas une liste, vous avez une chaîne. Pour obtenir la liste correspondante, vous pouvez utiliser l'opérateur `m//g` en contexte de liste pour capturer tous les couples clé/valeur de la chaîne :

```
%hash = $chaine =~ /(\w+)=(\w+)/g;
```

La séquence `(\w+)` capture un mot alphanumérique. Voir la section *Capture et regroupement*.

Utilisé en contexte scalaire, le modificateur `/g` indique une *détection progressive*, qui reprend une nouvelle recherche sur la même chaîne à la position qui suit celle où la dernière s'est arrêtée. L'assertion `\G` représente cette position dans la chaîne. Voir la section *Positions* plus loin dans ce chapitre pour une description de `\G`. Si vous utilisez le modificateur `/c` (pour « continuer ») en plus de `/g`, alors quand le `/g` ne détecte plus rien, la détection ratée ne réinitialise pas le pointeur de position.

Si le délimiteur est `?`, comme dans `?MOTIF?`, la recherche fonctionne comme un `/MOTIF/` normal, sauf qu'elle ne trouve qu'une seule occurrence entre deux appels à l'opérateur `reset`. Cela peut être une optimisation utile, quand vous désirez détecter seulement la première occurrence du motif pendant l'exécution du programme, et pas toutes les occurrences. L'opérateur fait la recherche à chaque fois que vous l'appelez, jusqu'à ce qu'il trouve finalement quelque chose, à la suite de quoi il se bloque de lui-même, renvoyant faux jusqu'à ce que vous le réenclenchiez avec `reset`. Perl se souvient de l'état de la détection pour vous.

L'opérateur `??` est particulièrement utile quand une recherche de motif ordinaire trouverait la dernière occurrence plutôt que la première :

```
open DICT, "/usr/dict/words_fr" or die "Impossible d'ouvrir words_fr: $!\n";
while (<DICT>) {
    $premier = $1 if ?(^neur.*)?;
    $dernier = $1 if /(^neur.*)/;
}
print $premier,"\n";          # affiche "neural"
print $dernier,"\n";          # affiche "neurula"
```

L'opérateur `reset` ne réinitialisera que les instances de `??` qui ont été compilées dans le même paquetage que l'appel à `reset`. `m??` et `??` sont équivalents.

L'opérateur s/// (substitution)

```
LVALUE =~ s/MOTIF/REMPLACEMENT/egimosx
s/MOTIF/REMPLACEMENT/egimosx
```

Cet opérateur recherche *MOTIF* dans une chaîne, et s'il le détecte, remplace la chaîne correspondante par le texte de *REMPLACEMENT*. (Les modificateurs sont décrits plus loin dans cette section.)

```
$lotr = $hobbit;             # Copie juste Bilbon le hobbit
$lotr =~ s/Bilbon/Frodon/g;  # et écrit très simplement une suite.
```

La valeur de retour d'un `s///` (en contexte scalaire comme en contexte de liste) est le nombre de fois qu'il a réussit (ce qui peut être plus d'une fois s'il a été utilisé avec le modificateur `/g` décrit plus tôt). En cas d'échec, il renvoie faux (`""`), qui est numériquement équivalent à `0`.

```
if ($lotr =~ s/Bilbon/Frodon/) { print "Suite écrite avec succès." }
$changements = $lotr =~ s/Bilbon/Frodon/g;
```

La partie de remplacement est traitée comme une chaîne entre apostrophes doubles. Vous pouvez utiliser toutes les variables à portée dynamique décrites plus tôt (`$`` , `$&`, `$'`, `$1`, `$2`, et ainsi de suite) dans la chaîne de remplacement, ainsi que tous les trucs utilisables avec des apostrophes doubles. Voici un exemple qui trouve toutes les chaînes « `revision` », « `version` », ou « `release` », et les remplace chacune par son équivalent avec une majuscule, à l'aide de la séquence d'échappement `\u` dans la partie de remplacement :

```
s/revision|version|release/\u$&/g;  # Utilisez | pour dire "ou"
                                    # dans un motif
```

Toutes les variables scalaires sont interpolées en contexte d'apostrophes doubles, et pas seulement les étranges que nous venons de voir. Supposons que vous ayez un hachage `%Noms` qui fasse correspondre aux numéros de version des noms de projet interne ; par exemple `$Noms{"3.0"}` pourrait avoir le nom de code « `Isengard` ». Vous pourriez utiliser `s///` pour trouver les numéros de version et les remplacer par le nom des projets correspondants :

```
s/version ([0-9.]+)/la livraison $Noms{$1}/g;
```

Dans la chaîne de remplacement, `$1` retourne ce que la première (et unique) paire de parenthèses a capturé. (Vous auriez pu aussi utiliser `\1` comme vous l'auriez fait dans le motif, mais cet emploi est obsolète dans le remplacement. Dans une chaîne entre apostrophes doubles normale, `\1` signifie Contrôle-A.)

Si *MOTIF* est une chaîne vide, la dernière expression régulière exécutée avec succès est utilisée à la place. *MOTIF* et *REMPLACEMENT* sont tous deux sujets à l'interpolation de variables. Cependant, un *MOTIF* est interpolé à chaque évaluation du `s///` en tant que tel, tandis que le *REMPLACEMENT* n'est évalué qu'à chaque fois qu'une correspondance est trouvée. (Le *MOTIF* peut correspondre plusieurs fois en une évaluation si vous utilisez le modificateur `/g`.)

Comme précédemment, les cinq modificateurs du *tableau 5-2* modifient le comportement de la regex ; ce sont les mêmes que pour `m//` et `qr//`. Les deux derniers affectent l'opérateur de substitution lui-même.

Tableau 5-2. Modificateurs de s///

Modificateur	Signification
`/i`	Ignore la casse des caractères (lors de la correspondance).
`/m`	Permet à `^` et `$` de fonctionner à côté d'un `\n`.
`/s`	Permet au `.` de détecter des sauts de ligne et ignore la variable obsolète `$*`.
`/x`	Ignore (presque tous) les blancs et permet les commentaires à l'intérieur du motif.
`/o`	Compile le motif une seule fois.
`/g`	Replacement global, c'est-à-dire de toutes les occurrences.
`/e`	Évalue la partie droite comme une expression.

Le modificateur `/g` est utilisé avec `s///` pour remplacer chaque occurrence de *MOTIF* par la valeur *REMPLACEMENT*, et pas seulement la première rencontrée. Un opérateur `s///g` agit comme un rechercher-remplacer global, en faisant toutes les modifications en une seule fois. Tout comme un `m//g` en contexte de liste, sauf que `m//g` ne change rien. (Et que `s///g` ne fait pas de détection progressive comme le faisait un `m//g` en contexte scalaire.)

Le modificateur `/e` traite le *REMPLACEMENT* comme s'il s'agissait d'un bout de code Perl plutôt que d'une chaîne interpolée. Le résultat de l'exécution de ce code est utilisé comme chaîne de remplacement. Par exemple, `s/([0-9]+)/sprintf("%#x", $1)/ge` remplacerait `2581` par `0xb23`. Ou supposez que dans notre exemple précédent, vous n'ayez pas été sûr de disposer d'un nom pour toutes les versions et que vous ayez voulu laisser inchangées les versions sans nom. Avec un formatage un peu créatif de `/x`, vous auriez pu écrire :

```
s{
    version
    \s+
    (
        [0-9.]+
    )
}{
    $Noms{$1}
        ? "la livraison $Names{$1}"
        : $&
}xge;
```

La partie droite de votre `s///e` (ou dans ce cas, la partie basse) est vérifiée syntaxiquement et compilée en même temps que le reste de votre programme. Toute erreur de syntaxe est détectée à la compilation, et les exceptions dynamiques ne sont pas relevées. Chaque nouveau `/e` ajouté à la suite du premier (comme `/ee`, `/eee` et ainsi de suite) revient à appeler `eval` *`CHAINE`* sur le résultat du code, une fois par `/e` supplémentaire. Ceci évalue le résultat du code dans l'expression et capture les exceptions dans la variable spéciale `$@`. Voir la section *Motifs programmatiques* plus loin dans ce chapitre pour plus de détails.

Modifier les chaînes en passant

Parfois vous voulez une nouvelle chaîne, modifiée sans altérer celle sur laquelle elle est basée. Au lieu d'écrire :

```
$lotr = $hobbit;
$lotr =~ s/Bilbon/Frodon/g;
```

vous pouvez combiner ces deux instructions en une seule. À cause de la précédence, des parenthèses sont requises autour de l'affectation, tout comme dans la plupart des combinaisons appliquant `=~` à une expression.

```
($lotr = $hobbit) =~ s/Bilbon/Frodon/g;
```

Sans les parenthèses autour de l'affectation, vous n'auriez fait que changer `$hobbit` et récupérer le nombre de modifications dans `$lotr`, ce qui nous aurait donné une suite fort peu intéressante.

Vous ne pouvez pas utiliser un opérateur `s///` directement sur un tableau. Pour cela, vous aurez besoin d'une boucle. Par une heureuse coïncidence, l'idiome Perl standard pour rechercher et modifier sur chaque élément d'un tableau s'obtient grâce au système d'alias de `for`/`foreach`, combiné avec l'utilisation de `$_` comme variable par défaut de la boucle :

```
for (@chapitres) { s/Bilbon/Frodon/g } # Fait les substitutions chapitre
                                       # par chapitre.
s/Bilbon/Frodon/g for @chapitres;      # Pareil.
```

Comme avec les variables scalaires, vous pouvez également combiner substitution et affectation si vous désirez conserver une copie de l'original :

```
@anciens = ('oiseau bleu', 'bleu roi',  'bleu nuit', 'bleu de travail');
for (@nouveaux = @anciens) { s/bleu/rouge/ }
print "@nouveaux\n";  # affiche : oiseau rouge rouge roi rouge
                      # nuit rouge de travail
```

La manière idiomatique d'appliquer des substitutions répétées sur une même variable est d'utiliser une boucle à un seul tour. Par exemple, voici comment normaliser les blancs dans une variable :

```
for ($chaine) {
    s/^\s+//;       # supprime les blancs au début
    s/\s+$//;       # supprime les blancs à la fin
    s/\s+/ /g;      # minimise les blancs internes
}
```

ce qui produit exactement le même résultat que :

```
$chaine = join(" ", split " ", $chaine);
```

Vous pouvez également utiliser une telle boucle dans une affectation, comme nous l'avons fait dans le cas du tableau :

```
for ($nouveau = $ancien) {
    s/Fred/Homer/g;
    s/Wilma/Marge/g;
    s/Pebbles/Lisa/g;
    s/Dino/Bart/g;
}
```

Quand une substitution globale n'est pas assez globale

De temps en temps, vous ne pouvez tout simplement pas utiliser un `/g` pour que tous les changements soient faits, soit parce que les substitutions se produisent de droite à gauche, soit parce que la longueur de `$`` doit changer entre deux occurrences. En général, vous pouvez faire cela en appelant `s///` à répétition. Mais vous voulez tout de même que la boucle s'arrête quand le `s///` finit par échouer, ce qui ne laisse rien à faire dans la partie principale de la boucle. Alors nous écrivons un simple `1`, qui est une chose plutôt ennuyeuse à faire, mais souvent le mieux que vous pouvez espérer. Voici quelques exemples qui utilisent quelques unes des ces étranges bestioles que sont les regex :

```
# met des points aux bons endroits dans un entier
1 while s/(\d)(\d\d\d)(?!\d)/$1.$2/;

# change les tabulations pour des espacements sur 8 colonnes
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;

# supprime les parenthèses (imbriquées (ou même très imbriquées (comme ça)))
1 while s/\([^()]*\)//g;

# supprime les doublons (et les triplons (et les quadruplons...))
1 while s/\b(\w+) \1\b/$1/gi;
```

Cette dernière nécessite une boucle car sinon elle transformerait ceci :

```
Paris AU AU AU AU printemps.
```

en cela :

```
Paris AU AU printemps.
```

L'opérateur tr/// (translittération)

```
LVALUE =~ tr/LISTEDERECHERCHE/LISTEDEREMPLACEMENT/cds
tr/LISTEDERECHERCHE/LISTEDEREMPLACEMENT/cds
```

Pour les fans de *sed*, `y///` est fourni comme synonyme de `tr///`. C'est la raison pour laquelle vous ne pouvez pas plus appeler une fonction `y` que vous ne pouvez l'appeler `q` ou `m`. Hormis cela, `y///` est exactement identique à `tr///` et nous ne le mentionnerons plus.

Cet opérateur n'aurait pas vraiment de raison d'apparaître dans un chapitre consacré à la détection de motif, puisqu'il n'utilise pas de motif. Cet opérateur balaie une chaîne caractère par caractère, et remplace chaque occurrence d'un caractère de *LISTEDERECHERCHE* (qui n'est pas une expression régulière) par le caractère correspondant dans *LISTEDEREMPLACEMENT* (qui n'est pas une chaîne de remplacement). Cependant, il ressemble un peu à `m//` et à `s///`, et vous pouvez même utiliser les opérateurs de lien `=~` et `!~` dessus. C'est pourquoi nous le décrivons ici. (`qr//` et `split` sont des opérateurs de recherche de motif, mais on ne peut pas utiliser les opérateurs de lien dessus, c'est pourquoi ils sont ailleurs dans ce livre. Allez comprendre.)

La translittération retourne le nombre de caractères remplacés ou effacés. Si aucune chaîne n'est spécfiée par les opérateurs `=~` ou `!~`, c'est la chaîne `$_` qui est modifiée. *LISTEDERECHERCHE* et *LISTEDEREMPLACEMENT* peuvent définir des intervalles de caractères successifs à l'aide d'un tiret :

```
$message =~ tr/A-Za-z/N-ZA-Mn-za-m/;    # cryptage rot13.
```

Remarquez qu'un intervalle comme `A-Z` suppose un ensemble de caractères linéaire comme l'est la table ASCII. Mais chaque ensemble de caractères a son propre point de vue concernant l'ordre de ses caractères, et donc de quels caractères font partie d'un intervalle particulier. Un principe sain est d'utiliser des intervalles qui commencent et finissent sur des caractères de la même casse (`a-e`, `A-E`) ou des chiffres (`0-4`). Tout le reste est suspect. En cas de doute, décrivez l'ensemble qui vous intéresse en entier : `ABCDE`.

LISTEDERECHERCHE et *LISTEDEREMPLACEMENT* ne subissent pas d'interpolation de variable comme des chaînes entre apostrophes doubles ; vous pouvez cependant utiliser les séquences avec antislash qui correspondent à des caractères spécifiques, comme `\n` ou `\015`.

Le tableau *tableau 5-3* donne la liste des modificateurs qui s'appliquent à l'opérateur `tr///`. Ils sont complètement différents de ceux que vous appliquez à `m//`, `s///` ou `qr//`, même si certains d'entre eux y ressemblent. Si le modificateur `/c` est spécifié, le jeu de caractères dans *LISTEDERECHERCHE* est complémenté ; c'est-à-dire que la liste de recherche effective comprend tous les caractères qui ne se trouvent *pas* dans *LISTEDERECHERCHE*. Dans le cas de l'Unicode, cela peut représenter un *grand nombre* de caractères, mais comme ils sont stockés logiquement et non physiquement, vous n'avez pas à vous inquiéter d'un éventuel manque de mémoire.

Tableau 5-3. Modificateurs de tr///

Modificateur	Signification
`/c`	Complémente la *LISTEDERECHERCHE*.
`/d`	Supprime les caractères trouvés mais non remplacés.
`/s`	Concentre les caractères dupliqués remplacés.

Le modificateur `/d` transforme `tr///` en ce qu'on pourrait appeler l'opérateur de « transoblitération » : tout caractère spécifié par *LISTEDERECHERCHE* mais qui n'a pas de caractère de remplacement dans *LISTEDEREMPLACEMENT* est effacé. (C'est un peu plus souple que le comportement de certains *tr*(1), qui effacent tout ce qu'ils trouvent dans *LISTEDERECHERCHE*, point.)

Si le modificateur `/s` est spécifié, les suites de caractères qui sont convertis en un caractère identique sont réduites à un seul exemplaire de ce caractère.

Si le modificateur `/d` est utilisé, *`LISTEDEREMPLACEMENT`* est toujours interprétée comme elle est spécifiée. Sinon, si *`LISTEDEREMPLACEMENT`* est plus courte que *`LISTEDERECHERCHE`*, le dernier caractère est répliqué jusqu'à ce qu'elle soit assez longue. Si *`LISTEDEREMPLACEMENT`* est vide, la *`LISTEDERECHERCHE`* est répliquée, ce qui est étonnamment utile si vous voulez juste compter les caractères, et non les modifier. C'est aussi utile pour compacter les séquences de caractères. avec `/s`.

```
tr/aeiou/!/;                  # change toutes les voyelles en !
tr{/\\\r\n\b\f. }{_};         # change les caractères bizarres en soulignés

tr/A-Z/a-z/ for @ARGV;        # normalise en minuscules ASCII

$compte = ($para =~ tr/\n//); # compte les sauts de lignes dans $para
$compte = tr/0-9//;           # compte les chiffres dans $_

$mot =~ tr/a-zA-Z//s;         # illettrisme -> iletrisme

tr/@$%*//d;                   # supprime tous ceux là
tr#A-Za-z0-9+/##cd;           # supprime les caractères hors base64

# change en passant
($HOTE = $hote) =~ tr/a-z/A-Z/;

$pathname =~ tr/a-zA-Z/_/cs;  # change les caractères ASCII non
                              # alphabétiques en un seul souligné

tr [\200-\377]
   [\000-\177];               # supprime le 8e bit de chaque octet
```

Si le même caractère apparaît plusieurs fois dans la *`LISTEDERECHERCHE`*, seul le premier est utilisé. Ainsi, ceci :

```
tr/AAA/XYZ/
```

changera tout A en X (dans `$_`).

Bien que les variables ne soient pas interpolées par `tr///`, vous pouvez obtenir le même effet en utilisant `eval` *`EXPR`* :

```
$compte = eval "tr/$ancien/$nouveau/";
die if $@;  # propage l'exception due à un contenu de eval incorrect
```

Encore un mot : si vous voulez passer votre texte en majuscules ou en minuscules, n'utilisez pas `tr///`. Utilisez plutôt les séquences `\U` ou `\L` dans des chaînes entre apostrophes doubles (l'équivalent des fonctions `ucet` `lc`) car elles seront attentives aux informations de locales et à l'Unicode, alors que `tr/a-z/A-Z` ne le sera pas. De plus, dans les chaînes Unicode, la séquence `\u` et la fonction correspondante `ucfirst` comprennent la notion de casse de titre, ce qui pour certaines langues peut être différent de simplement convertir en majuscules.

Métacaractères et métasymboles

Maintenant que nous avons admiré les jolies cages, nous pouvons de nouveau nous intéresser aux bestioles dans ces cages, les étranges caractères que vous mettez dans les motifs. Maintenant vous avez compris que ces symboles ne sont pas du code Perl normal comme les appels de fonction ou les opérateurs arithmétiques. Les expressions régulières ont leur propre petit langage inclus dans Perl. (Chacun a son jardin et sa jungle secrète.)

Malgré leur puissance et leur expressivité, les motifs de Perl reconnaissent les mêmes 12 caractères traditionnels (les « Douze salopards », s'il en est) qu'on trouve dans de nombreux autres modules d'expressions régulières.

```
\ | ( ) [ { ^ $ * + ? .
```

Certains d'entre eux changent les règles, rendant spéciaux les caractères normaux qui les suivent. Nous n'aimons pas appeler les séquences de plus d'un caractère des « caractères », c'est pourquoi nous les appelerons *métasymboles* (ou même seulement « symboles »). Mais au plus haut niveau, ces douze métacaractères sont tout ce dont vous (et Perl) avez à vous inquiéter. Tout le reste procède de là.

Certains métacaractères simples fonctionnent par eux-mêmes, comme `.`, `^` et `$`. Ils n'affectent directement rien de ce qui les entoure. Certains métacaractères fonctionnent commes des opérateurs préfixes, contrôlant ce qui les suit, comme `\`. D'autres fonctionnent comme des opérateurs postfixes, et contrôlent ce qui les précède immédiatement, comme `*`, `+` et `?`. Un métacaractère, `|`, fonctionne comme un opérateur infixe, en se tenant entre les deux opérandes qu'il affecte. Il existe même des métacaractères de parenthèsage, qui fonctionnent comme des opérateurs « circonfixes », et régissent ce qu'ils contiennent, comme `(...)` et `[...]`. Les parenthèses sont particulièrement importantes, car elles définissent les limites de `|` à l'intérieur, et celles de `*`, `+` et `?` à l'extérieur.

Si vous n'apprenez qu'un seul des douze métacaractères, choisissez l'antislash. (Euh... et les parenthèses.) C'est parce que l'antislash invalide les autres. Quand un antislash précède un caractère non alphanumérique dans un motif Perl, il en fait toujours un caractère littéral. Si vous avez besoin de faire une correspondance littérale avec l'un des douze métacaractères, vous n'avez qu'à l'écrire précédé d'un antislash. Ainsi, `\.` est un vrai point, `\$` un vrai dollar, `\\` un vrai antislash, et ainsi de suite. On appelle cela « protéger » le métacaractère, ou simplement l'« antislasher ». (Bien sûr, vous savez déjà que l'antislash sert à supprimer l'interpolation de variable dans les chaînes entre apostrophes doubles.)

Bien qu'un antislash transforme un métacaractère en caractère littéral, il a l'effet inverse sur un caractère alphanumérique le suivant. Il prend ce qui était normal et le rend spécial. C'est-à-dire qu'à eux deux, ils forment un métasymbole. Une liste alphabétique de ces métasymboles se trouve juste en dessous dans le *tableau 5-7*.

Tables de métasymboles

Dans les tableaux qui suivent, la colonne Atomique indique « Oui » si le métasymbole correspondant est quantifiable (c'est-à-dire s'il peut correspondre à quelque chose qui a une largeur, plus ou moins). Nous avons également utilisé « `...` » pour représenter

« quelque chose d'autre ». (Consultez la discussion qui suit pour savoir ce que « `...` » signifie, si le commentaire du tableau n'est pas suffisant.)

Le *tableau 5-4* liste les métasymboles traditionnels de base. Les quatre premiers sont les métasymboles structurels déjà mentionnés, tandis que les trois suivants sont de simples métacaractères. Le métacaractère `.` est un exemple d'atome car il correspond à quelque chose qui a une largeur (la largeur d'un caractère, dans ce cas) ; ^ et $ sont des exemples d'assertions, car ils correspondent à quelque chose de largeur nulle et ne sont évalués que pour savoir s'ils sont vrais ou faux.

Tableau 5-4. Métacaractères usuels de regex

Symbole	Atomique	Signification
`\...`	Variable	Rend le caractère non alphanumérique suivant littéral, rend le caractère alphanumérique suivant méta (peut-être).
`...|...`	Non	Alternative (correspond à l'un ou l'autre).
`(...)`	Oui	Regroupement (traite comme une unité).
`[...]`	Oui	Classe de caractères (correspond à un caractère d'un ensemble).
^	Non	Vrai au début d'une chaîne (ou après toute nouvelle ligne, éventuellement).
.	Oui	Correspond à un caractère (sauf le saut de ligne, normalement).
$	Non	Vrai en fin de chaîne (ou avant toute nouvelle ligne, éventuellement).

Les quantificateurs, décrits plus loin dans leur propre section, indiquent combien de fois l'atome qui les précède (caractère simple ou regroupement) doit correspondre. Ils sont listés dans le *tableau 5-5*.

Tableau 5-5. Quantificateurs de regex

Quantificateur	Atomique	Signification
*	Non	Correspond 0 fois ou plus (maximal).
+	Non	Correspond 1 fois ou plus (maximal).
?	Non	Correspond 0 ou 1 fois (maximal).
{*COMPTE*}	Non	Correspond exactement *COMPTE* fois.
{*MIN*,}	No	Correspond au moins *MIN* fois (maximal).
{*MIN*,*MAX*}	No	Correspond au moins *MIN* fois, mais pas plus de *MAX* fois (maximal).
*?	Non	Correspond 0 fois ou plus (minimal).
+?	Non	Correspond 1 fois ou plus (minimal).
??	Non	Correspond 0 ou 1 fois (minimal).
{*MIN*,}?	Non	Correspond au moins *MIN* fois (minimal).
{*MIN*,*MAX*}?	Non	Correspond au moins *MIN* fois, mais pas plus de *MAX* fois (minimal).

Un quantificateur minimal essaie de correspondre aussi *peu* que possible dans l'intervalle autorisé. Un quantificateur maximal essaie de correspondre le *plus* possible, dans l'intervalle autorisé. Par exemple, `.+` est sûr de correspondre à au moins un des caractères de la chaîne, mais correspondra à tous s'il en a l'occasion. Ces occasions seront décrites plus loin dans *« Le petit Moteur qui /(ne)?pouvait(pas)?/ »*.

Vous remarquerez qu'un quantificateur ne peut jamais être quantifié.

Nous voulions fournir une syntaxe extensible pour pouvoir introduire de nouveaux types de métasymboles. Étant donné que nous n'avions qu'une douzaine de métacaractères à notre disposition, nous avons choisi d'utiliser une séquence de regex anciennement incorrecte pour ces extensions syntaxiques arbitraires. Ces métasymboles sont tous de la forme `(?`*`CLE`*`...)` ; c'est-à-dire une parenthèse (équilibrée) suivie d'un point d'interrogration, suivie par une *`CLE`* et le reste du sous-motif. Le caractère *`CLE`* indique de quelle extension de regex il s'agit. Voyez le *tableau 5-6* pour en avoir la liste. La plupart d'entre eux se comportent bien structurellement, car ils sont basés sur des parenthèses, mais ont également une signification supplémentaire. De nouveau, seuls les atomes peuvent être quantifiés, car ils représentent quelque chose qui est vraiment là (potentiellement).

Tableau 5-6. Séquences de regex étendues

Extension	Atomique	Signification
`(?#...)`	Non	Commentaire, ignoré.
`(?:...)`	Oui	Parenthèses de regroupement seul, sans capture.
`(?imsx-imsx)`	No	Active/désactive les modificateurs de motif.
`(?imsx-imsx:...)`	Oui	Parenthèses de regroupement, et modificateurs.
`(?=...)`	Non	Vrai si l'assertion de pré-vision est un succès.
`(?!...)`	Non	Vrai si l'assertion de pré-vision est un échec.
`(?<=...)`	Non	Vrai si l'assertion de rétro-vision est un succès.
`(?<!...)`	Non	Vrai si l'assertion de rétro-vision est un échec.
`(?>>...)`	Yes	Correspond au motif sans retour arrière.
`(?{...})`	Non	Exécute le code Perl inclus.
`(??{...})`	Oui	Fait correspondre la regex du code Perl inclus.
`(?(...)...|...)`	Oui	Correspond avec un motif si-alors-sinon.
`(?(...)...)`	Oui	Correspond avec un motif si-alors.

Enfin, *tableau 5-7* donne la liste de tous les métasymboles alphanumériques usuels. (Les symboles qui sont traités durant la passe d'interpolation de variables sont indiqués avec un tiret dans la colonne Atomique, car le Moteur ne les voit jamais.)

Les accolades sont facultatives pour `\p` et `\P` si le nom de la propriété fait un seul caractère. Les accolades sont facultatives pour `\x` si le nombre hexadécimal fait deux chiffres ou moins. Les accolades ne sont jamais facultatives pour `\N`.

Tableau 5-7. Métasymboles alphanumériques de regex

Symbole	Atomique	Signification
`\0`	Oui	Correspond au caractère nul (ASCII NUL).
`\`*`NNN`*	Oui	Correspond au caractère donné en octal, jusqu'à `\377`.
`\`*`n`*	Oui	Correspond à la *n*ième chaîne capturée (en décimal).
`\a`	Oui	Correspond au caractère alarme (BEL).
`\A`	Non	Vrai au début d'une chaîne.
`\b`	Oui	Correspond au caractère espace arrière (BS).
`\b`	Non	Vrai à une limite de mot.
`\B`	Non	Vrai hors d'une limite de mot.
`\c`*`X`*	Oui	Correspond au caractère de contrôle Contrôle-*X* (`\cZ`, `\c[`, etc.).
`\C`	Oui	Correspond à un octet (un `char` en C), même en utf8 (dangereux).
`\d`	Oui	Correspond à tout caractère numérique.
`\D`	Oui	Correspond à tout caractère non numérique.
`\e`	Oui	Correspond au caractère d'échappement (ASCII ESC, pas l'antislash).
`\E`	—	Termine la traduction de casse (`\L`, `\U`) ou la citation de métacaractères.
`\f`	Oui	Correspond au caractère de fin de page (FF).
`\G`	Non	Vrai à la position de fin de recherche du `m//g` précédent.
`\l`	—	Passe le prochain caractère seulement en minuscules.
`\L`	—	Minuscules jusqu'au prochain `\E`.
`\n`	Oui	Correspond au caractère de saut de ligne (en général NL, mais CR sur les Mac).
`\N{`*`NAME`*`}`	Oui	Correspond au caractère nommé (`\N{greek:Sigma}`).
`\p{`*`PROP`*`}`	Oui	Correspond à tout caractère ayant la propriété citée.
`\P{`*`PROP`*`}`	Oui	Correspond à tout caractère n'ayant pas la propriété citée.
`\Q`	—	Cite les métacaractères (désactive leur côté méta) jusqu'au prochain `\E`.
`\r`	Oui	Correspond au caractère de retour chariot (CR en général, mais NL sur les Mac).
`\s`	Oui	Correspond à tout caractère blanc.
`\S`	Oui	Correspond à tout caractère non blanc.
`\t`	Oui	Correspond au caractère de tabulation (HT).
`\u`	—	Passe le prochain caractère seulement en grandes initiales.
`\U`	—	Passe en majuscules (et non en grandes initiales) jusqu'au prochain `\E`.
`\w`	Oui	Correspond à tout caractère de « mot » (alphanumériques et « _ »).

Tableau 5-7. Métasymboles alphanumériques de regex (suite)

Symbole	Atomique	Signification
`\W`	Oui	Correspond à tout caractère non « mot ».
`\x{`*abcd*`}`	Oui	Correspond au caractère donné en hexadécimal.
`\X`	Oui	Correspond à une « séquence de caractères Unicode combinés ».
`\z`	Non	Vrai seulement en fin de chaîne.
`\Z`	Non	Vrai en fin de chaîne ou avant un éventuel saut de ligne.

Seuls les métasymboles dont la description est « Correspond à... » peuvent être utilisés dans une classe de caractères. C'est-à-dire que les classes de caractères ne peuvent contenir que des ensembles spécifiques de caractères, et vous ne pouvez donc utiliser à l'intérieur que des métasymboles qui décrivent d'autres ensembles spécifiques de caractères ou qui décrivent des caractères individuels. Bien sûr, ces métasymboles peuvent aussi être utilisés en dehors des classes de caractères, avec les autres métasymboles (qui ne peuvent être utilisés qu'hors classe). Remarquez cependat que `\b` est en fait deux bestioles complètement différentes : c'est un espace arrière dans une classe de caractères, mais une assertion de limite de mot en dehors.

Il y a un certain chevauchement entre les caractères auxquels un motif peut correspondre et les caractères qu'une chaîne entre apostrophes doubles peut interpoler. Comme les regex subissent deux passes, il est parfois ambigu de savoir quelle passe doit traiter quel caractère. Quand il y a une ambiguïté, la passe d'interpolation de variable défère l'interprétation de ces caractères à l'analyseur d'expressions régulières.

Mais la passe d'interpolation de variable ne peut déférer à l'analyseur de regex que quand elle sait qu'elle est en train d'analyser une regex. Vous pouvez spécifier des expressions régulières comme des chaînes entre apostrophes doubles ordinaires, mais vous devez alors suivre les règles normales des apostrophes doubles. Chacun des métasymboles précédents qui correspond à un caractère existant continuera de marcher, même s'il n'est pas déféré à l'analyseur de regex. Mais vous ne pourrez utiliser aucun des autres métasymboles entre des apostrophes doubles ordinaires (ou dans toute construction analogue comme `` `...` ``, `qq(...)`, `qx(...)` ou les documents ici-même équivalents). Si vous voulez que votre chaîne soit analysée comme une expression régulière sans faire de correspondance, vous devriez utiliser l'opérateur `qr//` (quote regex).

Notez que les séquences d'échappement de casse et de métacitation (`\U` et compagnie) doivent être traités durant la passe d'interpolation, car leur rôle est de modifier la façon dont les variables sont interpolées. Si vous supprimez l'interpolation de variable avec des apostrophes simples, vous n'aurez pas non plus les séquences d'échappement. Ni les variables ni les séquences d'échappement (`\U`, etc.) ne sont étendues dans les chaînes entre apostrophes simples, ni dans les opérateurs `m'...'` ou `qr'...'` avec apostrophes simples. Et même si vous faites une interpolation, ces séquences d'échappement sont ignorées si elles apparaissent dans le *résultat* de l'interpolation de variable, car à ce moment-là il est trop tard pour influencer l'interpolation de variable.

Bien que l'opérateur de translittération ne prenne pas d'expression régulière, tout métasymbole précédemment discuté qui correspond à un caractère spécifique simple mar-

che aussi dans une opération `tr///`. Les autres ne marchent pas (sauf bien sûr l'antislash qui continue toujours à fonctionner à sa manière bien à lui).

Caractères spécifiques

Comme mentionné précédemment, tout ce qui n'est pas spécial dans un motif se correspond à lui-même. Cela signifie qu'un `/a/` correspond à un « a », un `/=/` correspond à un « = », et ainsi de suite. Certains caractères ne sont cependant pas très faciles à taper au clavier ou, même si vous y arrivez ne s'afficheront pas lisiblement ; les caractères de contrôle sont connus pour cela. Dans une expression régulière, Perl reconnaît les alias d'apostrophes doubles suivants :

Échappement	Signification
`\0`	Caractère nul (ASCII NUL)
`\a`	Alarme (BEL)
`\e`	Échappement (ESC)
`\f`	Fin de page (FF)
`\n`	Saut de ligne (NL, CR sur Mac)
`\r`	Retour chariot (CR, NL sur Mac)
`\t`	Tabulation (HT)

Tout comme dans les chaînes entre apostrophes doubles, Perl reconnaît également les quatre métasymboles suivants dans les motifs :

`\c`*X*

Un caractère de contrôle nommé, comme `\cC` pour Contrôle-C, `\cZ` pour Contrôle-Z, `\c[` pour ESC, et `\c?` pour DEL.

`\`*NNN*

Un caractère spécifié en utilisant son code octal à deux ou trois chiffres. Le `0` initial est facultatif, sauf pour les valeurs inférieures à `010` (8 en décimal), car (contrairement aux chaînes entre apostrophes doubles) les versions à un seul chiffre sont toujours considérées comme des références arrières à des chaînes capturées dans un motif. Les versions à plusieurs chiffres sont interprétées comme la *n*ième référence arrière si vous avez capturé au moins *n* sous-chaînes plus tôt dans le motif (avec *n* considéré comme un nombre décimal). Sinon, elle est interprétée comme un caractère octal.

`\x{`*HEXLONG*`}`

`\x`*HEX*

Un caractère spécifié par son numéro à un ou deux chiffres hexadécimaux (`[0-9a-fA-F]`), comme `\x1B`. La forme à un seul chiffre est utilisable quand le caractère suivant n'est pas un chiffre hexadécimal. Si vous utilisez des accolades, vous pouvez mettre autant de chiffres que vous voulez, ce qui peut conduire à un caractère Unicode. Par exemple, `\x{262f}` correspond à un YIN YANG Unicode.

`\N{`*NOM*`}`

Un caractère nommé, comme `\N{GREEK SMALL LETTER EPSILON}`, `\N{greek:epsilon}`, ou `\N{epsilon}`. Ceci requiert l'utilisation du pragma `use charnames` décrit au chapitre 31, *Modules de pragmas*, et qui détermine aussi lequel de ces noms vous pouvez utiliser (`":long"`, `":full"`, `":short"` qui correspondent respectivement aux trois styles que nous venons de citer).

Vous trouverez une liste de tous les noms de caractères Unicode dans vos documents de standard Unicode préférés ou dans le fichier *PATH_TO_PERLLIB/unicode/Names.txt*.

Métasymboles jokers

Trois métasymboles spéciaux sont utilisés comme des jokers génériques, chacun d'entre eux correspondant à « tout » caractère (pour certaines valeurs de « tout »). Ce sont le point (`"."`), `\C` et `\X`. Aucun d'entre eux ne peut être utilisé dans une classe de caractères. Vous ne pouvez pas y utiliser le point car il correspondrait à (presque) tout caractère existant. C'est donc déjà une sorte de classe de caractères universelle par lui-même. Si vous voulez tout inclure ou tout exclure, il n'y a pas grand intérêt à utiliser une classe de caractères. Les jokers spéciaux `\C` et `\X` ont une signification structurelle spécifique qui ne s'accorde pas bien avec l'idée de choisir un seul caractère Unicode, ce qui est le niveau auquel fonctionne une classe de caractères.

Le métacaractère point correspond à tout caractère autre que le saut de ligne. (Et avec le modificateur `/s`, il lui correspond aussi.) Comme chacun des autres caractères spéciaux dans un motif, pour correspondre à un point littéral, vous devez le protéger avec un antislash. Par exemple, ceci vérifie si un nom de fichier se termine par un point suivi d'une extension à un caractère :

```
if ($chemin =~ /\.(.)\z/s) {
    print "Se termine par $1\n";
}
```

Le premier point, celui qui est protégé, est un caractère littéral, tandis que le second dit « correspond à tout caractère ». Le `\z` demande de ne correspondre qu'à la fin de la chaîne, et le modificateur `/s` permet au point de correspondre également à un saut de ligne. (Certes, utiliser un saut de ligne comme extension n'est Pas Très Gentil, mais cela ne veut pas dire que cela ne peut pas se produire.)

Le métacaractère point est le plus souvent utilisé avec un quantificateur. Un `.*` correspond à un nombre maximal de caractères, tandis qu'un `.*?` correspond à un nombre minimal de caractères. Mais il est aussi utilisé parfois sans quantificateur pour sa propre largeur : `/(..):(..):(..)/` correspond à trois champs séparés par des deux-points, chacun faisant deux caractères de large.

Si vous utilisez un point dans un motif compilé sous le pragma à portée lexicale `use utf8`, alors il correspondra à tout caractère Unicode. (Vous n'êtes pas supposé avoir besoin de `use utf8` pour cela, mais il y a toujours des accidents. Le pragma peut ne plus être utile d'ici à ce que vous lisiez ces lignes.)

```
use utf8;
use charnames qw/:full/;
```

```
$BWV[887] = "G\N{MUSIC SHARP SIGN} minor"; # partition anglaise
($note, $noire, $mode) = $BWV[887] =~ /^([A-G])(.)\s+(\S+)/;
print "C'est en dièse !\n" if $n     eq chr(9839);
```

Le métasymbole `\X` correspond à un caractère Unicode dans un sens plus étendu. Il correspond en fait à une chaîne d'un ou plusieurs caractères Unicode connue sous le nom de « séquence de caractères combinés ». Une telle séquence consiste en un caractère de base suivi par l'un des caractères de « marquage » (signe diacritique comme une cédille ou un tréma) qui se combine avec le caractère de base pour former une unité logique. `\X` est exactement équivalent à `(?:\PM\pM*)`. Cela permet de correspondre à un seul caractère logique, même si celui est en fait constitué de plusieurs caractères distincts. La longueur de la correspondance de `/\X/` peut excéder un caractère si celui-ci a trouvé des caractères combinés. (Et nous parlons ici de la longueur en caractères, qui n'a pas grand chose à voir avec la longueur en octets.)

Si vous utilisez Unicode et que vous voulez vraiment travailler sur un seul octet plutôt que sur un seul caractère, vous pouvez utiliser le métasymbole `\C`. Ceci correspondra toujours à un seul octet (spécifiquement, à un seul élément de type `char` de C), même si cela vous désynchronise de votre flux de caractères Unicode. Référez-vous aux avertissements relatifs à cela au chapitre 15.

Classes de caractères

Dans une recherche de motif, vous pouvez établir une correspondance avec tout caractère qui a — ou n'a pas — une propriété particulière. Il existe quatre manières de spécifier les classes de caractères. Vous pouvez spécifier une classe de caractères de la manière traditionnelle en utilisant des crochets et en énumérant les caractères possibles, ou en utilisant l'un des trois raccourcis mnémotechniques : les classes Perl habituelles, les nouvelles propriétés Unicode de Perl ou les classes POSIX standard. Chacun de ces raccourcis correspond à un seul caractère de son ensemble. Quantifiez-les si vous voulez faire une correspondance avec un plus grand nombre d'entre eux, comme `\d+` pour correspondre à plusieurs chiffres. (Une erreur facile à faire est de croire que `\w` correspond à un mot. Utilisez `\w+` pour trouver un mot.)

Classes de caractères personnalisées

Une liste de caractères énumérée entre crochets est appelée une *classe de caractère* et correspond à tout caractère dans la liste. Par exemple, `[aeiouy]` correspond à une voyelle (non accentuée). Pour correspondre à un crochet fermant, antislashez-le ou bien placez-le en premier dans la liste.

Des intervalles de caractères peuvent être indiqués en utilisant un tiret et la notation `a-z`. Plusieurs intervalles peuvent être combinés ; par exemple `[0-9a-fA-F]`correspond à un « chiffre » hexadécimal. Vous pouvez utiliser un antislash pour protéger un tiret qui serait sinon interprété comme un délimiteur d'intervalle, ou simplement le placer en début de liste (une pratique probablement moins lisible mais plus traditionnelle).

Un chapeau (ou accent circonflexe, ou flèche vers le haut) au début de la classe de caractère l'inverse, la faisant correspondre à tout caractère ne se trouvant *pas* dans la liste. (Pour correspondre à un chapeau, ou bien ne le mettez *pas* au début, ou mieux, proté-

gez-le avec un antislash.) Par exemple, `[^aeiouy]` correspond à tout caractère qui n'est pas une voyelle (non accentuée). Cependant soyez prudent avec la négation de classe, car l'univers des caractères est en expansion. Par exemple, cette classe de caractère correspond aux consonnes — et aussi aux espaces, aux sauts de ligne, et à tout (y compris les voyelles) en cyrillique, en grec ou tout autre alphabet, sans compter tous les idéogrammes chinois, japonais et coréens. Et un jour aussi le cirth, le tengwar et le klingon. (Mais déjà le linéaire B et l'étrusque.) Il vaut donc peut-être mieux spécifier vos consonnes explicitement, comme `[bcdfghjklmnpqrstvwxyz]`, ou `[b-df-hj-np-tv-z]` pour faire plus court. (Cela résout aussi le problème du « y » qui devrait être à deux endroits à la fois, ce qu'interdit l'utilisation de l'ensemble complémentaire.)

Les métasymboles caractères normaux, comme `\n`, `\t`, `\c`*X*, `\`*NNN* et `\N{`*NOM*`}` sont utilisables dans une classe de caractères (voir la section *Caractères spécifiques*). De plus, vous pouvez utiliser `\b` dans une classe de caractères pour signifier un espace arrière, exactement comme dans une chaîne entre apostrophes doubles. Normalement, dans une recherche de motif, il représente une limite de mot. Mais les assertions de largeur nulle n'ont aucun sens dans une classe de caractères, c'est pourquoi `\b` retrouve la signification qu'il a dans les chaînes. Vous pouvez également utiliser les classes de caractères prédéfinies qui sont décrites plus loin dans ce chapitre (classique, Unicode ou POSIX), mais n'essayez pas de les utiliser comme point initial ou final d'un intervalle — cela n'ayant pas de sens, le « - » sera interprété comme un caractère littéral.

Tous les autres métasymboles perdent leur signification spéciale quand ils sont entre crochets. En particulier, vous ne pouvez utiliser aucun des trois jokers génériques : « `.` », `\X`ou `\C`. Le premier surprend souvent, mais cela n'a pas grand sens d'utiliser la classe de caractère universelle à l'intérieur d'une classe plus restrictive, et de plus vous avez souvent besoin de faire une correspondance avec un point littéral dans une classe de caractère — quand vous travaillez sur des noms de fichier, par exemple. Cela n'a pas grand sens non plus de spécifier des quantificateurs, des assertions ou des alternatives dans une classe de caractères, puisque les caractères sont interprétés individuellement. Par exemple, `[fee|fie|foe|foo]` a exactement la même signification que `[feio|]`.

Raccourcis de classes de caractères classiques Perl

Depuis l'origine, Perl a fourni un certain nombre de raccourcis pour les classes de caractères. La liste est fournie dans le *tableau 5-8*. Tous sont des métasymboles alphabétiques antislashés, et dans chaque cas, la version en majuscule est la négation de la version en minuscules. Leur signification est n'est pas aussi rigide que vous pourriez le croire : leur sens peut être influencé par les définitions de locales. Même si vous n'utilisez pas les locales, leur signification peut changer à chaque fois qu'un nouveau standard Unicode sortira, ajoutant de nouveaux scripts avec de nouveaux chiffres et de nouvelles lettres. (Pour conserver l'ancienne signification par octets, vous pouvez toujours utiliser `use bytes`. Voir *Propriétés Unicode* un peu plus loin dans ce chapitre pour des explications sur les correspondances utf8. Dans tous les cas, les correspondances utf8 sont un surensemble des correspondances sur les octets.) (Oui, nous savons que la plupart des mots ne comportent ni chiffres ni soulignés mais parfois des accents ; `\w` sert à trouver des « mots » au sens des tokens dans un langage de programmation usuel. Ou en Perl.)

Tableau 5-8. classe de caractères classiques

Symbole	Signification	En octets	En utf8
\d	Chiffre	[0-9]	\p{IsDigit}
\D	Non chiffre	[^0-9]	\P{IsDigit}
\s	Blanc	[\t\n\r\f]	\p{IsSpace}
\S	Non blanc	[^ \t\n\r\f]	\P{IsSpace}
\w	Caractère de mot	[a-zA-Z0-9_]	\p{IsWord}
\W	Non-(caractère mot)	[^a-zA-Z0-9_]	\P{IsWord}

Ces métacaractères peuvent être utilisés à l'extérieur ou à l'intérieur de crochets, c'est-à-dire soit par eux-mêmes ou comme élément d'une classe de caractères :

```
if ($var =~ /\D/)        { warn "contient des non-chiffres" }
if ($var =~ /[^\w\s.]/)  { warn "contient des non-(mot, espace, point)" }
```

Propriétés Unicode

Les propriétés Unicode sont disponibles en utilisant \p{*PROP*} et son ensemble complémentaire \P{*PROP*}. Pour les rares propriétés qui ont un nom en un seul caractère, les accolades sont facultatives, comme dans \pN pour indiquer un caractère numérique (pas nécessairement décimal — les chiffres romains sont aussi des caractères numériques). Ces classes de propriétés peuvent être utilisées par elles-mêmes ou combinées dans une classe de caractères :

```
if ($var =~ /^\p{IsAlpha}+$/)      { print "tout alphabétique" }
if ($var =~ s/[\p{Zl}\p{Zp}]/\n/g) { print "saut de lignes" }
```

Certaines propriétés sont directement définies dans le standard Unicode, et d'autres sont des composées définies par Perl, en s'appuyant sur les propriétés standard. Zl et Zp sont des propriétés Unicode standard qui représentent les séparateurs de lignes et de paragraphes, tandis qu'IsAlpha est définie par Perl comme une classe regroupant les propriétés standard Ll, Lu, Lt et Lo (c'est-à-dire les lettres qui sont en majuscules, en minuscules, en casse de titre ou autre). À partir de la version 5.6.0 de Perl, vous avez besoin de use utf8 pour que ces propriétés fonctionnent. Cette restriction sera assouplie dans le futur.

Il y a un grand nombre de propriétés. Nous allons lister celles que nous connaissons, mais la liste sera forcément incomplète. De nouvelles propriétés sont susceptibles d'apparaître dans les nouvelles versions d'Unicode, et vous pouvez même définir vos propres propriétés. Nous verrons cela plus loin.

Le Consortium Unicode produit les informations que l'on retrouve dans les nombreux fichiers que Perl utilise dans son implémentation d'Unicode. Pour plus d'informations sur ces fichiers, voyez le chapitre 15. Vous pouvez lire un bon résumé de ce qu'est Unicode dans le document *PATH_TO_PERLLIB/unicode/Unicode3.html* où *PATH_TO_PERLLIB* est ce qui est affiché par :

```
perl -MConfig -le 'print $Config{privlib}'
```

La plupart des propriétés Unicode sont de la forme \p{Is*PROP*}. Le Is est facultatif, mais vous pouvez le laisser pour des raisons de lisibilité.

Propriétés Unicode de Perl

Le *tableau 5-9* donne la liste des propriétés composites de Perl. Elles sont définies de façon à être raisonnablement proches des définitions POSIX standard pour les classes de caractères.

Tableau 5-9. Propriétés Unicode composites

Proprété	Équivalent
`IsASCII`	`[\x00-\x7f]`
`IsAlnum`	`[\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]`
`IsAlpha`	`[\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}]`
`IsCntrl`	`\p{IsC}`
`IsDigit`	`\p{Nd}`
`IsGraph`	`[^\pC\p{IsSpace}]`
`IsLower`	`\p{IsLl}`
`IsPrint`	`\P{IsC}`
`IsPunct`	`\p{IsP}`
`IsSpace`	`[\t\n\f\r\p{IsZ}]`
`IsUpper`	`[\p{IsLu}\p{IsLt}]`
`IsWord`	`[_\p{IsLl}\p{IsLu}\p{IsLt}\p{IsLo}\p{IsNd}]`
`IsXDigit`	`[0-9a-fA-F]`

Perl fournit également les composites suivants pour chacune des principales catégories des propriétés Unicode standard (voir la section suivante) :

Propriété	Signification	Normative
`IsC`	Codes de contrôle et équivalents	Oui
`IsL`	Lettres	Partiellement
`IsM`	Marqueurs	Oui
`IsN`	Nombres	Oui
`IsP`	Ponctuation	Non
`IsS`	Symboles	Non
`IsZ`	Séparateurs (Zéparateurs?)	Oui

Propriétés Unicode standard

Le *tableau 5-10* donne la liste des propriétés Unicode standard les plus basiques, qui dérivent de la catégorie de chaque caractère. Aucun caractère n'est membre de plus d'une catégorie. Certaines propriétés sont normatives ; d'autres sont simplement informatives. Reportez-vous au standard Unicode pour savoir combien l'information normative est normative, et combien l'information informative ne l'est pas.

Tableau 5-10. Propriétés Unicode standard

Propriété	Signification	Normative
`IsCc`	Autre, contrôle	Oui
`IsCf`	Autre, format	Oui
`IsCn`	Autre, non affecté	Oui
`IsCo`	Autre, usage privé	Oui
`IsCs`	Autre, substitut	Oui
`IsLl`	Lettre, minuscule	Oui
`IsLm`	Lettre, modificateur	Non
`IsLo`	Lettre, autre	Non
`IsLt`	Lettre, casse de titre	Oui
`IsLu`	Lettre, majuscule	Oui
`IsMc`	Marque, combinante	Oui
`IsMe`	Marque, entourant	Oui
`IsMn`	Marque, non-espacement	Oui
`IsNd`	Nombre, chiffre décimal	Oui
`IsNl`	Nombre, lettre	Oui
`IsNo`	Nombre, autre	Oui
`IsPc`	Ponctuation, connecteur	Non
`IsPd`	Ponctuation, tiret	Non
`IsPe`	Ponctuation, fermeture	Non
`IsPf`	Ponctuation, guillemet fermant	Non
`IsPi`	Ponctuation, guillemet ouvrant	Non
`IsPo`	Ponctuation, autre	Non
`IsPs`	Ponctuation, ouvert	Non
`IsSc`	Symbole, monnaie	Non
`IsSk`	Symbole, modificateur	Non
`IsSm`	Symbole, math	Non
`IsSo`	Symbole, autre	Non
`IsZl`	Séparateur, de ligne	Oui
`IsZp`	Séparateur, de paragraphe	Oui
`IsZs`	Séparateur, espace	Oui

Un autre ensemble de propriétés utiles est lié au fait qu'un caractère peut être décomposé (de façon canonique ou compatible) en d'autres caractères plus simples. La décomposition canonique ne perd aucune information de formatage. La décomposition compatible peut perdre des informations de formatage, comme le fait de savoir si le caractère est en exposant.

Propriété	**Information perdue**
`IsDecoCanon`	Rien
`IsDecoCompat`	Quelque chose (parmi les suivants)
`IsDCcircle`	Cercle autour du caractère
`IsDCfinal`	Préférence de position finale (Arabe)
`IsDCfont`	Préférence de variante de fonte
`IsDCfraction`	Caractéristique de fraction ordinaire
`IsDCinitial`	Préférence de position initiale (Arabe)
`IsDCisolated`	Préférence de position isolée (Arabe)
`IsDCmedial`	Préférence de position médiane (Arabe)
`IsDCnarrow`	Caractéristique d'étroitesse
`IsDCnoBreak`	Préférence de non-coupure sur un espace ou un tiret
`IsDCsmall`	Caractéristique de petite taille
`IsDCsquare`	Carré autour d'un caractère CJK
`IsDCsub`	Position d'indice
`IsDCsuper`	Position d'exposant
`IsDCvertical`	Rotation (d'horizontal à vertical)
`IsDCwide`	Caractéristique de largeur
`IsDCcompat`	Identité (variable)

Voici quelques propriétés qui intéressent les personnes faisant du rendu bidirectionnel :

Proprité	**Signification**
`IsBidiL`	Gauche à droite (Arabe, Hébreu)
`IsBidiLRE`	Inclusion de gauche à droite
`IsBidiLRO`	Force de gauche à droite
`IsBidiR`	Droite à gauche
`IsBidiAL`	Droite à gauche arabe
`IsBidiRLE`	Inclusion de droite-àgauche
`IsBidiRLO`	Force de droite à gauche
`IsBidiPDF`	Dépile le format directionnel
`IsBidiEN`	Nombre européen
`IsBidiES`	Séparateur de nombre européen
`IsBidiET`	Terminateur de nombre européen
`IsBidiAN`	Nombre arabe
`IsBidiCS`	Séparateur de nombres communs
`IsBidiNSM`	Marque de non-espacement
`IsBidiBN`	Limite neutre
`IsBidiB`	Séparateur de paragraphe

Proprité	Signification
IsBidiS	Séparateur de segment
IsBidiWS	Blanc
IsBidiON	Autres neutres
IsMirrored	Image miroir quand utilisé de droite à gauche

Les propriétés suivantes classent différents syllabaires selon les sonorités des voyelles :

```
IsSylA      IsSylE      IsSylO      IsSylWAA    IsSylWII
IsSylAA     IsSylEE     IsSylOO     IsSylWC     IsSylWO
IsSylAAI    IsSylI      IsSylU      IsSylWE     IsSylWOO
IsSylAI     IsSylII     IsSylV      IsSylWEE    IsSylWU
IsSylC      IsSylN      IsSylWA     IsSylWI     IsSylWV
```

Par exemple, `\p{IsSylA}` correspondrait à `\N{KATAKANA LETTER KA}` mais pas à `\N{KATAKANA LETTER KU}`.

Maintenant que nous vous avons parlé de toutes ces propriétés Unicode 3.0, nous devons tout de même lister quelques unes des plus ésotériques qui ne sont pas implémentées dans Perl 5.6.0, car cette implémentation était basée en partie sur Unicode 2.0, et que des choses comme l'algorithme bidirectionnel étaient encore sur le métier. Cependant, comme d'ici à ce que vous lisiez ceci, les propriétés manquantes pourraient bien avoir été implémentées, nous les avons listées quand même.

Propriétés de bloc Unicode

Certaines propriétés Unicode sont de la forme `\p{In`*`SCRIPT`*`}`. (Remarquez la distinction entre `Is` et `In`.) Les propriétés `In` servent à tester l'appartenance à des blocs d'un *`SCRIPT`* unique. Si vous avez un caractère et que vous vous demandez s'il est écrit en écriture hébraïque, vous pouvez le tester avec :

```
print "Pour moi c'est de l'Hébreu!\n" if chr(1488) =~ /\p{InHebrew}/;
```

Cela fonctionne en vérifiant si un caractère est « dans » l'intervalle valide de ce type d'écriture. Cela peut être inversé avec `\P{In`*`SCRIPT`*`}` pour savoir si quelque chose n'est *pas* dans un bloc de script particulier, comme `\P{InDingbats}` pour savoir si une chaîne contient un caractère non dingbat. Dans les propriétés de bloc on trouve les éléments suivants :

```
InArabic        InCyrillic      InHangulJamo    InMalayalam     InSyriac
InArmenian      InDevanagari    InHebrew        InMongolian     InTamil
InArrows        InDingbats      InHiragana      InMyanmar       InTelugu
InBasicLatin    InEthiopic      InKanbun        InOgham         InThaana
InBengali       InGeorgian      InKannada       InOriya         InThai
InBopomofo      InGreek         InKatakana      InRunic         InTibetan
InBoxDrawing    InGujarati      InKhmer         InSinhala       InYiRadicals
InCherokee      InGurmukhi      InLao           InSpecials      InYiSyllables
```

Sans oublier les impressionnants :

```
InAlphabeticPresentationForms       InHalfwidthandFullwidthForms
InArabicPresentationForms-A         InHangulCompatibilityJamo
InArabicPresentationForms-B         InHangulSyllables
```

```
InBlockElements
InBopomofoExtended
InBraillePatterns
InCJKCompatibility
InCJKCompatibilityForms
InCJKCompatibilityIdeographs
InCJKRadicalsSupplement
InCJKSymbolsandPunctuation
InCJKUnifiedIdeographs
InCJKUnifiedIdeographsExtensionA
InCombiningDiacriticalMarks
InCombiningHalfMarks
InCombiningMarksforSymbols
InControlPictures
InCurrencySymbols
InEnclosedAlphanumerics
InEnclosedCJKLettersandMonths
InGeneralPunctuation
InGeometricShapes
InGreekExtended
InHighPrivateUseSurrogates
InHighSurrogates
InIdeographicDescriptionCharacters
InIPAExtensions
InKangxiRadicals
InLatin-1Supplement
InLatinExtended-A
InLatinExtended-B
InLatinExtendedAdditional
InLetterlikeSymbols
InLowSurrogates
InMathematicalOperators
InMiscellaneousSymbols
InMiscellaneousTechnical
InNumberForms
InOpticalCharacterRecognition
InPrivateUse
InSuperscriptsandSubscripts
InSmallFormVariants
InSpacingModifierLetters
```

Et le mot le plus long :

```
InUnifiedCanadianAboriginalSyllabics
```

Pas mieux.

Consultez *PATH_TO_PERLLIB/unicode/In/*.pl* pour obtenir une liste à jour de toutes ces propriétés de blocs de caractères. Remarquez que les propriétés `In` ne font que vérifier si le caractère est dans l'intervalle alloué pour ce script. Rien ne vous garantit que tous les caractères dans cet intervalle sont définis ; vous devrez également tester l'une des propriétés `Is` discutées précédemment pour vérifier si le caractère est défini. Rien ne vous garantit non plus qu'une langue particulière n'utilise pas des caractères en dehors de l'intervalle qui lui est affecté. En particulier, de nombreuses langues européennes mélangent les caractères latins étendus avec des caractères Latin-1.

Mais bon, si vous avez besoin d'une propriété particulière qui n'est pas fournie, ce n'est pas un gros problème. Continuez à lire.

Définir vos propres propriétés de caractères

Pour définir vos propres propriétés, vous devrez écrire un sous-programme portant le nom de la propriété que vous voulez (voir chapitre 6, *Sous-programmes*). Le sous-programme doit être défini dans le paquetage qui a besoin de la propriété, ce qui signifie que si vous désirez l'utiliser dans plusieurs paquetages, vous devrez soit l'importer d'un autre module (voir chapitre 11, *Modules*), ou l'hériter comme méthode de classe depuis le paquetage dans lequel il est défini (voir chapitre 12, *Objets*).

Une fois que cela est fait, le sous-programme doit renvoyer des données dans le même format que les fichiers du répertoire *PATH_TO_PERLLIB/unicode/Is*. C'est-à-dire que vous devez juste retourner une liste de caractères ou des intervalles de caractères en hexadécimal, un par ligne. S'il y a un intervalle, les deux nombres sont séparés par une tabulation. Supposons que vous vouliez créer une propriété qui serait vraie si votre caractère

est dans l'intervalle de l'un des deux syllabaires japonais, nommés hiragana et katakana. (Ensemble on les appelle simplement kana.) Vous pouvez juste mettre les deux intervalles comme suit :

```
sub InKana {
    return <<'END';
3040    309F
30A0    30FF
END
}
```

Vous pourriez également le définir à l'aide de noms de propriétés existantes :

```
sub InKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}
```

Vous pouvez aussi faire une soustraction en utilisant un préfixe « - ». Supposons que vous ne vouliez que les véritables caractères, et pas seulement les blocs de caractères. Vous pouvez retirer tous les caractères indéfinis comme ceci :

```
sub IsKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}
```

Vous pouvez aussi démarrer avec le complément d'un ensemble de caractères en utilisant le préfixe « ! » :

```
sub IsNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}
```

Perl lui-même utilise les mêmes trucs pour définir la signification de ses propres classes de caractères « classiques » (comme `\w`) quand vous les incluez dans vos classes de caractères personnalisées (comme `[-.\w\s]`). Vous pourriez penser que plus les règles deviennent compliquées, plus lentement cela tournera ; mais en fait, une fois que Perl a calculé le motif de bits pour le bloc de 64 bits de votre propriété, il le mémorise pour n'avoir jamais à le recalculer. (Cela est fait dans des blocs de 64 bits pour ne même pas avoir à décoder votre utf8 pour faire les recherches.) Ainsi, toutes les classes de caractères, internes ou personnalisées, tournent toutes à peu près à la même vitesse (rapide) une fois mises en route.

Classes de caractères de style POSIX

Contrairement aux autres raccourcis de classes de caractères, la notation de style POSIX, `[:CLASSE:]`, n'est utilisable *que* pour la construction d'autres classes de caractères, c'est-à-dire à l'intérieur d'une autre paire de crochets. Par exemple, `/[.,[:alpha:][:digit:]]/` va chercher un caractère qui est soit un point (littéral, car dans une classe de caractères), une virgule, ou un caractère alphabétique ou un nombre.

Les classes POSIX disponibles à partir de la version 5.6 de Perl sont listées dans le *tableau 5-11*.

Tableau 5-11. Classes de caractères POSIX

Classe	Signification
`alnum`	Tout alphanumérique, c'est-à-dire un `alpha` ou un `digit`.
`alpha`	Toute lettre. (Cela fait beaucoup plus de lettres que vous pensez, sauf si vous pensez en Unicode, auquel cas cela fait quand même beaucoup.)
`ascii`	Tout caractère avec une valeur numérique comprise entre 0 et 127.
`cntrl`	Tout caractère de contrôle. En général ce sont des caractères qui ne produisent pas d'affichage par eux-mêmes, mais contrôlent le terminal d'une façon ou d'une autre. Par exemple, les caractères de nouvelle ligne, de fin de page ou d'espace arrière sont des caractères de contrôle. Les caractères ayant une valeur numérique inférieure à 32 sont le plus souvent considérés comme des caractères de contrôle.
`digit`	Un caractère représentant un chiffre décimal, comme ceux entre `0` et `9`. (Cela comprend d'autres caractères en Unicode.) Équivalent à `\d`.
`graph`	Tout caractère alphanumérique ou de ponctuation.
`lower`	Une lettre minuscule.
`print`	Tout caractère alphanumérique, de ponctuation ou d'espacement.
`punct`	Tout caractère de ponctuation.
`space`	Tout caractère d'espacement. Cela inclut tabulation, nouvelle ligne, saut de page et retour chariot (et encore beaucoup plus en Unicode). Équivalent à `\s`.
`upper`	Tout caractère en majuscule (ou casse de titre).
`word`	Tout caractère d'identificateur, soit un `alnum` ou un souligné.
`xdigit`	Tout chiffre hexadécimal. Bien que cela semble un peu bête (`[0-9a-fA-F]` marche très bien), c'est inclus pour être complet.

Vous pouvez inverser la classe de caractères POSIX en préfixant le nom de la classe par un `^` après le `[:`. (C'est une extension apportée par Perl.) Par exemple :

POSIX	Classique
`[:^digit:]`	`\D`
`[:^space:]`	`\S`
`[:^word:]`	`\W`

Si le pragma `use utf8` n'est pas demandé, mais que le pragma `use locale` l'est, la classe correspond directement avec les fonctions équivalentes de l'interface *isalpha*(3) de la librairie C (excepté `word`, qui est une extension Perl correspondant à `\w`).

Si le pragma `utf8` est utilisé, les classes de caractères POSIX sont exactement équivalentes aux propriétés `Is` listées dans le *tableau 5-9*. Par exemple, `[:lower:]` et `\p{Lower}` sont équivalentes, sauf que les classes POSIX ne peuvent être utilisées que dans les classes de caractères construites, tandis que les propriétés Unicode ne sont pas soumises à de telles restrictions et peuvent être utilisées dans des motifs partout où les raccourcis Perl comme `\s` et `\w` peuvent l'être.

Les crochets font partie de la construction de style POSIX `[::]` et pas de la classe de caractères. Ce qui nous amène à écrire des motifs tels que `/^[[:lower:][:digit:]]+$/` pour correspondre à une chaîne consistant entièrement de lettres minuscules ou de chiffres (avec un saut de ligne final facultatif). En particulier, ceci ne fonctionne pas :

```
42 =~ /^[:digit:]+$/          # FAUX
```

Cela est dû au fait qu'on n'est pas dans une classe de caractères. Ou plutôt, *c'est* une classe de caractères, celle qui représente les caractères « `:` », « `i` », « `t` », « `g` » et « `d` ». Perl se moque du fait que vous ayez utilisé deux « `:` » (et deux « `i` »).

Voici ce dont vous avez en fait besoin :

```
42 =~ /^[[:digit:]]+$/
```

Les classes de caractères POSIX `[.cc.]` et `[=cc=]` sont reconnues, mais produisent une erreur indiquant qu'elles ne sont pas utilisables. L'utilisation de *n'importe quelle* classe de caractères POSIX dans une version de Perl plus ancienne ne marchera pas, et de façon probablement silencieuse. Si vous comptez utiliser les classes de caractères POSIX, il est préférable d'imposer l'utilisation d'une nouvelle version de Perl en écrivant :

```
use 5.6.0;
```

Quantificateurs

À moins que vous n'en décidiez autrement, chaque élément d'une expression régulière ne correspond qu'une fois à quelque chose. Avec un motif comme `/nop/`, chacun de ces caractères doit correspondre, l'un à la suite de l'autre. Les mots comme « panoplie » ou « xénophobie » marchent bien, car *où* la correspondance se fait n'a aucune importance.

Si vous vouliez trouver à la fois « xénophobie » et « Snoopy », vous ne pourriez pas utiliser le motif `/nop/`, puisque cela requiert qu'il n'y ait qu'un seul « o » entre le « n » et le « p », et que Snoopy en a deux. C'est ici que les *quantificateurs* montrent leur utilité : ils indiquent combien de fois quelque chose peut correspondre, au lieu de une fois par défaut. Les quantificateurs dans une expression régulière sont comme les boucles dans un programme ; en fait, si vous regardez une expression régulière comme un programme, alors ce *sont* des boucles. Certaines boucles sont exactes, comme « répète cette correspondance cinq fois seulement » (`{5}`). D'autres vous donnent à la fois la limite inférieure et la limite supérieure du nombre de correspondances, comme « répète ceci au moins deux fois, mais pas plus de quatre » (`{2,4}`). D'autres n'ont pas de limite supérieure du tout, comme « trouve ceci au moins deux fois, mais ensuite autant de fois que tu veux » (`{2,}`).

Le *tableau 5-12* liste les quantificateurs que Perl reconnaît dans un motif.

Tableau 5-12. Comparaison des quantificateurs de motifs

Maximum	Minimum	Plage autorisée
{*MIN*,*MAX*}	{*MIN*,*MAX*}?	Doit apparaître au moins *MIN* fois mais pas plus de *MAX* fois
{*MIN*,}	{*MIN*,}?	Doit apparaître au moins *MIN* fois
{*COMPTE*}	{*COMPTE*}?	Doit apparaître exactement *COMPTE* fois
*	*?	0 ou plus (comme {0,})
+	+?	1 fois ou plus (comme {1,})
?	??	0 ou 1 fois (comme {0,1})

Quelque chose suivi d'une * ou d'un ? n'est pas obligé de correspondre à tout prix. C'est parce que cela peut correspondre 0 fois et toujours être considéré comme un succès. Un + est souvent un meilleur choix, car l'élément recherché doit être présent au moins une fois.

Ne vous laissez pas embrouiller par l'emploi de « exactement » dans le tableau précédent. Il ne fait référence qu'au nombre de répétitions, et pas à la chaîne complète. Par exemple, `$n =~ /\d{3}/` ne dit pas « est-ce que cette chaîne fait exactement trois chiffres de long ? ». En fait le motif se préoccupe de savoir s'il existe une position dans `$n` à laquelle trois chiffres apparaissent les uns à la suite des autres. Des chaînes comme « 101 rue de la République » correspondent, tout comme des chaînes telles que « 75019 » ou « 0 825 827 829 ». Toutes *contiennent* trois chiffres à la suite à un ou plusieurs endroits, ce qui est tout ce que vous demandiez. Voir dans la section *Positions* l'utilisation des assertions de position (comme dans `/^\d{3}$/`) pour préciser votre recherche.

Si on leur donne la possibilité de correspondre un nombre variable de fois, les quantificateurs maximaux vont essayer de maximiser le nombre de répétitions. C'est pourquoi quand nous disons « autant de fois que tu veux », les quantificateurs gourmands le comprennent comme « le plus de fois que tu le peux », à la seule condition que cela n'empêche pas des spécifications plus loin dans l'expression de correspondre. Si un motif contient deux quantificateurs ouverts, les deux ne peuvent évidemment pas consommer l'intégralité de la chaîne : les caractères utilisés par la première partie de l'expression ne sont plus disponibles pour la partie qui suit. Chaque quantificateur est gourmand aux dépens de ceux qui le suivent, en lisant le motif de gauche à droite.

C'est du moins le comportement traditionnel des quantificateurs dans les expressions régulières. Cependant Perl vous permet de modifier le comportement de ses quantificateurs : en mettant un ?après un quantificateur, vous le transformez de maximal en minimal. Cela ne signifie pas qu'un quantificateur minimal va toujours correspondre au nombre minimum de répétitions permises par ses spécifications, pas plus qu'un quantificateur maximal correspond toujours au plus grand nombre défini dans ses spécifications. La correspondance globale doit toujours se faire, et le quantificateur minimal prendra juste ce qu'il faut pour que ce soit un succès, et pas plus. (Les quantificateurs minimaux préfèrent la satisfaction à la gourmandise.)

Par exemple dans la correspondance :

```
"exigence" =~ /e(.*)e/    #  $1 vaut maintenant "xigenc"
```

le `.*` correspond à « `xigenc` », la plus longue chaîne possible à laquelle il peut correspondre. (Il stocke également cette valeur dans `$1`, comme cela est expliqué dans la section *Capture et regroupement* plus loin dans ce chapitre.) Bien qu'une correspondance plus courte fut possible, un quantificateur gourmand s'en moque. S'il a le choix entre deux possibilités à partir de la même position, il retournera toujours la *plus longue* des deux.

Comparez ceci avec cela :

```
"exigence" =~ /e(.*?)e/   #  $1 vaut maintenant "xig"
```

Ici, c'est la version minimale, `.*?`, qui est utilisée. L'ajout du `?` à l'* fait que `*?` a un comportement opposé : ayant le choix entre deux possibilités à partir du même point, il prend toujours la *plus courte* des deux.

Bien que vous puissiez lire `*?` comme une injonction à correspondre à zéro ou plus de quelque chose en préférant zéro, cela ne veut pas dire qu'il correspondra toujours à zéro caractère. Si c'était le cas ici, par exemple, et qu'il laissait `$1` à la valeur `""`, alors le second « `e` » ne serait pas trouvé, puisqu'il ne suit pas immédiatement le premier.

Vous pourriez aussi vous demander pourquoi, alors qu'il cherchait une correspondance minimale avec `/e(.*?)e/`, Perl n'a pas plutôt mis « `nc` » dans `$1`. Après tout, « `nc` » se trouve lui aussi entre deux `e` et est plus court que « `xig` ». En Perl, le choix minimal/maximal se fait seulement lorsqu'on recherche la plus courte ou la plus longue parmi plusieurs correspondances ayant toutes le même point de départ. Si plusieurs correspondances possibles existent, mais commencent à des endroits différents dans la chaîne, alors leurs longueurs n'ont aucune importance — pas plus que le fait que vous ayez utilisé un quantificateur minimal ou maximal. La première de plusieurs correspondances possibles est toujours prépondérante devant celles qui la suivent. C'est seulement lorsque plusieurs correspondances possibles démarrent du même point dans la chaîne que le côté minimal ou maximal de la correspondance sert à les départager. Si les points de départ sont différents, il n'y a rien à départager. Les correspondances de Perl sont normalement de type *la plus à gauche la plus longue* ; avec une correspondance minimale, cela devient *la plus à gauche la plus courte*. Mais la partie *la plus à gauche* ne change pas et reste le critère dominant.[8]

Il y a deux manières de passer l'orientation à gauche du détecteur de motif. Premièrement vous pouvez utiliser un quantificateur gourmand au début (typiquement `.*`) pour essayer de consommer le début de la chaîne. En cherchant la correspondance pour un quantificateur gourmand, il essaie d'abord la correspondance la plus longue ce qui provoque effectivement une recherche de droite à gauche dans le reste de la chaîne :

```
"exigence" =~ /.*e(.*?)e/   # $1 vaut maintenant "nc"
```

Mais faites attention avec cette méthode, car la correspondance complète contient maintenant toute la chaîne jusqu'à ce point.

8. Tous les moteurs de regex ne fonctionnent pas comme cela. Certains croient plutôt en la gourmandise globale, où la correspondance la plus longue gagne toujours, même si elle apparaît plus tard dans la chaîne. Perl ne fonctionne pas comme cela. Vous pourriez dire que l'ardeur prime sur la gourmandise (ou le rationnement). Pour une discussion plus formelle de ce principe et de beaucoup d'autres, voir la section « *Le petit Moteur qui /(ne)?pouvait(pas)?/* ».

La seconde manière de défaire cette inclination à gauche est d'utiliser les assertions de position, qui sont discutées dans la section suivante.

Positions

Certaines constructions de regex représentent des *positions* dans la chaîne à traiter, qui est juste un emplacement à gauche ou à droite d'un véritable caractère. Ces métasymboles sont des exemples d'assertions de *largeur nulle*. Nous les appellerons souvent simplement « assertions ». (On les connait aussi sous le nom d'« ancres », car elles lient une partie du motif à une position particulière.)

Vous pouvez toujours manipuler les positions dans une chaîne sans utiliser de motif. La fonction intégrée `substr` vous permet d'extraire et d'affecter des sous-chaînes, calculées à partir du début de la chaîne, de la fin de la chaîne ou à partir d'une position numérique particulière. C'est juste ce dont vous avez besoin si vous travaillez avec des enregistrements de longueur fixe. Les motifs deviennent nécessaires quand un décalage numérique n'est plus suffisant. Et la plupart du temps, les décalages ne sont pas suffisants — disons pas suffisament commodes, par rapport aux motifs.

Les débuts : assertions \A et ^

L'assertion `\A` correspond seulement au début de la chaîne, quelles que soient les circonstances. Cependant l'assertion `^` est l'assertion traditionnelle de début de ligne, ainsi que celle de début de chaîne. C'est pourquoi si le motif utilise le modificateur `/m`[9] et que la chaîne contient des sauts de ligne, `^` correspond également n'importe où dans la chaîne immédiatement après un caractère de saut de ligne :

```
/\Abar/      # Correspond à "bar" et "barbecue"
/^bar/       # Correspond à "bar" et "barbecue"
/^bar/m      # Correspond à "bar" et "barbecue" et "zinc\nbar"
```

Utilisée en même temps que `/g`, le modificateur `/m` permet à `^` de correspondre plusieurs fois dans la même chaîne :

```
s/^\s+//gm;             # Supprime les blancs en tête de chaque ligne
$total++ while /^./mg;  # Compte les lignes qui ne sont pas vides
```

Fins : assertions \z, \Z et $

Le métasymbole `\z` correspond à la fin de la chaîne, quoi qu'il y ait à l'intérieur. `\Z` correspond juste avant le saut de ligne en fin de chaîne s'il y en a un, ou à la fin s'il n'y en a pas. Le métacaractère `$` signifie en général la même chose que `\Z`. Cependant, si le modificateur `/m` a été spécifié et que la chaîne contient des sauts de ligne, alors `$` peut aussi correspondre n'importe où dans la chaîne, juste avant un saut de ligne :

```
/bot\z/      # Correspond avec "robot"
/bot\Z/      # Correspond avec "robot" et "poulbot\n"
```

9. Ou si vous avez mis la variable obsolète `$*` à `1` et que vous n'ignorez pas `$*` avec le modificateur `/s`.

```
/bot$/       # Correspond avec "robot" et "poulbot\n"
/bot$/m      # Correspond avec "robot" et "poulbot\n" et "robot\nménager"

/^robot$/    # Correspond avec "robot" et "robot\n"
/^robot$/m   # Correspond avec "robot" et "robot\n" et "ce\nrobot\n"
/\Arobot\Z/  # Correspond avec "robot" et "robot\n"
/\Arobot\z/  # Correspond avec "robot" uniquement -- mais pourquoi ne
             #pas avoir utilisé eq ?
```

Tout comme avec ^, le modificateur /mpermet à $ de correspondre plusieurs fois dans la même chaîne quand il est utilisé avec /g. (Ces exemples supposent que vous avez lu un enregistrement multiligne dans $_, par exemple en mettant $/ à "" avant la lecture.)

```
s/\s*$//gm;   # Supprime les blancs en fin de chaque ligne du paragraphe

while (/^([^:]+):\s*(.*)/gm ) {  # obtient les en-têtes de mail
    $headers{$1} = $2;
}
```

Plus loin dans ce chapitre, à la section *Interpolation de variables*, nous discuterons de la façon dont vous pouvez interpoler des variables dans des motifs : si $toto vaut « bc », alors /a$toto/ est équivalent à /abc/. Ici, le $ ne correspond pas à la fin de la chaîne. Pour qu'un $ corresponde à la fin de la chaîne, il doit être à la fin du motif ou être immédiatement suivi d'une barre verticale ou d'une parenthèse fermante.

Limites : assertions \b et \B

L'assertion \b correspond à toute limite de mot, celle-ci étant définie comme la position entre un caractère \w et un caractère \W, dans n'importe quel ordre. Si l'ordre est \W\w c'est la limite en début de mot, et si l'ordre est \w\W c'est la limite en fin de mot. (Une extrémité de chaîne compte comme un caractère \W ici.) L'assertion \B correspond à toute position qui *n'est pas* une limite de mot, c'est-à-dire au milieu de \w\w ou de \W\W.

```
/\best\b/   # correspond à "ce qu'il est" et "à l'est d'Eden"
/\Best\B/   # correspond à "zeste" et "intestin"
/\best\B/   # correspond à "estival" et "il t'estime"
/\Best\b/   # correspond à "al         et "ouest puis nord"
```

Comme \W comprend tous les caractères de ponctuation, (sauf le souligné), il y a des limites \b au milieu de chaînes comme « aujourd'hui », « booktech@oreilly.com », « S.N.C.F. » et « clé/valeur ».

À l'intérieur d'une classe de caractères ([\b]), un \b représente un caractère espace arrière plutôt qu'une limite de mot.

Reconnaissance progressive

Utilisée avec le modificateur /g, la fonction pos vous permet de connaître ou de modifier la position à partir de laquelle la prochaine comparaison progressive commencera :

```
$voleur = "Bilbon Baggins";
while ($voleur =~ /b/gi) {
    printf "Trouvé un B en %d\n", pos($voleur)-1;
}
```

(Nous retirons un de la position car c'est la longueur de la chaîne que nous voulions, alors que pos renvoie toujours la position juste après la fin de la reconnaissance précédente.)

Le code ci-dessus affiche :

```
Trouvé un B en 0
Trouvé un B en 3
Trouvé un B en 7
```

Après un échec, la position de reconnaissance est normalement remise à zéro. Si vous utilisez le modificateur /c (pour « continue »), alors quand le /g se termine, l'échec de la reconnaissance ne réinitialise pas le pointeur de position. Cela vous permet de continuer votre recherche au-delà de ce point, sans recommencer à partir du début.

```
$voleur = "Bilbon Baggins";
while ($voleur =~ /b/gci) {        # AJOUT DE /c
    printf "Trouvé un B en %d\n", pos($voleur)-1;
}
while ($voleur =~ /i/gi) {
    printf "Trouvé un I en %d\n", pos($voleur)-1;
}
```

En plus des trois B trouvés précédemment, Perl indique maintenant qu'il a trouvé un i à la position 11. Sans le /c, la seconde boucle de reconnaissance aurait recommencé depuis le début de la chaîne et d'abord trouvé un autre i à la position 1.

Là où vous en étiez : assertion \G

Dès que vous commencez à réfléchir en termes de la fonction pos, il est tentant de commencer à creuser dans votre chaîne à coups de substr, mais c'est rarement la bonne chose à faire. La plupart du temps, si vous avez commencé avec une recherche de motif, vous devriez continuer avec une recherche de motif. Cependant, si vous recherchez une assertion de position, c'est probablement \G qu'il vous faut.

L'assertion \G représente à l'intérieur du motif le même point que pos représente à l'extérieur. Quand vous faites une recherche progressive avec le modificateur /g (ou que vous avez utilisé la fonction pos pour sélectionner directement le point de départ), vous pouvez utiliser \G pour spécifier la position qui suit la fin de la reconnaissance précédente. C'est-à-dire qu'il reconnaît la position immédiatement avant le caractère qui serait identifié par pos. Cela vous permet de vous souvenir où vous en étiez :

```
($recette = <<'DISH') =~ s/^\s+//gm;
    # Anchois sauce Roswel
    Préchauffer le four à 451 deg. fahrenheit.
    Mélanger 1 ml. de dilithium avec 3 oz. de
    NaCl et y plonger 4 anchois. Glacer avec
    1 gr. de mercure. Faire cuire 4 heures et
    laisser refroidir 3 secondes. Pour 10 martiens.
DISH

$recette =~ /\d+ /g;
$recette =~ /\G(\w+)/;          # $1 vaut "deg"
```

```
$recette =~ /\d+ /g;
$recette =~ /\G(\w+)/;          # $1 vaut "ml"
$recette =~ /\d+ /g;
$recette =~ /\G(\w+)/;          # $1 vaut "gr"
```

Le métasymbole \G est souvent utilisé dans une boucle, comme nous allons le monrer dans notre prochain exemple. Nous faisons une « pause » après chaque suite de chiffres, et à cette position, nous testons s'il y a une abréviation. Si oui, nous récupérons les deux mots qui suivent. Sinon, nous récupérons seulement le mot suivant :

```
pos($recette) = 0;                      # Par sécurité, initialise \G à 0
while ( $recette =~ /(\d+) /g ) {
    my $quantite = $1;
    if ($recette =~ / \G (\w{0,3}) \. \s+ de \s+ (\w+) /x) { # abrév. + mot
        print "$quantite $1 de $2\n";
    } else {
        $recette =~ / \G (\w+) /x;                          # juste un mot
        print "$quantite $1\n";
    }
}
```

Ce qui donne :

```
451 deg
1 ml de dilithium
3 oz de NaCl
4 anchois
1 gr de mercure
4 heures
3 secondes
10 martiens
```

Capture et regroupement

Les motifs vous permettent de regrouper des portions de votre motif dans des sous-motifs et de vous souvenir des chaînes reconnues par ces sous-motifs. Nous appellerons le premier comportement *regroupement* et le second *capture*.

Capture

Pour capturer une sous-chaîne pour une utilisation ultérieure, mettez des parenthèses autour du motif qui la reconnaît. La première paire de parenthèses stocke sa sous-chaîne dans $1, la deuxième paire dans $2 et ainsi de suite. Vous pouvez utilisez autant de parenthèses que vous voulez ; Perl continuera à définir des variables numérotées pour que vous puissiez représenter ces chaînes capturées.

Quelques exemples :

```
/(\d)(\d)/ # Trouve deux chiffres, qui sont capturés dans $1 et $2
/(\d+)/    # Trouve un ou plusieurs chiffres, capturés ensemble dans $1
/(\d)+/    # Trouve un chiffre une fois ou plus, et capture le dernier dans $1
```

Remarquez la différence entre le deuxième et le troisième motifs. La deuxième forme

est en général ce que vous voulez. La troisième forme *ne crée pas* plusieurs variables pour plusieurs chiffres. Les parenthèses sont numérotées quand le motif est compilé, pas quand il est utilisé.

Les chaînes capturées sont souvent appelées *références arrières* car elles font référence à des parties du texte situées en arrière par rapport à la position courante. Il existe en fait deux manières d'utiliser ces références arrières. Les variables numérotées que vous avez vues donnent accès en dehors du motif aux références arrières, mais à l'intérieur du motif, cela ne marche pas. Vous devez utiliser `\1`, `\2`, etc.[10] Pour trouver les mots doublés comme « `le le` » ou « `est est` », vous pourriez utilisez ce motif :

```
/\b(\w+) \1\b/i
```

Mais la plupart du temps, vous utiliserez la forme `$1`, car en général on applique un motif, pour ensuite faire quelque chose des sous-chaînes. Supposons que vous ayez du texte (des en-têtes de mail) qui ressemble à ça :

```
From: gnat@perl.com
To: camelot@oreilly.com
Date: Mon, 17 Jul 2000 09:00:00 -1000
Subject: Eye of the needle
```

et que vous vouliez construire un hachage qui fasse le lien entre le texte avant les deux-points et celui après. Si vous boucliez sur ce texte ligne à ligne (par exemple parce que vous lisez un fichier), vous pourriez faire comme suit :

```
while (<>) {
    /^(.*?): (.*)$/;    # Texte avant les deux-points dans $1, après dans $2
    $champs{$1} = $2;
}
```

Comme `$`` , `$&` et `$'`, ces variables numérotées sont à portée dynamique jusqu'à la fin du bloc ou de la chaîne `eval` englobant, ou jusqu'à la prochaine recherche de motif réussie, selon lequel se produit le premier. Vous pouvez également les utiliser dans la partie droite (la zone de remplacement) d'une substitution :

```
s/^(\S+) (\S+)/$2 $1/;  # Intervertit les deux premiers mots
```

Les regroupements peuvent s'emboîter, et quand ils le font, les groupes sont comptés par l'ordre de leurs parenthèses ouvrantes. Donc, si on donne la chaîne « Primula Brandebouc » au motif :

```
/^((\w+) (\w+))$/
```

il capturerait « `Primula Brandebouc` » dans `$1`, « `Primula` » dans `$2`, et « `Brandebouc` » dans `$3`. Cela est décrit à la *figure 5-1*.

10. Vous ne pouvez pas utiliser `$1` pour une référence arrière à l'intérieur d'un motif, car il aurait déjà été interpolé au moment où la regex a été compilée. C'est pourquoi nous utilisons la notation traditionnelle `\1` pour les références arrières à l'intérieur d'un motif. Pour les références à deux ou trois chiffres, il y a une ambiguïté avec la notation octale des caractères, mais c'est finement résolu en regardant combien de motifs capturés sont disponibles. Par exemple, si Perl voit un métasymbole `\11`, c'est équivalent à `$11` seulement s'il y a au moins 11 sous-chaînes capturés plus tôt dans le motif. Sinon c'est équivalent à `\011`, c'est-à-dire un caractère de tabulation.

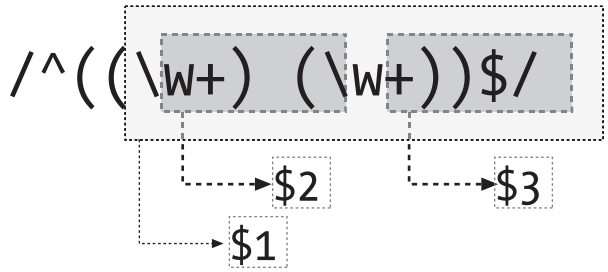


Figure 5-1. Création de références arrières avec des parenthèses

Les motifs avec capture sont souvent utilisés en contexte de liste pour remplir une liste de valeurs, car le motif est assez malin pour retourner les sous-chaînes comme une liste :

```
($premier, $dernier)        =  /^(\w+) (\w+)$/;
($tout, $premier, $dernier) =  /^((\w+) (\w+))$/;
```

Avec le modificateur /g, un motif peut renvoyer plusieurs sous-chaînes issues de plusieurs correspondances, le tout dans une seule liste. Supposons que vous ayez l'en-tête que nous avons vu précédemment dans une seule chaîne (disons dans $_). Vous pourriez faire la même chose que notre boucle ligne à ligne avec une seule instruction :

```
%champs = /^(.*?): (.*)$/gm;
```

Le motif réussit quatre fois, et à chaque fois il trouve deux sous-chaînes. La recherche /gm retourne l'ensemble comme une liste plate de huit chaînes, que l'affectation de liste à %champs interprètera comme quatre couples clé/valeur, ramenant ainsi l'harmonie dans l'univers.

Plusieurs autres variables spéciales traitent du texte capturé dans des recherches de motif. $& contient la chaîne trouvée dans son ensemble, $` tout ce qui est à gauche de la correspondance, $' tout ce qui est à droite. $+ garde le contenu de la dernière référence arrière.

```
$_ = "Parlez, <EM>ami</EM>, et entrez.";
m[ (<.*?>) (.*?) (</.*?>) ]x;     # Une balise, du texte, et une balise de
fin
print "prematch: $`\n";           # Parlez,
print "match: $&\n";              # <EM>ami</EM>
print "postmatch: $'\n";          # , et entrez.
print "lastmatch: $+\n";          # </EM>
```

Pour plus d'explication sur ces variables elfiques magiques (et pour savoir comment les écrire en anglais), voir le chapitre 28, *Noms spéciaux*.

Le tableau @- (@LAST_MATCH_START) contient les positions des débuts de chacunes des correspondances, et @+ (@LAST_MATCH_END) contient les positions de leurs fins :

```
#!/usr/bin/perl
$alphabet = "abcdefghijklmnopqrstuvwxyz";
$alphabet =~ /(hi).*(stu)/;

print "La recherche complète a commencé en $-[0] et s'est terminée en
$+[0]\n";
```

```
print "La première recherche a commencé en $-[1] et s'est terminée en
$+[1]\n";
print "La deuxième recherche a commencé en $-[2] et s'est terminée en
$+[2]\n";
```

Si vous voulez trouver une parenthèse littérale plutôt que de l'interpréter comme un métacaractère, antislashez-la :

```
/\(p.e., .*?\)/
```

Ceci trouve un exemple entre parenthèses (p.e., celui-ci). Mais comme le point est un joker, cela trouve aussi toute phrase entre parenthèses dont la première lettre est un `p` et la troisième un `e` (pied, par exemple).

Regroupement

Les parenthèses simples regroupent et capturent à la fois. Mais parfois vous n'en demandez pas tant. Parfois vous voulez juste regrouper des portions du motif sans créer de référence arrière. Vous pouvez utiliser une forme étendue de parenthèses pour supprimer la capture : la notation `(?:`*`PATTERN`*`)` va *regrouper* sans capturer.

Il y a au moins trois raisons pour lesquelles vous voudrez regrouper sans capturer :

1. Pour quantifier quelque chose.
2. Pour limiter la portée d'une alternative ; par exemple, `/^chat|chien|vache$/` aura besoin d'être écrit `/^(?:chat|chien|vache)$` si vous ne voulez pas que le chat file avec le `^`.
3. Pour limiter la portée d'un modificateur de motif à un sous-motif spécifique, comme dans `/toto(?-i:Attention_A_La_Casse)titi/i`. (Voir la section suivante, *Modificateurs cloîtrés*.)

De plus, il est plus efficace de supprimer la capture de quelque chose que vous n'allez pas utiliser. En revanche, la notation est un peu moins lisible.

Dans un motif, une parenthèse ouvrante immédiatement suivie d'un point d'interrogation indique une *extension* de regex. Le bestiaire actuel des expressions régulières est relativement fixé — nous n'osons pas créer un nouveau métacaractère, de peur de casser d'anciens programmes Perl. Au lieu de cela, la syntaxe d'extension permet d'ajouter de nouvelles fonctionnalités au bestiaire.

Dans le reste du chapitre, nous allons voir beaucoup d'autres extensions de regex, lesquelles feront toutes du regroupement sans capture, avec quelque chose d'autre. L'extension `(?:`*`MOTIF`*`)` est spéciale en ce qu'elle ne fait rien d'autre. Donc si vous écrivez :

```
@champs = split(/\b(?:a|b|c)\b/)
```

c'est comme :

```
@champs = split(/\b(a|b|c)\b/)
```

sauf que cela ne renvoie pas de champs supplémentaires. (L'opérateur `split` est un peu comme `m//g` en ce qu'il retournera des champs supplémentaires pour chaque chaîne capturée dans le motif. D'ordinaire, `split` ne retourne que ce qui n'a pas été trouvé. Pour en savoir plus sur `split`, voir le chapitre 29.)

Modificateurs cloîtrés

Vous pouvez *cloîtrer* les modificateurs `/i`, `/m`, `/s` et `/x` dans une portion de votre motif en les insérant (sans le slash) entre le `?` et le `:` de la notation de regroupement. Si vous écrivez :

```
/Harry (?i:s) Truman/
```

cela correspondra à la fois à « `Harry S Truman` » et à « `Harry s Truman` », tandis que :

```
/Harry (?x: [A-Z] \.? \s )?Truman/
```

correspondra à la fois à « `Harry S Truman` » et « `Harry S. Truman` », aussi bien qu'à « `Harry Truman` », et :

```
/Harry (?ix: [A-Z] \.? \s )?Truman/
```

trouvera les cinq, en combinant les modificateurs `/i` et `/x` dans le cloître.

Vous pouvez également soustraire des modificateurs d'un cloître avec un signe moins :

```
/Harry (?x-i: [A-Z] \.? \s )?Truman/i
```

Ceci trouve le nom dans toutes les combinaisons de majuscules et minuscules — mais si l'initiale centrale est fournie, elle doit être en majuscules, car le `/i` appliqué au motif dans son ensemble est suspendu à l'intérieur du cloître.

En omettant les deux-points et le *`MOTIF`*, vous pouvez exporter les réglages des modificateurs à un regroupement extérieur, en le transformant en cloître. C'est-à-dire que vous pouvez sélectivement activer ou désactiver les modificateurs pour le regroupement qui se trouve un niveau à l'extérieur des parenthèses du modificateur, comme ceci :

```
/(?i)toto/             # Équivalent à /toto/i
/toto((?-i)titi)/i     # "titi" doit être en minuscules
/toto((?x-i) titi)/    # Active /x et désactive /i pour "titi"
```

Remarquez que le deuxième et le troisième exemples créent des références arrières. Si ce n'était pas ce que vous vouliez, alors vous auriez dû utiliser respectivement `(?-i:titi)` et `(?x-i: titi)`.

Activer les modificateurs sur une partie de votre motif est particulièrement utile quand vous voulez que « `.` » corresponde à des sauts de ligne dans une partie de votre motif, mais pas dans le reste. Activer `/s` sur tout le motif ne vous sera d'aucune utilité dans ce cas.

Alternative

À l'intérieur d'un motif ou d'un sous-motif, utilisez le métacaractère `|` pour spécifier un ensemble de possibilités dont chacune peut correspondre. Par exemple :

```
/Gandalf|Saroumane|Radagaste/
```

correspond à `Gandalf`, `Saroumane` ou `Radagaste`. L'alternative s'étend seulement jusqu'aux parenthèses qui l'entourent les plus proches (qu'elles capturent ou non) :

```
/perc|l|s|tes/      # Trouve perc, l, s ou tes
/per(c|l|s|t)es/    # Trouve perces, perles, perses ou pertes
/per(?:c|l|s|t)es/  # Trouve perces, perles, perses ou pertes
```

La deuxième et la troisième formes correspondent aux mêmes chaînes, mais la deuxième capture le caractère changeant, tandis que la troisième ne le fait pas.

Dans toutes les positions, le Moteur essaie de trouver le premier élément de l'alternative, puis le deuxième, et ainsi de suite. La longueur de chaque élément n'a pas d'importance, ce qui signifie que dans ce motif :

```
/(Sam|Samsagace)/
```

`$1` ne vaudra jamais `Samsagace`, quelle que soit la chaîne sur laquelle on l'utilise, car `Sam` sera toujours trouvé en premier. Quand vous avez des recherches qui se superposent comme ceci, mettez les plus longues au début.

Mais l'ordre des éléments de l'alternative n'a d'importance qu'à une position donnée. La boucle externe du Moteur fait une recherche de gauche à droite, c'est pourquoi ce qui suit trouve toujours le premier `Sam` :

```
"'Sam je suis,' dit Samsagace" =~ /(Samsagace|Sam)/;   # $1 eq "Sam"
```

Mais vous pouvez forcer une recherche de droite à gauche en utilisant les quantificateurs gourmands, comme nous l'avons vu précédemment dans *Quantificateurs* :

```
"'Sam je suis,' dit Samsagace" =~ /.*(Samsagace|Sam)/; # $1 eq "Samsagace"
```

Vous pouvez défaire toute recherche gauche à droite (ou de droite à gauche) en incluant n'importe laquelle des assertions de position vues précédemment, comme `\G`, `^` et `$`. Ici nous ancrons le motif à la fin de la chaîne :

```
"'Sam je suis,' dit Samsagace" =~ /(Samsagace|Sam)$/;  # $1 eq "Samsagace"
```

Cet exemple met le `$` en facteur en dehors de l'alternative (car nous avions déjà une paire de parenthèses en dehors de laquelle le mettre), mais en l'absence de parenthèses vous pouvez également distribuer les assertions sur tout ou partie des éléments de l'alternative, selon comment vous voulez qu'ils trouvent. Ce petit programme affiche les lignes qui commencent par un token `__DATA__` ou `__END__` :

```
#!/usr/bin/perl
while (<>) {
    print if /^__DATA__|^__END__/;
}
```

Mais faites attention avec cela. Souvenez-vous que le premier et le dernier éléments de l'alternative (avant le premier `|` et après le dernier) ont tendance à avaler les autres de chaque côté, sauf s'il y a des parenthèses autour. Une erreur courante est de demander :

```
/^chat|chien|vache$/
```

quand vous voulez en fait :

```
/^(chat|chien|vache)$/
```

La première trouve « `chat` » au début de la chaîne, ou « `chien` » n'importe où, ou bien « `vache` » en fin de chaîne. La deuxième trouve n'importe quelle chaîne consistant simplement de « `chat` », « `chien` » ou « `vache` ». Elle capture également `$1`, ce qui n'est pas forcément ce que vous voulez. Vous pouvez également écrire :

```
/^chat$|^chien$|^vache$/
```

Nous vous montrerons une autre solution plus loin.

Un élément de l'alternative peut être vide, auquel cas il réussit toujours.

```
/com(posite|)/;      # Trouve "composite" ou "com"
/com(posite(s|)|)/;  # Trouve "composites", "composite" ou "com"
```

Cela revient à utiliser le quantificateur ?, qui trouve 0 ou 1 fois :

```
/com(posite)?/;      # Trouve "composite" ou "com"
/com(posite(s?))?/;  # Trouve "composites", "composite" ou "com"
/com(posites?)?/;    # Pareil, mais n'utilise pas $2
```

Il y a cependant une différence. Quand vous appliquez le ? à un sous-motif qui fait une capture dans une variable numérotée, cette variable sera indéfinie s'il n'y a pas de chaîne à y mettre. Si vous avez mis l'élément vide dans l'alternative, elle serait toujours fausse, mais ce serait en fait une chaîne vide définie.

Garder le contrôle

Comme le sait tout bon chef d'équipe, cela ne sert à rien d'être toujours sur le dos de vos employés. Dites-leur simplement ce que vous voulez, et laissez-les trouver la meilleure manière de le faire. De même, la meilleure manière de voir une expression régulière, c'est comme une spécification : « Voilà ce que je veux ; trouve-moi une chaîne qui correspond ».

D'un autre côté, les meilleurs chefs d'équipe comprennent ce que leurs employés essayent de faire. C'est également vrai pour les expressions régulières en Perl. Mieux vous comprenez comment Perl s'acquitte de la tâche de trouver un motif particulier, plus vous serez capable d'utiliser les capacités de recherche de motif de Perl en connaissance de cause.

Une des choses les plus importantes à comprendre à propos de la recherche de motif en Perl, c'est quand *ne pas* l'utiliser.

Laisser Perl faire le boulot

Quand certaines personnes apprennent les expressions régulières, elles sont souvent tentées de voir tout problème comme un problème de recherche de motif. Et quand bien même ce serait vrai en général, la recherche de motif est plus que la simple évaluation d'expressions régulières. C'est aussi comme rechercher vos clés de voiture où vous les avez fait tomber, et pas seulement sous le réverbère, là où vous y voyez le mieux. Dans le monde réel, nous savons tous qu'il est plus efficace de chercher aux bons endroits qu'aux mauvais.

De même, vous devriez utiliser le contrôle de flux de Perl pour décider quels motifs exécuter et lesquels ignorer. Une expression régulière est plutôt maligne, mais pas plus qu'un cheval. Elle peut être distraite si elle voit trop de choses à la fois. C'est pourquoi il vous faut parfois lui mettre des œillères. Par exemple, souvenez-vous de notre précédent exemple d'alternative :

```
/Gandalf|Saroumane|Radagaste/
```

Cela marche comme prévu, mais pas aussi bien que possible, car elle cherche chaque nom à chaque position dans la chaîne avant de passer à la suivante. Les lecteurs attentifs du *Seigneur des anneaux* se souviendront que, des trois magiciens mentionnés ci-dessus, Gandalf est mentionné plus souvent que Saroumane, et que Saroumane est mentionné

plus souvent que Radagaste. Il est donc en général plus efficace d'utiliser les opérateurs logiques de Perl pour réaliser l'alternative :

```
/Gandalf/ || /Saroumane/ || /Radagaste/
```

C'est également une autre manière de passer outre la recherche « orientée à gauche » du Moteur. Il ne recherche `Saroumane` que si `Gandalf` est introuvable. Et il ne cherche `Radagaste` que si `Saroumane` est également absent.

Non seulement cela change l'ordre de recherche des différents termes, but cela permet également à l'optimiseur d'expressions régulières de mieux fonctionner. Il est en général plus simple d'optimiser la recherche pour une seule chaîne que pour plusieurs à la fois. De même, les recherches ancrées peuvent souvent être optimisées si elles ne sont pas trop compliquées.

Vous n'avez pas à vous limiter à l'opérateur || pour le contrôle de flux. Vous pouvez souvent contrôler les choses au niveau des instructions. Vous devriez toujours penser à vous débarrasser des cas les plus courants en premier. Supposons que vous écriviez une boucle pour traiter un fichier de configuration. La plupart des fichiers de configuration sont composés principalement de commentaires. C'est souvent mieux de supprimer les commentaires et les lignes blanches avant de s'attaquer aux lourds traitements, même si les lourds traitements supprimeraient également les commentaires et lignes blanches en cours de route :

```
while (<CONF>) {
    next if /^#/;
    next if /^\s*(#|$)/;
    chomp;
    meugneumeugneu($_);
}
```

Même si vous n'essayez pas d'être efficace, vous avez souvent besoin d'alterner entre les expressions Perl normales et les expressions régulières, simplement parce que vous voulez faire quelque chose qui n'est pas possible (ou qui est très difficile) à l'intérieur d'une expression régulière, comme afficher quelque chose. Voici un classificateur de nombres bien utile :

```
warn "contient des non-chiffres" if     /\D/;
warn "pas un nombre naturel"     unless /^\d+$/;             # rejette -3
warn "pas un entier"             unless /^-?\d+$/;           # rejette +3
warn "pas un entier"             unless /^[+-]?\d+$/;
warn "pas un nombre décimal"     unless /^-?\d+\.?\d*$/;     # rejette .2
warn "pas un nombre décimal"     unless /^-?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "pas un flottant de C"
       unless /^([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+))?$/;
```

Nous pourrions encore beaucoup allonger cette section, mais ceci est vraiment le sujet de tout le livre. Vous verrez encore beaucoup d'exemples mélant du code Perl et des recherches de motif au long du livre. En particulier, reportez-vous à la section suivante *Motifs programmatiques*. (Mais vous pouvez quand même commencer par lire ce qui suit, bien sûr.)

Interpolation de variables

L'utilisation des mécanismes de contrôle de flux de Perl pour contrôler les expressions régulières a ses limites. La difficulté principale est qu'il s'agit d'une approche « tout ou rien » ; soit vous faites tourner le motif, soit vous ne le faites pas tourner. Parfois vous savez vaguement ce que vous cherchez, mais vous aimeriez avoir la possibilité de le paramétrer. L'interpolation de variables fournit cette capacité, tout comme le paramètrage des sous-programmes vous permet d'avoir une plus grande influence sur leur comportement que simplement de choisir entre les appeler ou non. (Vous en verrez plus sur les sous-programmes au chapitre suivant.)

Un des intérêts de l'interpolation est de fournir un peu d'abstraction, ainsi qu'un peu de lisibilité. Avec des expressions régulières, il est certes possible d'écrire les choses de façon concise :

```
if ($num =~ /^[-+]?\d+\.?\d*$/) { ... }
```

Mais ce que vous voulez dire est plus compréhensible ainsi :

```
$signe = '[-+]?';
$chiffres = '\d+';
$point_decimal = '\.?';
$encore_des_chiffres = '\d*';
$nombre = "$signe$chiffres$point_decimal$encore_des_chiffres";
...
if ($num =~ /^$nombre$/o) { ... }
```

Nous couvrirons un peu plus cette utilisation de l'interpolation dans la section *Motifs générés*, plus loin dans ce chapitre. Notez juste que nous avons utilisé le modificateur `/o` pour supprimer la recompilation du motif, car nous ne nous attendons pas à ce que `$nombre` change de valeur au cours du programme.

Un autre truc sympa est de renverser vos tests et de vous servir de la chaîne variable comme motif vis-à-vis d'un ensemble de chaînes connues :

```
chomp($reponse = <STDIN>);
if    ("SEND"  =~ /^\Q$reponse/i) { print "Action : send\n"  }
elsif ("STOP"  =~ /^\Q$reponse/i) { print "Action : stop\n"  }
elsif ("ABORT" =~ /^\Q$reponse/i) { print "Action : abort\n" }
elsif ("LIST"  =~ /^\Q$reponse/i) { print "Action : list\n"  }
elsif ("EDIT"  =~ /^\Q$reponse/i) { print "Action : edit\n"  }
```

Ceci permet à l'utilisateur de lancer l'action « send » en tapant `S`, `SE`, `SEN` ou `SEND` (dans n'importe quel mélange de majuscules et minuscules). Pour faire « stop », il lui faudra au moins taper `ST` (ou `St`, `sT` ou `st`).

Retour de l'antislash

Quand vous pensez à l'interpolation entre apostrophes doubles, vous pensez généralement à l'interpolation des variables et des antislashs. Mais comme nous l'avons indiqué précédemment, il y a deux passes pour les expressions régulières, et la passe d'interpolation laisse la plupart du travail d'interprétation des antislashs à l'analyseur d'expressions régulières (dont nous parlerons plus tard). D'ordinaire, vous ne remarquerez pas de différence, car Perl fait bien attention à les masquer. (Une séquence qui est évidem-

ment différente est le métasymbole `\b`, qui se transforme en assertion de limite — en dehors des classes de caractères, en tout cas. À l'intérieur d'une classe de caractères, où les assertions n'ont aucun sens, il redevient un espace arrière, comme d'habitude.)

Il est en fait assez important que l'analyseur de regex gère les antislashs. Imaginons que vous cherchiez les caractères de tabulation avec un motif sous le modificateur `/x` :

```
($col1, $col2) = /(.*?) \t+ (.*?)/x;
```

Si Perl ne laissait pas l'interpolation de `\t` à l'analyseur de regex, le `\t` se serait transformé en un blanc que l'analyseur de regex aurait bêtement ignoré à cause du `/x`. Mais Perl n'est ni si ignoble, ni si piégeant.

Vous pouvez cependant vous piéger tout seul. Imaginons que vous ayez un peu virtualisé le séparateur de colonne, comme ceci :

```
$sepcol = "\t+";                          # (apostrophes doubles)
($col1, $col2) = /(.*?) $sepcol (.*?)/x;
```

Et maintenant vous voilà fait comme un rat, car le `\t` se transforme en véritable tabulation avant d'atteindre l'analyseur de regex, qui croira que vous avez écrit `/(.*?)+(.*?)/` une fois les blancs supprimés. Oups. Pour corriger cela, évitez `/x` ou bien utilisez des apostrophes simples. Ou mieux, utilisez `qr//`. (Voir la section suivante.)

Les seules séquences d'échappement entre apostrophes doubles qui sont traitées comme telles sont les six séquences de traduction : `\U`, `\u`, `\L`, `\l`, `\Q` et `\E`. Si jamais vous vous plongez un jour dans le fonctionnement interne du compilateur d'expressions régulières, vous y trouverez du code destiné à gérer les séquences d'échappement comme `\t` pour la tabulation, `\n` pour le saut de ligne, et ainsi de suite. Mais vous n'y trouverez pas de code pour ces six séquences de traduction. (Nous avons listées dans le *tableau 5-7* uniquement car c'est là qu'on s'attend à les trouver.) Si vous vous débrouillez pour les inclure dans le motif sans passer par l'évaluation entre apostrophes doubles, elles ne seront pas reconnues.

Comment ont-elles pu s'y glisser ? Vous pouvez contrecarrer l'interpolation en utilisant des apostrophes simples comme délimiteurs de motifs. Les apostrophes simples suppriment l'interpolation de variables et le traitement des séquences de traduction dans `m'...'`, `qr'...'` et `s'...'...'`, comme ils le feraient dans une chaîne entre apostrophes simples. Écrire `m'\ufrodon'` ne trouvera pas une version en majuscules de ce pauvre frodon. Cependant, comme les antislash « normaux » ne sont pas vraiment traités à ce niveau, `m'\t\d'` trouvera toujours une vraie tabulation suivie d'un chiffre quelconque.

Une autre manière de contrecarrer l'interpolation est de passer par l'interpolation elle-même. Si vous écrivez :

```
$var = '\U';
/${var}frodon/;
```

ce pauvre frodon reste en minuscules. Perl ne refera pas la passe d'interpolation pour vous simplement parce que vous avez interpolé quelque chose qui a l'air de vouloir être interpolé. Vous ne pouvez pas plus espérer que cela soit interpolé que vous ne pourriez attendre que cette double interpolation marche :

```
$hobbit = 'Frodon';
$var = '$hobbit';          # (apostrophes simples)
/$var/;                    # signifie m'$hobbit', et pas m'Frodon'.
```

Voici un autre exemple qui montre comment les antislashs sont interprétés par l'analyseur de regex et pas par l'interpolation de variables. Imaginons que vous ayez un petit utilitaire simple à la *grep* :[11]

```
#!/usr/bin/perl
$motif = shift;
while (<>) {
    print if /$motif/o;
}
```

Si vous appelez ce programme *pgrep* et l'invoquez comme suit :

```
% pgrep '\t\d' *.c
```

alors vous découvrirez qu'il affiche toutes les lignes de tous vos fichiers source C dans lesquelles un chiffre suit une tabulation. Vous n'avez rien eu à faire de spécial pour que Perl se rende compte que `\t` était une tabulation. Si les motifs de Perl *étaient* simplement interpolés en contexte d'apostrophes doubles, vous auriez dû faire quelque chose ; heureusement ils ne le sont pas. Ils sont directement reconnus par l'analyseur de regex.

Le vrai programme *grep* a une option **`-i`** qui rend la recherche insensible à la casse. Vous n'avez pas besoin d'ajouter une telle option à votre programme *pgrep* ; il peut déjà le faire sans modification. Il suffit de lui passer un motif un peu plus élaboré, avec un modificateur `/i` inclus :

```
% pgrep '(?i)anneau' LotR*.pod
```

On cherche maintenant « Anneau », « anneau », « ANNEAU » et ainsi de suite. Vous ne verrez pas beaucoup cette fonctionnalité dans les motifs littéraux, puisque vous pouvez simplement écrire `/anneau/i`. Mais pour les motifs passés sur la ligne de commande, dans les formulaires web ou inclus dans les fichiers de configuration, cela peut s'avérer primordial.

L'opérateur qr// de citation de regex

Les variables qui sont interpolées en motifs le sont nécessairement à l'exécution et non à la compilation. Ceci ralentit l'exécution, car Perl doit vérifier que vous n'avez pas changé le contenu de la variable ; si c'est le cas, il faut alors recompiler l'expression régulière. Comme mentionné dans *Opérateurs de recherche de motifs*, si vous promettez de ne jamais modifier le motif, vous pouvez utiliser l'option `/o` pour interpoler et compiler une seule fois :

```
print if /$pattern/o;
```

Bien que cela fonctionne correctement dans notre programme *pgrep*, ce n'est pas le cas en général. Imaginez que vous ayez une floppée de motifs et que vous vouliez tous les tester dans une boucle, peut-être comme ceci :

```
foreach $elem (@donnees) {
    foreach $motif (@motifs) {
```

11. Si vous ne saviez pas ce qu'est un programme comme *grep*, vous le saurez. Il ne devrait pas y avoir de système sans *grep* — nous pensons que *grep* est le plus utile des petits programmes jamais inventés. (Ce qui logiquement indique que nous ne croyons pas que Perl soit un petit programme.)

```
            if ($elem =~ /$motif/) { ... }
        }
    }
```

Vous ne pourriez pas écrire `/$motif/o`, car la signification de `$motif` change à chaque tour de la boucle interne.

Ce problème est résolu par l'opérateur `qr/`*`MOTIF`*`/imosx`. Cet opérateur cite — et compile — son *`MOTIF`* comme une expression régulière. *`MOTIF`* est interpolé de la même façon que dans `m/`*`MOTIF`*`/`. Si `'` est utilisé comme délimiteur, aucune interpolation de variable (ni des séquences de traduction) n'est faite. L'opérateur renvoie une valeur Perl qui peut être utilisée à la place de la chaîne littérale équivalente dans la recherche ou la substitution correspondante. Par exemple :

```
$regex = qr/ma.CHAINE/is;
s/$regex/quelque chose d'autre/;
```

est équivalent à :

```
s/ma.CHAINE/quelque chose d'autre/is;
```

Pour résoudre notre problème précédent de boucles emboîtées, vous pouvez donc d'abord traiter le motif dans une boucle séparée :

```
@regexes = ();
foreach $motif (@motifs) {
    push @regexes, qr/$motif/;
}
```

Ou bien d'un seul coup en utilisant l'opérateur `map` de Perl :

```
@regexes = map { qr/$_/ } @motifs;
```

Puis modifier la boucle pour utiliser ces regex précompilées :

```
foreach $elem (@donnees) {
    foreach $re (@regexes) {
        if ($elem =~ /$re/) { ... }
    }
}
```

Désormais quand vous faites tourner la recherche, Perl n'est plus obligé de créer une expression régulière compilée à chaque test `if`, car il se rend compte qu'il en possède déjà une.

Le résultat d'un `qr//` peut même être interpolé dans un motif plus grand, comme s'il s'agissait d'une simple chaîne :

```
$regex = qr/$motif/;
$chaine =~ /toto${regex}titi/;   # interpolation dans un motif plus grand
```

Cette fois Perl recompile le motif, mais vous pourriez chaîner plusieurs opérateurs `qr//` en un seul.

La raison pour laquelle cela fonctionne est que l'opérateur `qr//` retourne un objet spécial pour lequel la conversion en chaîne est surchargée comme décrit au chapitre 13, *Surcharge*. Si vous affichez la valeur de retour, vous verrez la chaîne équivalente :

```
$re = qr/ma.CHAINE/is;
print $re;                  # affiche (?si-xm:ma.CHAINE)
```

Les modificateurs `/i` et `/s` ont été activés dans le motif car ils ont été fournis à `qr//`. De même, `/x` et `/m` sont désactivés car ils n'ont pas été fournis.

À chaque fois que vous interpolez des chaînes de provenance inconnue dans un motif, vous devriez être prêt à traiter les exceptions renvoyées par le compilateur de regex, dans le cas où l'on vous aurait passé une chaîne contenant des bestioles indomptables :

```
$re = qr/$motif/is;                     # peut s'échapper et vous mordre
$re = eval { qr/$motif/is } || warn ...  # pris dans une cage extérieure
```

Pour en savoir plus sur l'opérateur `eval`, voir le chapitre 29.

Le compilateur de regex

Après que la passe d'interpolation de variable est passée sur la chaîne, l'analyseur de regex a enfin une occasion d'essayer de comprendre votre expression régulière. À ce niveau-là, il n'y a plus grand chose qui peut mal se passer, sinon embrouiller les parenthèses ou utiliser une séquence de métacaractères qui n'a aucun sens. L'analyseur fait une analyse récursive descendante de votre expression régulière et si elle est correcte, la met sous une forme interprétable par le Moteur (voir la section suivante). La plupart des choses intéressantes qui se passent dans l'analyseur sont liées à l'optimisation de votre expression régulière afin qu'elle soit exécutée le plus rapidement possible. Nous ne chercherons pas à expliquer cette partie-là. C'est un secret de fabrication. (Ne croyez pas les rumeurs qui disent que regarder le code lié aux expressions régulières vous rendra fou ; elles sont très exagérées. Espérons-le.)

Mais vous pourriez avoir envie de savoir ce que l'analyseur a vraiment pensé de votre expression régulière. Si vous lui demandez poliment, il vous le dira. En utilisant `use re "debug";`, vous pouvez examiner le traitement de votre motif par l'analyseur. (Vous pouvez également obtenir la même information en utilisant l'option de ligne de commande **`-Dr`**. Celle-ci est disponible si votre Perl a été compilé avec l'option **`-DDEBUGGING`** à l'intallation.)

```
#!/usr/bin/perl
use re "debug";
"Smeagol" =~ /^Sm(.*)g[aeiou]l$/;
```

La sortie suit. Vous pouvez voir qu'avant l'exécution Perl compile la regex et donne une signification aux composantes du motif : `BOL` pour le début de ligne (^ ou *Beginning Of Line*), `REG_ANY` pour le point, et ainsi de suite :

```
Compiling REx `^Sm(.*)g[aeiou]l$'
size 24 first at 2
rarest char l at 0
rarest char S at 0
   1: BOL(2)
   2: EXACT <Sm>(4)
   4: OPEN1(6)
   6:   STAR(8)
   7:     REG_ANY(0)
   8: CLOSE1(10)
  10: EXACT <g>(12)
  12: ANYOF[aeiou](21)
```

```
  21: EXACT <l>(23)
  23: EOL(24)
  24: END(0)
anchored `Sm' at 0 floating `l'$ at 4..2147483647 (checking anchored)
anchored(BOL) minlen 5
```

Certaines de ces lignes résument les conclusions de l'optimiseur de regex. Il sait que la chaîne doit commencer par « Sm » et qu'il n'y a donc aucune raison de faire l'habituelle recherche de gauche à droite. Il sait que la chaîne doit se terminer par un « l » et peut donc rejeter directement celles qui ne le sont pas. Il sait que la chaîne doit faire au moins cinq caractères de long, et peut donc ignorer toute chaîne plus courte que cela sans autre forme de procès. Il sait également quels sont les caractères les moins courants dans chaque sous-chaîne constante, ce qui peut aider avec les chaînes « étudiées » par study. (Voir study au chapitre 29.)

Ensuite il liste la façon dont le motif est exécuté :

```
Guessing start of match, REx `^Sm(.*)g[aeiou]l$' against `Smeagol'...
Guessed: match at offset 0
Matching REx `^Sm(.*)g[aeiou]l$' against `Smeagol'
  Setting an EVAL scope, savestack=3
   0 <> <Smeagol>         |  1:  BOL
   0 <> <Smeagol>         |  2:  EXACT <Sm>
   2 <Sm> <eagol>         |  4:  OPEN1
   2 <Sm> <eagol>         |  6:  STAR
                           REG_ANY can match 5 times out of 32767...
  Setting an EVAL scope, savestack=3
   7 <Smeagol> <>         |  8:    CLOSE1
   7 <Smeagol> <>         | 10:    EXACT <g>
                             failed...
   6 <Smeago> <l>         |  8:    CLOSE1
   6 <Smeago> <l>         | 10:    EXACT <g>
                             failed...
   5 <Smeag> <ol>         |  8:    CLOSE1
   5 <Smeag> <ol>         | 10:    EXACT <g>
                             failed...
   4 <Smea> <gol>         |  8:    CLOSE1
   4 <Smea> <gol>         | 10:    EXACT <g>
   5 <Smeag> <ol>         | 12:    ANYOF[aeiou]
   6 <Smeago> <l>         | 21:    EXACT <l>
   7 <Smeagol> <>         | 23:    EOL
   7 <Smeagol> <>         | 24:    END
Match successful!
Freeing REx: `^Sm(.*)g[aeiou]l$'
```

Si vous suivez le blanc au milieu de Smeagol, vous pouvez voir le Moteur aller au plus loin pour que le .* soit aussi gourmand que possible, puis revient en arrière jusqu'à trouver une façon pour le reste du motif de réussir. Mais c'est justement le sujet de la section suivante.

« Le petit Moteur qui /(ne)?pouvait(pas)?/ »

Et maintenant nous aimerions vous raconter l'histoire du petit Moteur de Regex qui dit « Je crois que j'peux. Je crois que j'peux. Je crois que j'peux. ».[12]

Dans cette section, nous allons présenter les règles que le Moteur utilise pour effectuer les recherches de correspondance de motifs. Le Moteur est extrêmement persévérant et travailleur. Il est tout à fait capable de continuer à travailler alors que vous croyiez qu'il aurait déjà abandonné. Le Moteur n'abandonne pas tant qu'il n'est pas absolument sûr qu'il n'y a aucun moyen pour lui de trouver une correspondance entre le motif et la chaîne. Les règles ci-dessous expliquent comment le Moteur « croit qu'il peut » aussi longtemps que possible, jusqu'à ce qu'il *sache* qu'il peut ou ne peut pas. Le problème de notre Moteur est qu'il ne s'agit pas simplement de faire monter un train en haut d'une colline. Il s'agit de parcourir un espace (potentiellement) très compliqué de possibilités, en se souvenant de là où il est passé ou pas.

Le Moteur utilise un automate fini non déterministe (NFA, *nondeterministic finite-state automaton*) pour trouver une correspondance. Cela signifie simplement qu'il garde une trace de ce qu'il a et n'a pas essayé, et quand cela ne donne aucun résultat, il revient en arrière et essaye autre chose. C'est le retour arrière, ou backtracking. Le Moteur Perl est capable d'explorer un million de possibilités pour une partie de la recherche, pour les abandonner et aller à la partie de recherche suivante, puis recommencer à explorer pour celle-ci le même million de possibilités. Le Moteur n'est pas très intelligent ; il est juste persévérant et exhaustif. Il est cependant possible d'écrire des motifs efficaces qui n'occasionnent pas trop de retours arrières inutiles.

Quand on vous dit « Les regex choisissent la correspondance la plus à gauche la plus longue », cela signifie que Perl préfère la plus à gauche à la plus longue. Mais le Moteur ne sait pas qu'il privilégie quelque chose à ce niveau. Le choix global résulte de nombreux choix particuliers et indépendants. Voici ces choix :[13]

Règle 1

> Le Moteur essaye de trouver une correspondance aussi loin que possible à gauche de la chaîne, afin que la totalité de l'expression régulière corresponde selon la règle 2.
>
> Le Moteur démarre juste avant le premier caractère et essaye de faire correspondre la totalité de l'expression régulière depuis ce point. L'expression régulière correspond si et seulement si le moteur atteint la fin de l'expression sans s'être arrêté à la fin de la chaîne. S'il a trouvé, il s'arrête immédiatement — il ne recherche pas s'il existe une « meilleure » solution, même si le motif peut correspondre à la recherche de plusieurs autres manières.
>
> S'il n'arrive pas à trouver une correspondance au motif à la première position de la

12. NdT : « The little Engine that could » est l'histoire d'une petite locomotive qui tire un gros chargement de toutes ses forces sans abandonner. Elle est connue de beaucoup d'enfants américains, mais n'a rien de remarquable si ce n'est le « I think I can. I think I can. I think I can... » qui ressemble légérement au bruit d'un moteur à vapeur (en anglais).

13. Certains de ces choix peuvent être ignorés si l'optimiseur a son mot à dire, ce qui revient à faire des coupes dans un arbre de décision. Dans le cadre de cette discussion, nous faisons comme si l'optimiseur n'existait pas.

chaîne, il reconnaît temporairement sa défaite et va à la position suivante de la chaîne, entre le premier et le deuxième caractères, puis réessaie à nouveau toutes les possibilités. En cas de succès, il s'arrête. Sinon, il continue plus avant dans la chaîne. La recherche de correspondance dans son ensemble ne sera considérée comme un échec qu'après avoir essayé de faire correspondre toute l'expression régulière à toutes les positions de la chaîne, y compris après le dernier caractère.

Une chaîne de *n* caractères fournit donc en fait `n + 1` positions où correspondre. C'est parce que les début et fin de correspondance se trouvent *entre* les caractères de la chaîne. Cette règle surprend parfois les gens quand il écrivent un motif tel que `/x*/` qui recherche zéro ou plus « `x` ». Si vous essayez ce motif sur une chaîne comme « `dix` », il ne trouvera pas le « `x` ». Il trouvera en fait la chaîne nulle juste avant le « `d` » et n'ira jamais chercher plus loin. Si vous voulez trouver un ou plusieurs `x`, vous devriez plutôt utiliser le motif `/x+/`. Voir les quantificateurs à la règle 5.

Un corollaire à cette règle est que toute expression régulière qui peut trouver la chaîne nulle est assurée de correspondre à la position la plus à gauche de la chaîne (en l'absence de toute assertion de largeur vide qui vérifie le contraire).

Règle 2

Quand le Moteur rencontre une alternative (aux éléments séparés par des `|`), que ce soit au niveau global ou au niveau du regroupement courant, il les essaie successivement de gauche à droite, en s'arrêtant à la première correspondance qui assure le succès du motif dans son ensemble.

Une alternative correspond à une chaîne si un élément quelconque de l'alternative correspond au sens de la règle 3. Si aucun des éléments de l'alternative ne correspond, il retourne à la règle qui a invoqué cette règle, qui est en générale la règle 1, mais pourrait très bien être la règle 4 ou 6 dans un regroupement. Cette règle cherchera alors une nouvelle position où appliquer la règle 2.

S'il n'existe qu'un seul terme d'alternative, alors celui-ci est vérifié ou non, et la règle 2 est toujours valable. (Il n'existe pas de zéro alternative, car une chaîne vide correspond toujours à quelque chose de longueur nulle.)

Règle 3

Un terme donné d'une alternative est vérifié si chacun de ses éléments est lui aussi vérifié selon les règles 4 et 5 (de façon à ce que toute l'expression régulière soit satisfaite).

Un élément peut consister en une *assertion*, qui est régie par la règle 4, ou en un *atome quantifié*, qui est régi par la règle 5. Les éléments à choix multiples sont hiérarchisés de la gauche vers la droite. Si les éléments ne peuvent être vérifiés, le Moteur rebrousse chemin jusqu'au choix suivant selon la règle 2.

Les éléments qui doivent être vérifiés séquentiellement ne sont pas séparés dans l'expression régulière par quoi que ce soit de syntaxique ; ils sont simplement juxtaposés dans l'ordre dans lequel ils doivent être vérifiés. Lorsque vous cherchez `/^toto/`, vous demandez en fait la détection de cinq éléments les uns à la suite des autres. Le premier est une assertion de longueur nulle et les quatre autres des lettres ordinaires qui se correspondent à elles-mêmes, l'une après l'autre, selon la règle 5.

L'ordre hiérarchique de gauche à droite implique que dans un motif tel que :

```
/x*y*/
```

x* choisit une manière de correspondre, puis y* essaie toutes ses possibilités. Si cela échoue, x* choisit alors sa deuxième possibilité, et fait réessayer toutes ses possibilités à y*. Et ainsi de suite. Les éléments à droite « varient plus vite », pour emprunter une expression au vocabulaire des tableaux multidimensionnels.

Règle 4

Si une assertion ne correspond pas à la position courante, le Moteur revient en arrière à la règle 3 et réessaie les éléments plus haut dans la hiérarchie avec des choix différents.

Certaines assertions sont plus fantaisistes que d'autres. Perl connaît beaucoup d'extensions de regex, dont certaines sont des assertions de largeur nulle. Par exemple, l'assertion positive de prévision `(?=...)` et l'assertion négative de prévision `(?!...)` ne correspondent en réalité à aucun caractère, mais s'assurent plutôt que l'expression régulière représentée par `...` correspondrait (ou non), si nous l'avions essayée.[14]

Règle 5

Un atome quantifié n'est vérifié que si et seulement si l'atome lui-même est vérifié un nombre de fois autorisé par le quantificateur. (L'atome est vérifié selon la règle 6).

Des quantificateurs différents impliquent un nombre différent de vérifications, et la plupart d'entre eux permettent un nombre variable de vérifications. Les correspondances multiples doivent être faites à la suite, c'est-à-dire qu'elles sont adjacentes dans la chaîne. Un atome non quantifié est supposé avoir un quantificateur ne demandant qu'une seule vérification (c'est-à-dire que `/x/` est identique à `/x{1}/`). Si aucune correspondance n'est trouvée à la position courante pour aucune des quantités autorisées pour l'atome en question, le Moteur revient à la règle 3 et réessaye des éléments d'ordre hiérarchique plus élevé avec des valeurs différentes.

Les quantificateurs sont *, +, ?, *?, +?, ?? et les différentes formes d'accolades. Si vous utilisez la forme {*COMPTE*}, alors il n'y a pas le choix et l'atome doit correspondre le nombre exact de fois précisé, ou pas du tout. Sinon, l'atome peut être recherché parmi un ensemble de possibilités de répétition, et le Moteur garde une trace de tous les choix pour revenir en arrière si besoin est. Mais la question est alors de savoir quel choix faire en premier. On pourrait commencer par le nombre maximal et le réduire au fur et à mesure, ou démarrer avec le nombre minimal et l'augmenter petit à petit.

Les quantificateurs traditionnels (sans point d'interrogation à la suite) spécifient une recherche *gourmande* ; c'est-à-dire qu'ils essaient de vérifier autant de caractères que possible. Pour trouver la correspondance maximale, le Moteur doit faire un petit peu attention. Les mauvais choix coûtent potentiellement cher, c'est pourquoi le Moteur ne compte pas vraiment à partir de la valeur maximale, qui pourrait après tout être Très Grande et provoquer des millions de mauvais choix. Le Moteur fait en réalité quelque chose d'un petit peu plus malin : il commence par compter

14. En réalité, elle est *testée* par le Moteur. Celui-ci retourne à la règle 2 pour tester le sous-motif, puis efface toute trace de ce qui avait été consommé de la chaîne, pour ne retourner que le succès ou l'échec du sous-motif comme valeur de l'assertion. (Il se souvient cependant de toute chaîne capturée.)

de façon *croissante* combien d'atomes correspondants (à la suite) sont effectivement présents dans la chaîne, puis utilise *ce* maximum réel comme premier essai. (Il se souvient aussi des choix les plus courts, au cas où le plus long ne marcherait pas.) Il essaie (enfin) de vérifier le reste du motif, en supposant que le choix le plus long est le bon. Si le choix le plus long ne produit de correspondance pour le reste du motif, il revient en arrière et essaie le plus long suivant.

Si vous écrivez `/.*toto/` par exemple, il essayera de trouver le nombre maximum de caractères « quelconques » (représentés par le point) jusqu'à la fin de la ligne avant même de commencer à chercher « `toto` » ; puis quand le « `toto` » ne correspond pas à cet endroit (et il ne peut pas, puisqu'il n'y a pas assez de place en fin de chaîne pour le trouver), le Moteur va reculer d'un caractère à la fois jusqu'à trouver un « `toto` ». S'il y a plus d'un « `toto` » sur la ligne, il s'arrêtera au dernier, puisque ce sera en fait le *premier* qu'il rencontre au cours de sa remontée. Quand le motif complet réussit en utilisant une longueur particulière de `.*`, le Moteur sait qu'il peut abandonner les autres choix plus courts pour `.*` (ceux qu'il aurait utilisés si le « `toto` » en cours n'avait finalement pas fonctionné).

En mettant un point d'interrogation après n'importe quel quantificateur gourmand, vous le transformez en quantificateur frugal qui choisit la plus petite quantité pour son premier essai. Donc, si vous écrivez `/.*?toto/`, le `.*?` essayera d'abord de correspondre à 0 caractères, puis 1, puis 2, et ainsi de suite jusqu'à trouver le « `toto` ». Au lieu de rebrousser chemin en arrière, il rebrousse chemin en avant, pour ainsi dire. Et finit par trouver le premier « `toto` » de la ligne au lieu du dernier.

Règle 6

Chaque atome correspond selon son type. Si l'atome n'est pas vérifié (ou l'est, mais que sa vérification ne permet pas de correspondance pour le reste du motif), le Moteur revient en arrière à la règle 5 et essaie le choix suivant possible en quantité pour cet atome.

Les atomes correspondent selon les types suivants :

- Une expression régulière entre parenthèses, `(...)`, trouve tout ce que l'expression régulière représentée par `...` trouve selon la règle 2. Les parenthèses servent donc d'opérateur de regroupement pour la quantification. Les parenthèses simples ont également pour effet de capturer la sous-chaîne trouvée pour une utilisation ultérieure comme *référence arrière* (ou *backreference* en anglais). Cet effet de bord peut être supprimé en utilisant plutôt `(?:...)`, qui n'a que les propriétés de regroupement ; il ne stocke rien dans `$1`, `$2` et autres. Il existe d'autres formes d'atomes (et d'assertions) entre parenthèses — voir la suite de ce chapitre.
- Un point correspond à tout caractère, sauf éventuellement le saut de ligne.
- Une liste de caractères entre crochets (une *classe de caractères*) correspond à n'importe lequel des caractères spécifiés dans la liste.
- Une lettre précédée d'un antislash correspond soit à un caractère particulier, soit à un caractère d'un ensemble particulier, comme indiqué dans le *tableau 5-7*.
- Tout autre caractère précédé d'un antislash correspond à ce caractère lui-même.
- Tout caractère non mentionné ci-dessus se correspond à lui-même.

Tout cela peut sembler plutôt compliqué, mais l'avantage est que pour chaque liste de choix donnée par un quantificateur ou une alternative, le Moteur a un bouton à tourner. Il tournera tous ces boutons jusqu'à ce que tout le motif corresponde. Les règles ne font qu'indiquer dans quel ordre le Moteur peut tourner ces boutons. Dire que le Moteur préfère la correspondance la plus longue à gauche signifie simplement que le bouton qu'il tourne le plus lentement est celui correspondant à la position de départ. Revenir en arrière consiste simplement à tourner dans l'autre sens le bouton tourné à l'instant de façon à essayer un autre bouton plus haut dans l'ordre hiérarchique, c'est-à-dire un qui varie plus lentement.

Voici un exemple plus concret sous la forme d'un programme qui détecte quand deux mots consécutif partagent un suffixe et un préfixe communs :

```
$a = 'conflit';
$b = 'litige';
if ("$a $b" =~ /^(\w+)(\w+) \2(\w+)$/) {
    print "$2 superposé dans $1-$2-$3\n";
}
```

Ceci affiche :

```
lit superposé dans conf-lit-ige
```

Vous pourriez croire que `$1` capture d'abord l'intégralité de « `chienlit` » par gourmandise. C'est exactement ce qu'il fait — d'abord. Mais une fois rendu là, il n'y a plus de caractères à mettre dans `$2`, qui a besoin de recevoir des caractères à cause du quantificateur `+`. Le Moteur bat donc en retraite et `$1` donne à contrecœur un caractère à `$2`. Cette fois l'espace est trouvé correctement, mais quand il voit le `\2`, qui représente un simple « `t` ». Le caractère suivant dans la chaîne n'est pas un « `t` », mais un « `l` ». Ceci fait donc revenir le Moteur en arrière et réessayer plusieurs fois, pour finalement forcer `$1` à laisser le lit à `$2`.

En fait, cela ne marchera pas forcément bien si la superposition elle-même est un doublon, comme pour les mots « `rococo` » et « `cocon` ». L'algorithme ci-dessus aurait simplement décidé que la chaîne en superposition, `$2`, devait être « `co` » plutôt que « `coco` ». Mais nous ne voulons pas de « `rococococon` » ; nous voulons plutôt un « `rococon` ». Nous voici dans un cas où il est possible d'être plus malin que le Moteur. L'ajout d'un quantificateur minimal à la partie correspondant à `$1` donne le motif bien meilleur `/^(\w+?)(\w+) \2(\w+)$/`, qui fait exactement ce que nous voulons.

Pour une discussion plus détaillée des avantages et inconvénients des différentes sortes de moteurs d'expressions rationnelles, reportez-vous au livre de Jeffrey Friedl, *Maîtrise des expressions régulières* (ou en version originale *Mastering Regular Expressions*). Le Moteur d'expressions régulières de Perl fonctionne très bien pour tous les problèmes quotidiens que vous voulez résoudre avec Perl, mais tout aussi correctement pour ce genre de problème pas si quotidien, pourvu que vous soyez un peu compréhensif.

Motifs étendus

Assertions périphériques

Parfois vous voulez juste jeter un œil. Voici quatre extensions de regex qui vous aident justement à faire cela. Nous les appelons *périphériques* (en anglais *lookaround*) car elles vous permettent de jeter un œil alentour de façon hypothétique, sans correspondre effectivement à des caractères. Ces assertions testent le fait qu'un motif correspondrait (ou non) dans le cas où nous l'essayerions. Le Moteur fait cela en essayant effectivement de faire correspondre le motif puis en prétendant par la suite qu'il n'y a pas eu de correspondance de faite (même si c'est le cas).

Quand le Moteur regarde en avant par rapport à sa position actuelle dans la chaîne, nous parlons d'assertion de *prévision*[15] (*lookahead*). S'il regarde en arrière, nous parlons d'assertion de *rétrovision*. Les motifs de prévision peuvent être n'importe quelle expression rationnelle, tandis que les motifs de rétrovision ne peuvent qu'être de longueur fixe, car ils doivent savoir d'où partir pour leur correspondance hypothétique.

Alors que ces quatre extensions sont toutes de largeur nulle, et ne consomment donc pas de caractères (du moins pas officiellement), vous pouvez en fait capturer des sous-chaînes à l'intérieur si vous fournissez un niveau supplémentaire de parenthèses de capture.

`(?=`*`MOTIF`*`)` *(positive de prévision)*

Quand le Moteur rencontre `(?=`*`MOTIF`*`)`, il regarde en avant dans la chaîne pour s'assurer que *`MOTIF`* se produit. Pour mémoire, dans notre précédent suppresseur de doublons, nous avons dû écrire une boucle car notre motif consommait trop de texte à chaque tour :

```
$_ = "Paris AU AU AU AU printemps."

# supprime les doublons (et les triplons (et les quadruplons...))
1 while s/\b(\w+) \1\b/$1/gi;
```

À chaque fois que vous entendez « consomme trop », vous devriez immédiatement penser « assertion de pré-vision ». (Enfin, presque toujours.) En regardant à l'avance au lieu de directement avaler le deuxième mot, vous pouvez écrire un suppresseur de doublons qui fonctionne en une passe, comme ceci :

```
s/ \b(\w+) \s (?= \1\b ) //gxi;
```

Évidemment, il y a encore un problème, puisque cela va malmener des phrases parfaitement valides comme « J'ai une BELLE BELLE-mère ».

`(?!`*`MOTIF`*`)` *(négative de prévision)*

Quand le Moteur rencontre `(?!`*`MOTIF`*`)`, il regarde en avant dans la chaîne pour s'assurer que *`MOTIF`* n'apparaît *pas*. Pour corriger notre exemple précédent, nous pouvons ajouter une assertion négative de prévision après l'assertion positive pour

15. NdT : Ce terme et le suivant ne sont pas forcément très heureux, c'est pourquoi nous vous fournissons le vocabulaire original pour que vous soyez pas étonnés en lisant des documentations utilisant ces termes.

éliminer le cas des mots composés :

```
s/ \b(\w+) \s (?= \1\b (?! -\w))//xgi;
```

Ce `\w` final est nécessaire pour éviter de confondre un mot composé avec mot suivi d'un tiret. Nous pouvons même aller plus loin, puisque plus tôt dans ce chapitre nous nous sommes intentionnellement servis de l'expression « nous nous sommes », et que nous aimerions que notre programme ne « corrige » pas cela pour nous. Nous pouvons donc ajouter une alternative à l'assertion négative de pré-vision de manière à ne pas corriger cela (et montrer ainsi que n'importe quel type de parenthèses peut servir à regrouper des alternatives) :

```
s/ \b(\w+) \s (?= \1\b (?! -\w | \s sommes))//gix;
```

Nous nous sommes maintenant assurés que cette formulation spécifique n'aura pas de problème. Hélas le poème *Mes petites amoureuses* de Rimbaud est toujours cassé[16]. Aussi ajoutons-nous une nouvelle exception :

```
s/ \b(\w+) \s (?= \1\b (?! -\w | \s sommmes | \s aimions))//igx;
```

Voilà qui commence à s'emballer. Faisons plutôt une Liste Officielle d'Exceptions, en utilisant un mignon petit truc d'interpolation avec la variable `$"` pour séparer les éléments de l'alternative avec le caractère `|` :

```
@nousnous = qw(sommes aimions);
local $" = '|';
s/ \b(\w+) \s (?= \1\b (?! -\w | \s (?: @nousnous )))//xig;
```

`(?<=`*`PATTERN`*`)` *(positive de rétrovision)*

When the Engine encounters `(?<=`*`PATTERN`*`)`, it looks backward in the string to ensure that *`PATTERN`* already occurred.

Notre exemple a toujours un problème. Bien que l'on permette à Rimbaud de dire « Nous nous aimions », cela autorise également « La totalité des des sommes seront converties en Euros ». Nous pouvons ajouter une assertion positive de rétrovision en tête de notre liste d'exceptions pour nous assurer que nous n'appliquons nos exceptions `@nousnous` qu'à un vrai « nous nous ».

```
s/ \b(\w+) \s (?= \1\b (?! -\w | (?<= nous) \s (?: @nousnous )))//ixg;
```

Oui, tout cela devient bien compliqué, mais rappelons que cette section s'appelle *Motifs étendus*. Si vous avez besoin de compliquer encore le motif après ce que nous lui avons fait subir, l'utilisation judicieuse des commentaires et de `qr//` devrait vous aider à ne pas devenir fou.

`(?<!`*`MOTIF`*`)` *(négative de rétrovision)*

Quand le Moteur rencontre `(?<!`*`MOTIF`*`)`, il regarde en arrière dans la chaîne pour s'assurer que *`MOTIF`* n'a pas été trouvé.

Essayons avec un exemple simple cette fois-ci. Pourquoi pas une règle d'orthographe facile ? Si vous ne savez plus si un mot se termine en « euil » ou en « ueil », et

16. NdT : Très exactement à la troisième strophe :

Nous nous aimions à cette époque,
Bleu laideron !
On mangeait des œufs à la coque
Et du mouron !

que vous avez plutôt la manie d'écrire « ueil » (parce que vous êtes né à Rueil, par exemple). L'expression régulière suivante peut remettre les choses en ordre :

```
s/(?<!c|g)ueil/euil/g
```

Ne pas savoir écrire « fauteuil » ou « cerfeuil » n'est pas un écueil et ne doit pas blesser votre orgueil.

Sous-motifs sans retour arrière

Comme nous l'avons décrit dans « *Le petit Moteur qui /(ne)?pouvait(pas)?/* », le Moteur fait souvent machine arrière alors qu'il progresse le long d'un motif. Vous pouvez empêcher le Moteur de faire demi-tour au milieu d'une série de possibilités en créant un *sous-motif sans retour arrière*. Un sous-motif sans retour arrière se présente comme `(?>`*`MOTIF`*`)`, et fonctionne exactement comme un simple `(?:`*`MOTIF`*`)`, sauf qu'une fois que *`MOTIF`* a trouvé une correspondance, il supprime toute possibilité de retour arrière sur tous les quantificateurs ou alternatives à l'intérieur du sous-motif. (C'est pourquoi il est inutile d'utiliser cela sur un *`MOTIF`* qui ne contient pas de quantificateurs ou d'alternatives.) La seule manière de le faire changer d'avis est de rebrousser chemin jusqu'à quelque chose avant ce sous-motif et de réentrer dans le sous-motif par la gauche.

C'est comme rendre visite à un concessionaire automobile. Après un certain temps de discussion sur le prix, vous finissez par donner un ultimatum : « Voici ma meilleure offre : c'est à prendre ou à laisser. ». S'ils ne la prennent pas, vous ne recommencez pas à marchander. Vous opérez un retour arrière vers la porte. Vous pouvez ensuite aller voir un autre concessionaire et recommencer à marchander. Vous pouvez recommencer à marchander, mais seulement parce que vous êtes réentré dans le sous-motif sans retour arrière dans un contexte différent.

Pour les adorateurs de Prolog ou SNOBOL, vous pouvez voir cela comme un opérateur cut ou fence à portée limitée.

Voyons comment dans `"aaab" =~ /(?:a*)ab/`, le `a*` commence par trouver trois `a`, puis en laisse un car le dernier `a` est utilisé plus loin. Le sous-motif sacrifie une partie de son butin pour permettre à toute la correspondance de réussir. (Ce qui revient à laisser le marchand de voitures obtenir un peu plus de votre argent par peur de ne pas pouvoir conclure l'affaire.) En revanche, le sous-motif dans `"aaab" = ~ /(?>a*)ab/` n'abandonnera rien de ce qu'il a pu trouver, même si cette attitude fait échouer la correspondance toute entière.

Bien que `(?>`*`MOTIF`*`)` soit utile pour modifier le comportement d'un motif, il est souvent utilisé pour accélérer l'échec de certaines recherches dont vous savez qu'elles vont échouer (à moins qu'elles ne réussissent tout à fait). Le Moteur peut prendre un temps extraordinairement long pour échouer, en particulier avec des quantificateurs emboîtés. Le motif suivant réussira de façon quasi-instantanée :

```
$_ = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab";
/a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*[b]/;
```

Mais ce n'est pas le succès qui est un problème. C'est l'échec qui en est un. Si vous retirez ce « `b` » final de la chaîne, le motif va probablement tourner pendant des années et des années avant d'échouer. Pendant des millénaires et des millénaires. En réalité des mil-

liards et des milliards d'années.[17] Vous pouvez constater en examinant le motif qu'il ne peut réussir s'il n'y a pas de « b » à la fin de la chaîne ; mais l'optimiseur n'est pas assez malin (au moment où nous écrivons ces lignes) pour se rendre compte que `/[b]/` est équivalent à `/b/`. Mais si vous lui donnez un indice, vous pouvez le faire échouer rapidement tout en le laissant réussir là où il peut :

```
/(?>a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*)[b]/;
```

Voici un exemple plus réaliste (espérons). Imaginez un programme qui est supposé lire paragraphe par paragraphe et n'afficher que les lignes qui se continuent, les lignes de continuation étant spécifiées par un antislash final. Voici un exemple tiré du *Makefile* de Perl qui utilise cette convention :

```
# Files to be built with variable substitution before miniperl
# is available.
sh = Makefile.SH cflags.SH config_h.SH makeaperl.SH makedepend.SH \
        makedir.SH myconfig.SH writemain.SH
```

Vous pourriez écrire votre programme de cette façon :

```
#!/usr/bin/perl -00p
while ( /( (.+) ( (?<=\\) \n .* )+ ) /gx) {
    print "J'ai $.: $1\n\n";
}
```

Ça marche, mais c'est vraiment très lent. C'est parce que le Moteur fait machine arrière un caractère à la fois depuis la fin de la ligne, en diminuant ce qui se trouve dans `$1`. C'est sans intérêt. L'écrire sans les captures surnuméraires n'améliore pas sensiblement les choses. Utiliser :

```
(.+(?:(?<=\\)\n.*)+)
```

pour un motif est un peu plus rapide, mais pas tellement. C'est dans ce cas qu'un motif sans retour arrière est très utile. Le motif :

```
((?>.+)(?:(?<=\\)\n.*)+)
```

fait la même chose, mais est plus rapide d'un ordre de grandeur. C'est parce qu'il ne perd pas son temps à chercher quelque chose qui n'est pas là.

Il n'est pas possible d'obtenir avec `(?>...)` un succès que vous n'auriez obtenu avec `(?:...)` ou même un simple `(...)`. Mais si votre recherche doit échouer, autant échouer rapidement et passer à la suite.

Motifs programmatiques

La plupart des programmes Perl sont écrits dans un style impératif (aussi appelé procédural). C'est un peu comme une suite d'ordres listés dans un facile à suivre : « Préchauffer le four, mélanger, glacer, cuire, laisser refroidir, servir aux martiens. ». Parfois vous rajoutez à ce mélange une bonne cuillerée de programmation fonctionnelle (« Utilisez un peu plus de glacage que vous ne pensez en avoir besoin, même après

17. En fait, c'est plutôt de l'ordre des septillions d'années. Nous ne savons pas exactement combien de temps cela prendrait. Nous n'avons pas eu la patience d'attendre de le voir échouer. En tous les cas, votre ordinateur risque fort de planter avant que l'univers ne disparaisse, et cette expression régulière met plus de temps que ces deux événements à se produire.

avoir tenu compte de ce conseil, récursivement ») ou vous le saupoudrez de quelques techniques orientées-objet (« Merci de laisser les objets anchois de côté »). Souvent c'est un mélange de tout cela à la fois.

Mais le Moteur d'expressions régulières a une approche complètement différente de la résolution de problèmes, une approche plus déclarative. Vous décrivez vos objectifs dans le langage des expressions régulières, et le Moteur implémente la logique nécessaire pour atteindre vos objectifs. Les langages de programmation logique (comme Prolog) ont une moins grande visibilité que les trois autres styles, mais ils sont plus répandus que vous ne pourriez le croire. Perl ne pourrait même pas être construit sans *make* ou *yacc* ; tous deux pouvant être considérés, sinon comme des langages purement déclaratifs, au moins comme des hybrides qui mélangent la programmation impérative et la programmation logique.

Vous pouvez aussi faire ce genre de choses en Perl, en mélangeant ensemble des déclarations d'objectifs avec du code impératif, de façon encore plus fluide que nous ne l'avons fait jusqu'à présent afin de profiter des forces de chacun. Vous pouvez construire programmatiquement la chaîne que vous allez finir par proposer au Moteur de regex, c'est-à-dire d'une certaine manière créer un programme qui écrit un nouveau programme à la volée.

Vous pouvez également fournir des expressions Perl normales comme remplacement dans `s///` à l'aide du modificateur `/e`. Ceci vous permet de générer dynamiquement la chaîne de remplacement en exécutant un morceau de code à chaque fois que le motif correspond.

D'une manière encore plus élaborée, vous pouvez inclure des bouts de code partout où vous le voulez au beau milieu d'un motif en utilisant l'extension `(?{`*`CODE`*`})`. Celui-ci sera exécuté à chaque fois que le Moteur rencontrera ce code, au fur et à mesure de ses pérégrinations, dans la danse compliquée des retours en arrière.

Enfin, vous pouvez utiliser `s///ee` ou `(??{`*`CODE`*`})` pour ajouter un nouveau niveau d'indirection : le *résultat* de l'exécution de ces bouts de code sera lui-même réévalué pour être réutilisé, créant des morceaux de programme et de motif au vol, juste à temps.

Motifs générés

Il a été dit[18] que les programmes qui écrivent des programmes sont les programmes les plus heureux au monde. Dans le livre de Jeffrey Friedl, *Maîtrise des expressions régulières* (*Mastering Regular Expressions*), le tour de force final montre comment écrire un programme qui produit une expression régulière capable de déterminer si une chaîne est conforme au standard défini par le RFC 822 ; c'est-à-dire s'il contient un en-tête de mail conforme au standard. Le motif produit fait plusieurs milliers de caractères de long, et est aussi facile à lire qu'un dump mémoire binaire après un crash. Mais le détecteur de motif de Perl s'en moque ; il compile juste le motif sans difficulté et, ce qui est encore plus intéressant, l'exécute très rapidement — bien plus rapidement, en fait, que beaucoup de motifs beaucoup plus courts qui ont des besoins de retour arrière plus compliqués.

C'est un exemple très compliqué. Nous vous avons précédemment montré un exemple très simple de la même technique quand nous avons fabriqué un motif `$nombre` à partir

18. Par Andrew Hume, le célèbre philosophe Unix.

de ses éléments constitutifs (voir la section *Interpolation de variables*). Mais pour vous montrer la puissance de l'approche programmatique pour la production d'un motif, nous allons nous pencher sur un problème de complexité moyenne.

Supposons que vous vouliez trouver tous les exemples de mots avec une certaine suite de voyelles et de consonnes ; par exemple « audio » et « aeree » suivent tous deux un motif VVCVV. Même si décrire ce qui compte comme voyelle ou consonne est simple[19], vous n'aurez pas vraiment envie de l'écrire plus d'une fois. Même pour un cas simple comme notre VVCVV, vous devrez écrire un motif comme :

```
^[aeiouy][aeiouy][bcdfghjklmnpqrstvwxzy][aeiouy][aeiouy]$
```

Un programme plus général accepterait une chaîne comme « VVCVV » pour générer (programmatiquement) le motif pour vous. Pour être même encore plus flexible, il pourrait même accepter un mot comme « audio », l'utiliser comme modèle pour en déduire « VVCVV » et de là construire le long motif ci-dessus. Cela a l'air compliqué, mais ne l'est vraiment pas, car nous allons laisser le programme générer le motif pour nous. Voici un simple programme *cvmap* qui fait tout cela :

```
#!/usr/bin/perl
$voyelles  = 'aeiouy';
$consonnes = 'bcdfghjklmnpqrstvwxzy';
%assoc = (C => $consonnes,
          V => $voyelles);            # initialise la liste pour V et C

for $classe ($voyelles, $consonnes) { # pour chaque type
    for (split //, $class) {    # récupère chaque lettre de chaque type
        $assoc{$_} .= $class;   # et réassocie chaque lettre à chaque type
    }
}

for $char (split //, shift) {         # pour chaque lettre du modèle
    $motif .= "[$assoc{$char}]";      # ajoute la classe de caractères
appropriée
}

$re = qr/^${motif}$/i;                # compile le motif
print "REGEX : $re\n";                # affiche une vérification
@ARGV = ('/usr/dict/words')           # prend un dictionnaire par défaut
    if -t && !@ARGV;

while (<>) {                          # et maintenant parcourt l'entrée
    print if /$re/;                   # en affichant les lignes qui
                                      # correspondent
}
```

19. NdT : Plus simple en anglais qu'en français, car nos classes de caractères seraient plutot [aàâeéèêëêiïîoôuùû¸y] pour les voyelles et [bcçdefghjklmnpqrstvwxyz] pour les consonnes. Et nous ne vous parlons même pas des ligatures ! Dans les exemples qui suivent, le dictionnaire utilisé est une version sans accents. L'adaptation pour traiter les accents est laissée en exercice au lecteur. Notez cependant que `use locale;` vous facilitera les `\w`.

La variable `%assoc` contient tous les morceaux intéressants. Elle utilise chaque lettre de l'alphabet comme clé, et la valeur correspondante est l'ensemble des lettres du même type. Nous ajoutons également V et C, pour que vous puissiez utiliser « `VVCVV` » ou « `audio` » et tout de même obtenir « `aeree` ». Chaque caractère de l'argument fourni est utilisé pour trouver la classe de caractères associée et l'ajouter au motif. Une fois le motif créé et compilé par `qr//`, la correspondance (même si le motif est très long) va tourner rapidement. Voici un *extrait* de ce que vous pourriez obtenir en faisant tourner ce programme « fortuitement » :

```
% cvmap fortuitement /usr/dict/words_fr
REGEX : (?i-xsm:^[cbdfghjklmnpqrstvwxzy][aeiouy][cbdfghjklmnpqrstvwxzy][cbd
fghjklmnpqrstvwxzy][aeiouy][aeiouy][cbdfghjklmnpqrstvwxzy][aeiouy][cbdfghjk
lmnpqrstvwxzy][aeiouy][cbdfghjklmnpqrstvwxzy][cbdfghjklmnpqrstvwxzy]$)
banqueterent
becqueterent
bestialement
biscuiterent
cabriolerent
cabriolerons
cabrioleront
certainement
circuiterent
...
```

En vous laissant admirer cette `REGEX`, nous vous *persuaderons verbeusement* mais *certainement* de l'économie de frappe ainsi *sommairement* réalisée.

Évaluations de substitution

Quand le modificateur `/e` (« e » pour évaluation d'expression) est utilisé sur une expression `s/`*`MOTIF`*`/`*`CODE`*`/e`, la partie de remplacement est interprétée comme une expression, et pas seulement comme une chaîne entre apostrophes doubles. C'est comme un `do {`*`CODE`*`}`. Même si cela a l'air d'une chaîne, c'est simplement un bloc de code qui sera compilé en même temps que le reste de votre programme, bien avant que la substitution ne soit réellement faite.

Vous pouvez utiliser le modificateur `/e` pour construire des chaînes de remplacement avec une logique plus fantaisiste que celle que permet l'interpolation entre doubles apostrophes. Voici la différence :

```
s/(\d+)/$1 * 2/;     # Remplace "42" par "42 * 2"
s/(\d+)/$1 * 2/e;    # Remplace "42" par "84"
```

Et ceci convertit les degrés Celsius en Fahrenheit :

```
$_ = "Préchauffez le four à 233C.\n";
s/\b(\d+\.?\d*)C\b/int($1 * 1.8 + 32) . "F"/e;   # convertit en 451F
```

Les applications sont sans limite. Voici un filtre qui modifie un fichier en place (comme un éditeur) en ajoutant 100 à tout nombre qui se trouve en début de ligne (et qui n'est pas suivi par des deux-points, que nous ne faisons que regarder, sans les trouver ni les remplacer) :

```
% perl -pi -e 's/^(\d+)(?=:)/100 + $1/e' fichier
```

De temps à autre, vous voudrez faire un peu plus que simplement utiliser dans un autre calcul la chaîne que vous avez trouvée. Vous voulez parfois que cette chaîne *soit* un calcul, dont vous utiliserez l'évaluation en tant que valeur de remplacement. Chaque modificateur `/e` supplémentaire rajoute un `eval` autour du code à exécuter. Les deux lignes suivantes font la même chose, mais la première est plus facile à lire :

```
s/MOTIF/CODE/ee
s/MOTIF/eval(CODE)/e
```

Vous pourriez utiliser cette technique pour remplacer les apparitions de valeurs scalaires simples par leurs valeurs :

```
s/(\$\w+)/$1/eeg;       # Interpole les valeurs de la plupart des scalaires
```

Comme c'est vraiment un `eval`, le `/ee` trouve même les variables lexicales. Un exemple un peu plus élaboré calcule la valeur de remplacement pour des expressions arithmétiques simples sur des entiers (positifs) :

```
$_ = "J'ai 4 + 19 euros et 8/2 centimes.\n";
s{ (
        \d+ \s*         # trouve un entier
        [+*/-]          # et un opérateur arithmétique
        \s* \d+         # et un autre entier
   )
}{ $1 }eegx;            # puis trouve $1 et exécute ce code
print;                  # "J'ai 23 euros et 4 centimes."
```

Comme tout autre `eval` *`CHAINE`*, les erreurs de compilation (comme des problèmes de syntaxe) et les exceptions à l'exécution (comme une division par zéro) sont capturées. Dans ce cas, la variable `$@` (`$EVAL_ERROR`) dit ce qui s'est mal passé.

Évaluation de code lors de la recherche

Dans la plupart des programmes qui utilisent les expressions régulières, ce sont les structures de contrôle du programme tout autour qui dirigent le flux logique de l'exécution. Vous écrivez des boucles `if` ou `while`, faites des appels de fonctions ou de méthodes, qui se trouvent appeler une opération de recherche de motif de temps à autre. Même avec `s///e`, c'est l'opérateur de substitution qui a le contrôle et n'exécute le code de remplacement qu'après une recherche réussie.

Avec les *sous-motifs de code*, la relation usuelle entre les expressions régulières et le code du programme est inversée. Alors que le Moteur applique ses règles à votre motif au moment de la recherche, il peut tomber sur une extension de regex de la forme `(?{`*`CODE`*`})`. Quand il est déclenché, ce sous-motif ne fait aucune reconnaissance ni aucune assertion périphérique. C'est une assertion de largeur nulle qui « réussit » toujours et n'est évaluée que pour ses effets de bord. À chaque fois que le moteur doit passer sur le sous-motif de code pour progresser dans le motif, il exécute ce code.

```
"glyphe" =~ /.+ (?{ print "yo" }) ./x;  # Affiche "yo" deux fois.
```

Alors que le Moteur essaie d'établir la correspondance entre « glyphe » et ce motif, il laisse d'abord `.+` avaler les cinq lettres. Puis il affiche « `yo` ». Quand il tombe sur le point final, les cinq lettres ont déjà été avalées ; il lui faut donc faire machine arrière sur le `.+` et lui faire rendre une des lettres. Il avance de nouveau dans le motif, faisant un nouvel affichage de « `yo` » au passage, fait correspondre `e` au point final et termine sa recherche avec succès.

Les accolades autour du fragment de *CODE* sont destinées à vous rappeler qu'il s'agit d'un bloc de code Perl, et qu'il se comporte comme un bloc au sens lexical. C'est-à-dire que si vous utilisez `my` pour déclarer une variable à portée lexicale, elle sera privée à ce bloc. Mais si vous utilisez `local` pour localiser une variable à portée dynamique, il risque de ne pas faire ce à quoi vous vous attendez. Un sous-motif (`?{`*CODE*`}`) crée une portée dynamique implicite qui reste valide pour le reste du motif, jusqu'à ce qu'il réussisse ou qu'il fasse machine arrière à travers ce sous-motif de code. Une manière de voir cela est que le bloc ne retourne pas vraiment quand il arrive à la fin. Au lieu de cela, il faut un appel récursif invisible au Moteur pour essayer de détecter le reste du motif. C'est seulement quand cet appel récursif se termine qu'il revient du bloc, en « délocalisant » les variables localisées.[20]

Dans l'exemple suivant, nous initialisons `$i` à `0` en incluant un sous-motif de code au début du motif. Ensuite nous capturons autant de caractères que possible avec `.*` — mais nous mettons un autre sous-motif de code entre le `.` et l'`*` comme cela nous pourrons savoir combien de fois le `.` trouve quelque chose.

```
$_ = 'lothlorien';
m/  (?{ $i = 0 })              # Met $i à 0
    (.    (?{ $i++ })    )*    # Met à jour $i, même après un retour
                               # en arrière
    lori                       # Force un retour arrière
 /x;
```

Le Moteur se met donc joyeusement en route, mettant `$i` à `0` et laissant `.*` gober les 10 caractères de la chaîne. Quand il recontre le `lori` dans le motif, il revient en arrière et abandonne quatre caractères du `.*`. À la fin de la recherche, `$i` vaudra toujours `10`.

Si vous vouliez que `$i` représente le nombre de caractères que le `.*` a effectivement conservés à la fin, vous pouvez utiliser la portée dynamique à l'intérieur du motif :

```
$_ = 'lothlorien';
m/  (?{ $i = 0 })
    (. (?{ local $i = $i + 1; }) )*  # Met à jour $i, avec protection contre
le retour arrière
    lori
    (?{ $resultat = $i })           # Copie dans une variable non localisée
 /x;
```

Ici nous utilisons `local` pour nous assurer que `$i` contient le nombre de caractères trouvés par `.*`, en tenant compte des retours en arrière. `$i` sera perdue après la fin de l'expression régulière, aussi le sous-motif de code `(?{ $result = $i })` conserve la valeur de `$i` dans `$resultat`.

---

20. Les personnes familières des analyseurs récursif descendants pourront trouver ce comportement perturbant parce que de tels compilateurs retournent d'un appel de fonction récursif dès qu'ils arrivent à trouver quelque chose. Le Moteur ne fait pas cela — quand il trouve quelque chose, il descend *plus profond* en recursion (même quand il sort d'un groupe entre parenthèses !). Un analyseur récursif descendant est à un minimum de récursion quand il réussit à la fin, mais le Moteur est à un *maximum* de récursion quand il réussit à la fin du motif. Vous pourriez trouver utile de faire pendre le motif par son extrémité gauche et de le voir comme une représentation de l'arbre des appels. Si vous arrivez à visualiser cette image, la portée dynamique des variables locales vous sera plus compréhensible. (Et si vous ne pouvez pas, ce n'est pas pire qu'avant.)

La variable spéciale `$^R` (décrite au chapitre 28) contient le résultat du dernier `(?{CODE})` qui a été exécuté au cours d'une recherche réussie.

Vous pouvez utiliser une extension `(?{CODE})` comme la *COND* d'une `(?(COND)SIVRAI|SIFAUX)`. Si vous faites cela, `$^R` ne sera pas mis à jour et vous pouvez omettre les parenthèses autour de la condition :

```
"glyphe" =~ /.+(?(?{ $toto{titi} gt "symbole" }).|signet)./;
```

Ici, nous testons si `$toto{titi}` est plus grand que `symbole`. Si oui, nous incluons `.` dans le motif, sinon nous incluons `signet` dans le motif. On peut l'étirer pour le rendre un peu plus lisible :

```
"glyphe" =~ m{
    .+                              # quelques n'importe quoi
    (?(?{                           # si
            $toto{titi} gt "symbole" # ceci est vrai
        })
            .                       # trouve un autre n'importe quoi
        |                           # sinon
            signet                  # trouve signet
    )
    .                               # et un n'importe quoi de plus
}x;
```

Quand `use re 'eval'` est enclenché, une regex a le droit de contenir un sous-motif `(?{CODE})` même si l'expression régulière interpole des variables :

```
/(.*?) (?{length($1) < 3 && warn}) $suffixe/;  # Erreur sans use re 'eval'
```

Ceci est normalement interdit pour des raisons de sécurité. Même si le motif ci-dessus est sans danger car `$suffixe` est sans danger, l'analyseur de regex ne peut pas savoir quelles parties de la regex ont été interpolées et lesquelles ne l'ont pas été. C'est pourquoi il interdit simplement les sous-motifs de code s'il y a eu de l'interpolation.

Si le motif s'obtient à partir de données entachées, même `use re 'eval'` n'autorisera pas la recherche de motif à continuer.

Quand `use re 'taint'` est enclenché et qu'une chaîne entachée est la cible d'une regex, les sous-motifs capturés (dans les variables numérotées ou dans la liste de valeurs renvoyées par `m//` en contexte de liste) sont entachés. Cela est utile quand les opérations de regex sur des données entachées sont destinées non pas à extraire des sous-chaînes sûres, mais plutôt pour faire de nouvelles transformations. Voir le chapitre 23, *Sécurité*, pour en savoir plus sur l'entachement. Pour ce pragma, les expressions régulières précompilées (obtenues en général par `qr//`) ne sont pas considérées comme interpolées :

```
/toto${motif}titi/
```

Ceci est autorisé si `$motif` est une expression régulière précompilée, même si `$motif` contient des sous-motifs `(?{CODE})`.

Précédemment nous vous avons montré un peu de ce que `use re 'debug'` affiche. Une solution de déboguage un peu plus primitive consiste à utiliser des sous-motifs `(?{CODE})` pour afficher ce qui a été trouvé jusqu'ici dans la recherche :

```
"abcdef" =~ / .+ (?{print "Trouvé jusqu'à présent : $&\n"}) bcdef $/x;
```

Ceci affiche :

```
Trouvé jusqu'à présent : abcdef
Trouvé jusqu'à présent : abcde
Trouvé jusqu'à présent : abcd
Trouvé jusqu'à présent : abc
Trouvé jusqu'à présent : ab
Trouvé jusqu'à présent : a
```

ce qui montre `.+` gobant toutes les lettres et les rendant une par une au fur et à mesure que le Moteur rebrousse chemin.

Interpolation de motif lors de la recherche

Vous pouvez construire des morceaux de votre motif depuis l'intérieur du motif lui-même. L'extension (`??{`*`CODE`*`}`) vous permet d'insérer du code qui s'évalue en un motif valide. C'est comme écrire `/$motif/`, sauf que vous pouvez générer `$motif` à l'exécution — plus spécifiquement, lors de la recherche. Par exemple :

```
/\w (??{ if ($seuil > 1) { "rouge" } else { "bleu" } }) \d/x;
```

C'est équivalent à `/\wrouge\d/` si `$seuil` est plus grand que 1 et à `/\wbleu\d/` sinon.

Vous pouvez inclure des références arrières à l'intérieur du code évalué pour déduire de motifs de chaînes trouvées à l'instant (même si elles ne seront plus trouvées plus tard pour cause de retour arrière). Par exemple, ceci trouve toutes les chaînes qui se lisent de la même façon à l'envers et à l'endroit (plus connues sous le nom de palindromadaires, ce sont des phrases avec une bosse au milieu) :

```
/^ (.+) .? (??{quotemeta reverse $1}) $/xi;
```

Vous pouvez équilibrer des parenthèses comme suit :

```
$texte =~ /( \(+ ) (.*?) (??{ '\)' x length $1 })/x;
```

Ceci trouve les chaînes de la forme `(shazam!)` et `(((shazam!)))`, et met `shazam!` dans `$2`. Hélas, il est incapable de savoir si les parenthèses au milieu sont équilibrées. Pour cela nous avons besoin de récursion.

Heureusement, vous pouvez aussi faire des motifs récursifs. Vous pouvez avoir un motif compilé utilisant (`??{`*`CODE`*`}`) et faisant référence à lui-même. La recherche récursive est plus irrégulière, pour des expressions régulières. Tous les textes sur les expressions régulières vous diront qu'une expression rationnelle standard ne peut pas trouver des parenthèses équilibrées correctement. Et c'est correct. Il est aussi vrai que que les regex de Perl ne sont pas standard. Le motif suivant[21] trouve un ensemble de parenthèses équilibrées, quelle que soit leur profondeur :

```
$pe = qr{
          \(
          (?:
             (?> [^()]+ )    # Non parenthèses sans retour arrière
           |
             (??{ $pe })     # Groupe avec des parenthèses équilibrées
          )*
```

21. Remarquez que vous ne pouvez pas déclarer la variable dans l'instruction où vous allez l'utiliser. Vous pouvez toujours la déclarer avant, bien sûr.

```
            \)
        }x;
```

Vous pourriez l'utiliser comme ceci pour trouver un appel de fonction :

```
$foncmotif = qr/\w+$pe/;
'mafonc(1,(2*(3+4)),5)' =~ /^$foncmotif$/;   # Trouvé !
```

Interpolation conditionnelle

L'extension de regex `(?(`*`COND`*`)`*`SIVRAI`*`|`*`SIFAUX`*`)` ressemble à l'opérateur `?:` de Perl. Si *`COND`* est vrai, le motif *`SIVRAI`* est utilisé ; sinon le motif *`SIFAUX`* est utilisé. *`COND`* peut être une référence arrière (exprimé comme un entier simple, sans le `\` ou le `$`), une assertion périphérique ou un sous-motif de code (voir *Assertions périphériques* et *Évaluation de code lors de la recherche* plus haut dans ce chapitre).

Si *`COND`* est un entier, il est traité comme une référence arrière. Par exemple :

```
#!/usr/bin/perl
$x = 'Perl est gratuit.';
$y = 'PatronGiciel coûte 99EUR.';

foreach ($x, $y) {
    /^(\w+) (?:est|(coûte)) (?(2)(\d+EUR)|\w+)/; # Soit (\d+EUR), soit \w+
    if ($3) {
        print "$1 coûte de l'argent.\n";         # PatronGiciel coûte
                                                 # de l'argent.
    } else {
        print "$1 ne coûte rien.\n";             # Perl ne coûte rien.
    }
}
```

Ici, *`COND`* est `(2)`, qui est vrai s'il existe une deuxième référence arrière. Si c'est le cas, `(\d+EUR)` est ajouté au motif (et crée la référence arrière `$3`) ; sinon, `\w+` est utilisé.

Si *`COND`* est un assertion périphérique ou un sous-motif de code, la véracité de l'assertion est utilisée pour déterminer s'il faut inclure *`SIVRAI`* ou *`SIFAUX`* :

```
/[ATGC]+(?(?<=AA)G|C)$/;
```

Ceci utilise une assertion de rétrovision comme *`COND`* pour trouver une séquence d'ADN qui se termine soir par `AAG` ou une autre combinaison de bases et un `C`.

Définir vos propres assertions

Vous ne pouvez pas modifier le fonctionnement du Moteur de Perl, mais si vous êtes suffisament cinglé, vous pouvez changer la façon dont il voit votre motif. Comme Perl interprète votre motif de manière analogue à une chaîne entre apostrophes doubles, vous pouvez utiliser les merveilles de la surcharge des constantes chaîne pour vous arranger pour que les suites de caractères de votre choix soient automatiquement transformées en d'autres suites de caractères.

Dans l'exemple qui suit, nous définissons deux transformations qui doivent se produire quand Perl rencontre un motif. D'abord nous définissons `\tag` de façon à ce que lorsqu'il apparaît dans un motif, il est automatiquement transformé en `(?:<.*?>)`, qui trou-

ve la plupart des balises HTML et XML. Ensuite, nous « redéfinissons » le métasymbole `\w` de façon à ce qu'il n'accepte que les lettres anglaises.

Nous allons définir un paquetage nommé `Tagger` qui cache la surcharge à notre programme principal. Une fois que cela est fait, nous pourrons écrire :

```
use Tagger;
$_ = '<I>chameau</I>';
print "Balise chameau trouvée" if /\tag\w+\tag/;
```

Voici *Tagger.pm*, soit la forme d'un module Perl (voir chapitre 11) :

```
package Tagger;
use overload;

sub import { overload::constant 'qr' => \&convert }

sub convert {
    my $re = shift;
    $re =~ s/ \\tag  /<.*?>/xg;
    $re =~ s/ \\w    /[A-Za-z]/xg;
    return $re;
}

1;
```

Le module `Tagger` reçoit le motif immédiatement avant l'interpolation, aussi pouvez vous sauter la surcharge en sautant l'interpolation, comme suit :

```
$re = '\tag\w+\tag';   # Cette chaîne commence avec \t, une tabulation
print if /$re/;        # Trouve une tabulation, suivie d'un "a"...
```

Si vous vouliez personnaliser la variable interpolée, appelez la fonction `convert` directement :

```
$re = '\tag\w+\tag';         # Cette chaîne commence avec \t, une tabulation
$re = Tagger::convert $re;   # convertit \tag et \w
print if /$re/;              # $re devient <.*?>[A-Za-z]+<.*?>
```

Et maintenant, si vous vous demandez toujours ce que sont ces fameuses `sub` dans le module `Tagger`, vous allez le découvrir bien vite, car c'est le sujet du chapitre suivant.

6

Sous-programmes

Comme beaucoup de langages, Perl fournit des sous-programmes définis par l'utilisateur.[1] Ces sous-programmes peuvent être définis n'importe où dans le programme principal, chargés depuis d'autres fichiers grâce aux mots-clés `do`, `require` ou `use` ou générés à l'exécution avec `eval`. Vous pouvez même les charger à l'exécution avec le mécanisme décrit dans la section *Autochargement* du chapitre 10, *Paquetages*. Vous pouvez appeler un sous-programme indirectement, en utilisant une variable contenant son nom ou une référence à la routine ou à travers un objet, en laissant l'objet déterminer quel sous-programme doit vraiment être appelé. Vous pouvez générer des sous-programmes anonymes, accessibles seulement à travers des références, et si vous voulez, vous en servir pour cloner de nouvelles fonctions quasiment identiques à l'aide de *fermetures*, qui sont décrites dans la section du même nom du chapitre 8, *Références*.

Syntaxe

Pour déclarer un sous-programme nommé sans le définir, utilisez l'une de ces formes :

```
sub NOM
sub NOM PROTO
sub NOM       ATTRS
sub NOM PROTO ATTRS
```

Pour déclarer et définir un sous-programme, ajoutez un *BLOC* :

```
sub NOM               BLOC
sub NOM PROTO         BLOC
sub NOM       ATTRS   BLOC
sub NOM PROTO ATTRS   BLOC
```

1. Nous les appellerons aussi *fonctions*, mais les fonctions sont identiques aux sous-programmes en Perl. Parfois nous les appellerons même *méthodes*, qui sont définies de la même façon, mais appelées différemment.

Pour créer un sous-programme anonyme ou une fermeture, omettez le *NOM* :

```
sub                    BLOC
sub       PROTO        BLOC
sub             ATTRS  BLOC
sub       PROTO ATTRS  BLOC
```

PROTO et *ATTRS* représentent le prototype et les attributs, chacun d'entre eux étant présenté dans sa propre section plus loin dans ce chapitre. Ils ne sont pas si importants ; le *NOM* et le *BLOC* sont les parties essentielles, y compris quand elles sont absentes.

Pour les formes sans *NOM*, vous devez toujours fournir une manière d'appeler le sous-programme. Assurez-vous donc de sauvegarder la valeur de retour, car non seulement cette forme de déclaration de `sub` est compilée comme vous pouviez vous y attendre, mais elle fournit également une valeur de retour à l'exécution :

```
$refsub = sub BLOCK;
```

Pour importer des sous-programmes définis dans un autre module, écrivez :

```
use MODULE qw(NOM1 NOM2 NOM3...);
```

Pour appeler les sous-programmes directement, écrivez :

```
NOM(LISTE)          # & est optionel avec des parenthèses.
NOM LISTE           # Parenthèses facultatives si sub prédéclarée/importée.
&NOM                # Passe le @_ courant à ce sous-programme
                    # (et contourne les prototypes).
```

Pour appeler des sous-programmes de façon indirecte (par leur nom ou par référence), utilisez n'importe laquelle de ces formes :

```
&$refsub(LISTE)     # Le & n'est pas facultatif pour un appel indirect
$refsub->(LISTE)    # (sauf en utilisant une notation infixe).
&$refsub            # Passe le @_ courant à ce sous-programme.
```

Le nom officiel d'un sous-programme comprend le préfixe `&`. Un sous-programme peut être appelé en utilisant le préfixe, mais le `&` est en général optionnel, ainsi que les parenthèses si le sous-programme a été prédéclaré. Cependant, le `&` n'est pas optionnel quand vous ne faites que nommer le sous-programme, comme quand il est utilisé comme argument de `defined` ou `undef` ou quand vous voulez générer une référence à un sous-programme nommé en écrivant `$refsub = \&nom`. Le `&` n'est pas non plus optionnel quand vous voulez faire un appel de sous-programme indirect en utilisant l'une des constructions `&$refsub()` ou `&{$refsub}()`. Cependant, la notation la plus commode `$refsub->()` n'en a pas besoin. Voir le chapitre 8 pour en savoir plus au sujet des références de sous-programmes.

Perl n'impose pas de modèle particulier d'écriture en majuscules ou minuscules pour vos noms de sous-programme. Cependant une convention vaguement suivie est que les fonctions appelées indirectement par Perl à l'exécution (`BEGIN`, `CHECK`, `INIT`, `END`, `AUTOLOAD`, `DESTROY` et toutes les fonctions mentionnées au chapitre 14, *Variables liées*) sont toutes en majuscules, aussi vaut-il probablement mieux éviter ce style. (Mais les sous-programmes utilisés pour des valeurs constantes portent d'habitude un nom tout en capitales. C'est normal. Nous espérons...)

Sémantique

Avant de vous mettre dans tous vos états à cause de toute cette syntaxe, souvenez-vous que la manière normale de définir un simple sous-programme finit par ressembler à cela :

```
sub guincher {
    print "Vous voilà guinché.\n";
}
```

et que la manière normale de l'appeler est simplement :

```
guincher();
```

Dans ce cas, nous avons ignoré l'entrée (les arguments) et la sortie (les valeurs de retour). Mais les modèles Perl pour passer des données, à et depuis un sous-programme, sont vraiment assez simples : tous les paramètres de la fonction sont passés comme une seule liste à plat de scalaires, et s'il y a plusieurs valeurs de retour, elles sont de même retournées à l'appelant comme une seule liste à plat de scalaires. Comme pour toute *LISTE*, les tableaux et les hachages passés dans cette liste s'interpoleront dans la liste à plat, perdant par là même leur identité. Mais il existe plusieurs manières de contourner cela, et l'interpolation automatique de liste est souvent très utile. Les listes de paramètres, comme les listes de retour, peuvent contenir autant (ou aussi peu) d'éléments que vous voulez (bien que vous puissiez imposer certaines contraintes aux listes de paramètres en utilisant des prototypes). En effet, Perl est conçu autour de cette notion de fonctions *variadiques* (qui peuvent prendre n'importe quel nombre d'arguments) ; en C, au contraire, elles sont plus ou moins bidouillées à contrecœur de façon à ce que vous puissiez appeler *printf*(3).

Maintenant, si vous voulez un langage conçu pour passer un nombre variable d'arguments arbitraires, vous devriez vous arranger pour qu'il soit facile de traiter ces listes arbitraires d'arguments. Tout argument passé à un sous-programme Perl est fourni dans le tableau `@_`. Si vous appelez une fonction avec deux arguments, ils sont accessibles à l'intérieur de la fonction comme les deux premiers éléments de ce tableau : `$_[0]` et `$_[1]`. Comme `@_` est un tableau ordinaire avec un nom extraordinaire, vous pouvez lui faire tout ce que vous pouvez normalement faire à un tableau.[2] Le tableau `@_` est un tableau local, mais ses valeurs sont des alias des véritables paramètres scalaires. (C'est connu sous le nom de sémantique de passage par référence.) Vous pouvez donc modifier les véritables paramètres si vous modifiez l'élément correspondant de `@_` (c'est cependant rarement fait, tellement il est facile de renvoyer des valeurs intéressantes en Perl).

La valeur de retour du sous-programme (ou de n'importe quel bloc, en fait) est la valeur de la dernière expression évaluée. Vous pouvez aussi utiliser une instruction `return` explicite pour spécifier la valeur de retour et sortir du sous-programme depuis n'importe quel point à l'intérieur. Dans les deux cas, si la fonction est appelée en contexte scalaire ou de liste, l'expression finale de cette fonction est appelée dans ce même contexte.

2. C'est un domaine où Perl est *plus* orthogonal que la plupart des autres langages de programmation.

Trucs avec la liste de paramètres

Perl n'a pas encore de paramètres formels nommés, mais en pratique tout ce que vous avez besoin de faire, c'est de copier les valeurs de `@_` dans une liste `my`, qui sert parfaitement de liste de paramètres formels. (Ce n'est pas une coïncidence si la copie des valeurs change un passage par référence en passage par valeur, ce qui est le fonctionnement que les gens attendent habituellement, même s'ils ne connaissent pas le vocabulaire de l'informatique théorique.) Voici un exemple typique :

```
sub setenv_peut_etre {
    my ($cle, $valeur) = @_;
    $ENV{$cle} = $valeur unless $ENV{$cle};
}
```

Mais vous n'êtes pas obligé de nommer vos paramètres, ce qui est tout l'intérêt du tableau `@_`. Par exemple, pour calculer un maximum, vous pouvez juste itérer directement sur `@_` :

```
sub max {
    my $max = shift(@_);
    for my $element (@_) {
        $max = $element if $max < $element;
    }
    return $max;
}

$meilleurjour = max($lun,$mar,$mer,$jeu,$ven);
```

Ou vous pouvez remplir tout un hachage d'un coup :

```
sub configuration {
    my %options = @_;
    print "Verbosité maximale.\n" if $options{VERBOSE} == 9;
}

configuration(PASSWORD => "xyzzy", VERBOSE => 9, SCORE => 0);
```

Voici un exemple où vos arguments formels ne sont pas nommés afin de pouvoir modifier les arguments eux-mêmes :

```
majuscules_sur_place($v1, $v2);  # ceci change $v1 et $v2
sub majuscules_sur_place {
    for (@_) { tr/a-z/A-Z/ }
}
```

Bien sûr, vous n'avez pas le droit de modifier des constantes de cette façon. Si l'un des arguments était en fait un scalaire comme `"hobbit"` ou une variable en lecture seule comme `$1` et que vous essayiez de le modifier, Perl lèverait une exception (vraisemblablement fatale et menaçant potentiellement votre carrière). Par exemple, ceci ne marchera pas :

```
majuscules_sur_place("philippe");
```

Ce serait beaucoup plus sûr si la fonction `majuscules_sur_place` était écrite de façon à renvoyer des copies de ses paramètres, au lieu de les modifier directement :

```
($v3, $v4) = majuscules($v1, $v2);
sub majuscules {
    my @parms = @_;
    for (@parms) { tr/a-z/A-Z/ }
    # Vérifie si nous avons été appelés en contexte de liste.
    return wantarray ? @parms : $parms[0];
}
```

Remarquez comme cette fonction (sans prototype) se moque de savoir si on lui a passé de véritables scalaires ou des tableaux. Perl aplatira tout dans une grande liste à plat de paramètres dans `@_`. C'est l'un des cas où Perl brille par son style simple de passage de paramètres. La fonction `majuscules` fonctionnera parfaitement bien sans changer sa définition, même si on lui passe des choses comme ça :

```
@nouvelleliste = majuscules(@liste1, @liste2);
@nouvelleliste = majuscules( split /:/, $var );
```

Ne vous laissez cependant pas tenter par cela :

```
(@a, @b) = majuscules(@liste1, @liste2);   # FAUX
```

Pourquoi ? Parce que, tout comme la liste des paramètres entrants dans `@_` est plate, la liste de retour est aussi à plat. Ceci stocke donc tout dans `@a`, et vide `@b` en y stockant la liste vide. Voir la section *Passage de références* pour d'autres manières de s'y prendre.

Indications d'erreur

Si vous voulez que votre fonction se termine de façon à ce que l'appelant réalise qu'une erreur s'est produite, la manière la plus naturelle de faire est d'utiliser un `return` simple sans argument. De cette façon, quand la fonction est appelée en contexte scalaire, l'appelant reçoit `undef` et quand elle utilisée en contexte de liste, l'appelant reçoit une liste vide.

Dans des circonstances extraordinaires, vous pourriez choisir de lever une exception pour indiquer une erreur. Utilisez cependant cette mesure avec parcimonie, sinon votre programme va se retrouver envahi de gestionnaires d'exception. Par exemple, ne pas réussir à ouvrir un fichier dans une fonction générique d'ouverture de fichier n'est pas vraiment un événement exceptionnel. En revanche ignorer une telle erreur en serait vraiment un. La fonction intégrée `wantarray` retourne `undef` si votre fonction a été appelée en contexte vide ; vous pouvez donc savoir si la valeur de retour de votre fonction est ignorée :

```
if ($quelquechose_a_plante) {
    return if defined wantarray;  # bon, ce n'est pas un contexte vide
    die "Fais attention à mon erreur, espèce d'inconscient !!!\n";
}
```

Problèmes de portée

Les sous-programmes peuvent être appelés de façon récursive, car chaque appel reçoit son propre tableau d'arguments, même quand la routine s'appelle elle-même. Si un sous-programme est appelé en utilisant la forme `&`, la liste d'arguments est optionnelle. Si la forme en `&` est utilisée, mais que la liste d'arguments est omise, quelque chose de spécial se produit : le tableau `@_` de la routine appelante est fourni implicitement. Il

s'agit d'un mécanisme d'efficacité que les nouveaux utilisateurs préféreront éviter.

```
&toto(1,2,3);    # passe trois arguments
toto(1,2,3);     # pareil

toto();          # passe une liste vide
&toto();         # pareil

&toto;           # toto() reçoit les arguments courants, comme toto(@_),
                 # mais plus vite !
toto;            # comme toto() si sub toto est prédéclaré, sinon le mot
                 # simple "toto"
```

Non seulement la forme & rend la liste d'arguments optionnelle, mais elle désactive également toute vérification de prototypage sur les arguments fournis. Ceci en partie pour des raisons historiques et en partie pour fournir un moyen efficace de tricher si vous savez ce que vous faites. Voir la section *Prototypes* plus loin dans ce chapitre.

Les variables qui sont utilisées dans la fonction, mais qui n'ont pas été déclarées privées à cette fonction, ne sont pas nécessairement des variables globales ; elles continuent à suivre les règles normales de portée de Perl. Comme expliqué dans la section *Noms* du chapitre 2, *Composants de Perl*, cela signifie qu'elles cherchent d'abord à être résolues dans la ou les portées lexicales englobantes, puis ensuite dans la portée du paquetage. Du point de vue d'un sous-programme, toute variable my d'une portée lexicale englobante est donc parfaitement visible.

Par exemple, la fonction bumpx, ci-dessous, a accès à la variable lexicale $x (à portée sur tout le fichier) car la portée dans laquelle ce my a été déclaré — le fichier lui-même — n'a pas été fermée avant la définition du sous-programme :

```
# début du fichier
my $x = 10;         # déclare et initialise la variable
sub bumpx { $x++ }  # la fonction peut voir la variable lexicale externe
```

Les programmeurs C et C++ verraient probablement $x comme une variable « statique de fichier ». Elle est privée en ce qui concerne les fonctions dans les autres fichiers, mais globale du point de vue des fonctions déclarées après le my. Les programmeurs C qui viennent à Perl en cherchant ce qu'ils appeleraient des « variables statiques » pour des fichiers ou des fonctions ne trouvent pas un tel mot-clé. Les programmeurs Perl évitent en général le mot « statique », parce que les systèmes statiques sont morts et ennuyeux et que le mot est très chargé historiquement.

Bien que Perl ne dispose pas du mot « static » dans son dictionnaire, les programmeurs Perl n'ont aucun problème pour créer des variables qui restent privées à une fonction et persistent d'un appel de fonction à un autre. C'est juste qu'il n'existe pas de mot spécial pour décrire cela. Les primitives de portée de Perl sont plus riches et se combinent avec sa gestion automatique de mémoire de telle manière que quelqu'un cherchant le mot-clé « static » pourrait ne jamais penser à les utiliser.

Les variables lexicales ne passent pas automatiquement au ramasse-miettes simplement parce que l'on est sorti de leur portée ; elles attendent de ne plus être *utilisées* pour être recyclées, ce qui est bien plus important. Pour créer des variables privées qui ne sont pas automatiquement réinitialisées entre deux appels de fonction, incluez la fonction tout entière dans un bloc supplémentaire et mettez à la fois la déclaration my et la définition

de la fonction à l'intérieur de ce bloc. Vous pouvez même y mettre plus d'une fonction pour leur donner un accès partagé à une variable qui, sinon, est privée.

```
{
    my $compteur = 0;
    sub next_compteur { return ++$compteur }
    sub prev_compteur { return --$compteur }
}
```

Comme toujours, l'accès à la variable lexicale est limité au code se trouvant dans la même portée lexicale. Les noms des deux fonctions sont cependant globalement accessibles (à l'intérieur du paquetage) et, comme les fonctions ont été définies dans la portée de `$compteur`, elles peuvent tout de même accéder à cette variable bien que personne d'autre ne le puisse.

Si cette fonction est chargée par `require` ou `use`, alors c'est probablement parfait. Si tout se passe dans le programme principal, vous devrez vous assurer que toute affectation à `my` à l'exécution est exécutée suffisament tôt, soit en mettant tout le bloc avant votre programme principal, soit en plaçant un bloc `BEGIN` ou `INIT` autour pour être sûr qu'il sera exécuté avant que votre programme ne démarre :

```
BEGIN {
    my @gamme = qw/ do re mi fa sol la si /;
    my $note   = -1;
    sub ton_suivant { return $gamme[ ($note += 1) %= @gamme ] };
}
```

Le `BEGIN` n'affecte ni la définition du sous-programme, ni la persistence des lexicales utilisées par le sous-programme. Il est juste là pour s'assurer que les variables sont initialisées avant que la routine soit appelée. Pour plus d'information sur la déclaration de variables privées et globales, voir respectivement `my` et `our` au chapitre 29, *Fonctions*. Les constructions `BEGIN` et `INIT` sont expliquées au chapitre 18, *Compilation*.

Passage de références

Si vous voulez passer plus d'un tableau ou d'un hachage, depuis ou vers une fonction, tout en maintenant leur intégrité, vous aurez alors besoin d'un mécanisme explicite de passage de références. Avant cela, vous aurez besoin de comprendre les références ainsi qu'elles sont expliquées au chapitre 8. Faute de quoi, cette section risque de ne pas être très compréhensible. Enfin, vous pourrez toujours regarder les images...

Voici quelques exemples simples. Définissons d'abord une fonction qui attend une référence à un tableau. Quand le tableau est grand, il est beaucoup plus rapide de le passer comme une simple référence plutôt que comme une longue liste de valeurs :

```
$total = somme ( \@a );

sub somme {
    my ($reftab) = @_;
    my ($total)  = 0;
    foreach (@$reftab) { $total += $_ }
    return $total;
}
```

Passons plusieurs tableaux à une fonction, et faisons-la dépiler (*via* pop) chacun d'entre eux, en renvoyant une nouvelle liste de leurs derniers éléments :

```
@derniers = popplus ( \@a, \@b, \@c, \@d );

sub popplus {
    my @retliste = ();
    for my $reftab (@_) {
        push @retliste, pop @$reftab;
    }
    return @retliste;
}
```

Voici comment vous pourriez écrire une fonction qui ferait une sorte d'intersection, en retournant la liste des clés apparaissant dans tous les hachages qui lui sont passés :

```
@commun = inter( \%toto, \%titi, \%tutu );
sub inter {
    my %vus;
    for my $href (@_) {
        while (my $k = each %$href ) {
            $vus{$k}++;
        }
    }
    return grep { $vus{$_} == @_ } keys %vus;
}
```

Jusqu'ici nous n'avons fait qu'utiliser le mécanisme normal de retour de liste. Que se passe-t-il si vous voulez passer ou renvoyer un hachage ? S'il ne s'agit que d'un seul ou que cela ne vous gêne pas qu'ils soient concaténés, les conventions d'appel habituelles conviennent, bien qu'elles soient un peu coûteuses.

Comme nous l'avons expliqué précédemment, les gens commencent à avoir des problèmes avec :

```
(@a, @b) = func(@c, @d);
```

ou bien :

```
(%a, %b) = func(%c, %d);
```

Cette syntaxe ne fonctionne tout simplement pas. Elle se contente de tout affecter à @a ou %a tout en vidant @b ou %b. De plus la fonction ne reçoit pas deux tableaux ou deux hachages séparés : elle reçoit une longue liste dans @_, comme toujours.

Vous pouvez vous arranger pour que votre fonction prenne des références en paramètre et en renvoie en résultat. Voici une fonction qui prend deux références de tableau en argument et renvoie les deux références de tableaux ordonnées selon le nombre d'éléments qu'ils contiennent.

```
($aref, $bref) = func(\@c, \@d);
print "@$aref en a plus que @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
```

```
    } else {
        return ($dref, $cref);
    }
}
```

Pour passer des handles de fichier ou de répertoire à des fonctions ou si vous voulez qu'elles en retournent, voir les sections *Références de handles* et *Références de table de symboles* au chapitre 8.

Prototypes

Perl vous permet de définir vos propres fonctions de façon à ce qu'elles soient appelées comme les fonctions internes de Perl. Considérez la fonction `push(@tableau, $element)`, qui doit tacitement recevoir une référence à `@tableau`, et pas seulement les valeurs de la liste contenues dans `@tableau`, afin que le tableau soit effectivement modifié. Les *prototypes* vous permettent de déclarer des sous-programmes qui prennent leurs arguments comme beaucoup de fonctions internes, c'est-à-dire avec certaines contraintes au niveau du nombre et du type des arguments. Nous les appelons *prototypes* mais ils fonctionnent comme des modèles automatiques du contexte d'appel plutôt que de la façon à laquelle penseraient des programmeurs C ou Java quand on leur parle de prototypes. Avec ces modèles, Perl ajoutera automatiquement des antislashs implicites ou des appels à `scalar` ou ce qu'il faut pour que les choses se présentent de manière à respecter le modèle fourni. Par exemple, si vous déclarez :

```
sub monpush (\@@);
```

alors `monpush` prendra ses arguments exactement comme `push` les prend. Pour que cela fonctionne, il faut que la déclaration de la fonction soit visible à la compilation. Le prototype n'affecte l'interprétation des appels de fonction que quand le caractère `&` est omis. En d'autres termes, si vous l'appelez comme une fonction intégrée, elle se comporte comme un fonction intégrée. Si vous l'appelez comme une bonne vieille routine, elle se comporte comme une bonne vieille routine. Le `&` supprime les vérifications de prototype et les effets contextuels associés.

Comme les prototypes sont pris en considération seulement à la compilation, il en découle naturellement qu'ils n'ont aucune influence sur les références de sous-programme comme `\&toto` ou sur les appels de sous-programme indirects comme `&{$refsub}` ou `$refsub->()`. Les appels de méthode ne sont pas non plus influencés par les prototypes. C'est parce que la véritable fonction à appeler est indéterminée au moment de la compilation, puisqu'elle dépend de l'héritage, qui est calculé dynamiquement en Perl.

Comme il s'agit avant tout de permettre de définir des sous-programmes qui fonctionnent comme les commandes internes, voici quelques prototypes de fonctions que vous pourriez utiliser pour émuler le fonctionnement des fonctions internes correspondantes :

Déclaré comme	**Appelé par**
`sub monlink ($$)`	`monlink $ancien, $nouveau.`
`sub monreverse (@)`	`monreverse $a,$b,$c.`
`sub monjoin ($@)`	`monjoin ":",$a,$b,$c.`
`sub monpop (\@)`	`monpop @array.`

Déclaré comme	Appelé par
`sub monsplice (\@$$@)`	`monsplice @tableau,@tableau,0,@pushmoi.`
`sub monkeys (\%)`	`monkeys %{$hashref}.`
`sub monpipe (**)`	`monpipe READHANDLE, WRITEHANDLE.`
`sub monindex ($$;$)`	`monindex &getstring, "substr".`
	`monindex &getstring, "substr", $start.`
`sub monsyswrite (*$;$$)`	`monsyswrite OUTF, $buf.`
	`monsyswrite OUTF, $buf, length($buf)-$off, $off.`
`sub monopen (*;$@)`	`monopen HANDLE.`
	`monopen HANDLE, $name.`
	`monopen HANDLE, "-|", @cmd.`
`sub mongrep (&@)`	`mongrep { /toto/ } $a,$b,$c.`
`sub monrand ($)`	`monrand 42.`
`sub montime ()`	`montime.`

Tout caractère de prototype précédé d'un antislash (entre parenthèses dans la colonne de gauche ci-dessus) représente un argument réel (dont la colonne de droite donne un exemple) qui doit impérativement commencer par ce caractère. Le premier argument de `monkeys` doit donc commencer par un `%`, tout comme le premier argument de `keys`.

Un point-virgule sépare les arguments obligatoires des arguments optionnels. (Il serait redondant devant un `@` ou un `%` puisque les listes peuvent être vides.) Les caractères de prototype sans antislash ont une signification spéciale. Tout caractère `@` ou `%` consomme tout le reste des arguments et force un contexte de liste. (Il est équivalent à *`LISTE`* dans un diagramme de syntaxe.) Un argument représenté par `$`force un contexte scalaire. Un `&` requiert une référence à un sous-programme nommé ou anonyme.

Une * permet au sous-programme d'accepter dans cet emplacement tout ce qui serait accepté comme un handle de fichier par une fonction interne : un nom simple, une constante, une expression scalaire, un typeglob ou une référence à un typeglob. Si vous voulez toujours convertir un tel argument en une référence de typeglob, utilisez `Symbol::qualify_to_ref` comme suit :

```
use Symbol 'qualify_to_ref';

sub toto (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

Remarquez que les trois derniers exemples dans le tableau sont traités d'une manière spéciale par l'analyseur. `mongrep` est analysé comme un véritable opérateur de liste, `monrand` est analysé comme un véritable opérateur unaire avec la même précédence unaire que `rand` et `montime` ne prend vraiment aucun argument, comme `time`.

C'est-à-dire que si vous écrivez :

```
montime +2;
```

vous aurez `montime() + 2` et non `montime(2)`, qui serait le résultat d'une analyse sans prototype ou avec un prototype unaire.

L'exemple `mongrep` illustre aussi comment `&` est traité spécialement quand il est le premier argument. D'ordinaire, un prototype en `&` nécessiterait un argument comme `\&toto` ou `sub{}`. Cependant, quand il est le premier argument, vous pouvez supprimer le `sub` de votre sous-programme anonyme et simplement passer le bloc simple à l'emplacement de l'« objet indirect » (sans virgule après). L'un des aspects attrayant du prototype `&` est qu'il permet de générer une nouvelle syntaxe, tant que le `&` est en première position :

```
sub try (&$) {
    my ($try, $catch) = @_;
    eval { &$try };
    if ($@) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
}                   # ce n'est pas la fin de l'appel de fonction !
catch {
    /phooey/ and print "unphooey\n";
};
```

Ceci affiche « `unphooey` ». Ici, `try` est appelé avec deux arguments, la fonction anonyme `{die "phooey";}` et la valeur de retour de la fonction `catch`, qui dans ce cas n'est rien d'autre que son propre argument, tout le bloc d'une autre fonction anonyme. Dans le `try`, la première fonction en argument est appelée sous la protection d'un bloc `eval` pour intercepter tout ce qui pourrait exploser. Si quelque chose explose effectivement, la seconde fonction est appelée avec une version locale de la variable globale `$_` contenant l'exception qui a été levée.[3] Si tout cela est du chinois pour vous, vous allez devoir vous reporter à `die` et `eval` au chapitre 29, puis vous informer au sujet des fonctions anonymes et des fermetures au chapitre 8. D'un autre côté, si tout cela vous intrigue, vous pouvez jeter un œil au module `Error` sur CPAN, qui utilise cette méthode pour implémenter une gestion d'exceptions minutieusement structurée avec les clauses `try`, `catch`, `except`, `otherwise` et `finally`.

Voici une réimplémentation de l'opérateur `grep` (l'opérateur interne est plus efficace, bien sûr) :

```
sub mongrep (&@) {
    my $coderef = shift;
```

3. Oui, il reste des questions en suspens par rapport à la visibilité de `@_`. Nous les ignorerons pour le moment. Mais si un jour nous rendons `@_` à portée lexicale, comme cela se fait actuellement dans les versions expérimentales de Perl avec des threads, ces sous-programmes anonymes pourront se comporter comme des fermetures.

```
    my @resultat;
    foreach $_ (@_) {
        push(@resultat, $_) if &$coderef;
    }
    return @resultat;
}
```

Certains préféreraient des prototypes complètement alphanumériques. Les caractères alphanumériques ont été intentionnellement laissés en dehors des prototypes dans le but avoué d'ajouter un jour des paramètres formels nommés. (Peut-être.) L'objectif majeur du mécanisme actuel est de permettre aux auteurs de modules d'imposer un certain nombre de vérifications à la compilation aux utilisateurs de leurs modules.

Substitution en ligne de fonctions constantes

Les fonctions prototypées avec `()`, ce qui signifie qu'elles ne prennent aucun argument, sont analysées comme la fonction interne `time`. Mieux, le compilateur traite de telles fonctions comme de bonnes candidates pour la mise en ligne. Si le résultat de cette fonction, après la passe d'optimisation et de précalcul des expressions constantes, est soit une constante, soit un scalaire à portée lexicale sans autre référence, alors cette valeur sera utilisée au lieu des appels à la fonction. Les appels faits avec *&NOM* ne sont cependant jamais mis en ligne, tout comme ils ne sont sujets à aucun autre effet de prototype. (Pour une méthode simple de déclaration de telles constantes, voir le pragma `use constant` au chapitre 31, *Modules de pragmas*.)

Les deux versions de ces fonctions calculant π seront mises en ligne par le compilateur :

```
sub pi ()     { 3.14159 }            # Pas exact, mais proche
sub PI ()     { 4 * atan2(1, 1) }    # Le mieux qu'on puisse obtenir
```

En fait, toutes les fonctions suivantes seront mises en ligne, car Perl peut tout déterminer à la compilation :

```
sub FLAG_TOTO ()     { 1 << 8 }
sub FLAG_TITI ()     { 1 << 9 }
sub FLAG_MASK ()     { FLAG_TOTO | FLAG_TITI }

sub OPT_BLORGH ()    { (0x1B58 & FLAG_MASK) == 0 }
sub BLORGH_VAL ()    {
    if (OPT_BLORGH)  { return 23 }
    else             { return 42 }
}

sub N () { int(BLORGH_VAL) / 3 }
BEGIN {                 # le compilateur exécute ce bloc à la compilation
    my $prod = 1;       # variable privée, persistante
    for (1 .. N) { $prod *= $_ }
    sub NFACT () { $prod }
}
```

Dans le dernier exemple, la fonction `NFACT` est mise en ligne car elle a un prototype vide et que la variable qu'elle retourne n'est pas modifiée par la fonction — et ne peut plus ensuite être modifiée par personne, puisqu'elle est dans une portée lexicale. Le compi-

lateur remplace donc toutes les utilisations de `NFACT` par cette valeur, qui a été précalculée à la compilation grâce au `BEGIN` qui l'entoure.

Si vous redéfinissez un sous-programme qui était susceptible d'être mis en ligne, vous aurez un avertissement obligatoire. (Vous pouvez vous servir de cet avertissement pour savoir si le compilateur a mis en ligne une routine en particulier.) L'avertissement est considéré suffisament important pour ne pas être optionnel, car les invocations précédemment compilées de la fonction continueront d'utiliser l'ancienne valeur de la fonction. Si vous voulez redéfinir la routine, assurez-vous qu'elle n'a pas été mise en ligne en retirant le prototype `()` (qui change la sémantique d'appel, aussi faites attention) ou en contrecarrant le mécanisme de mise en ligne d'une autre manière, comme par exemple :

```
sub non_enligne () {
    return 23 if $$;
}
```

Pour en savoir plus sur ce qui se passe durant les phases de compilation et d'exécution de votre programme, voyez le chapitre 18.

Précautions à prendre avec les prototypes

Il vaut probablement mieux réserver les prototypes aux nouvelles fonctions et ne pas les appliquer à d'anciennes fonctions. Ce sont des descripteurs de contexte, pas des protypes du C ANSI, il vous faut donc être extrêmement attentif à ne pas imposer silencieusement un contexte différent. Supposez par exemple que vous décidiez qu'une fonction ne doit prendre qu'un seul paramètre, comme ceci :

```
sub func ($) {
    my $n = shift;
    print "vous m'avez donné $n\n";
}
```

Cela la transforme en opérateur unaire (comme l'opérateur interne `rand`) et modifie la façon dont le compilateur détermine les arguments de la fonction. Avec le nouveau prototype, la fonction consomme un seul argument en contexte scalaire, au lieu de plusieurs arguments en contexte de liste. Si quelqu'un l'a appelée avec un tableau ou une expression de liste, même si ce tableau ou cette liste ne contenait qu'un seul élément, là où cela marchait vous aurez maintenant quelque chose de complètement différent :

```
func @toto;                # compte les éléments de @toto
func split /:/;            # compte le nombre d'éléments retournés
func "a", "b", "c";        # passe "a" seulement, le jette avec "b"
                           # et retourne "c"
func("a", "b", "c");       # soudain, une erreur du compilateur !
```

Vous venez juste de fournir un `scalar` implicite devant la liste d'arguments, ce qui peut être très surprenant. Ce n'est pas le vieux `@toto` qui contenait une seule valeur qui est passé. À la place, c'est 1 (le nombre d'éléments de `@toto`) qui est maintenant passé à `func`. Le `split`, étant appelé en contexte scalaire, scribouille partout dans votre liste de paramètres `@_`. Dans le troisième exemple, comme `func` a été prototypée comme un opérateur unaire, seul « `a` » lui est passé ; puis la valeur de retour de `func` est jetée tandis que l'opérateur virgule évalue les deux éléments suivants et retourne « `c` ». Dans le dernier exemple, l'utilisateur obtient maintenant une erreur de syntaxe à la compilation,

sur du code qui compilait et fonctionnait parfaitement.

Si vous écrivez du nouveau code et que vous voudriez un opérateur unaire qui ne prend qu'une variable scalaire, et pas une expression scalaire, vous pouvez toujours le prototyper pour qu'il prenne une référence à un scalaire :

```
sub func (\$) {
    my $nref = shift;
    print "vous m'avez donné $$nref\n";
}
```

Maintenant le compilateur ne laissera rien passer qui ne commence pas par un dollar :

```
func @toto;             # erreur de compilation, voit @, veut $
func split/:/;          # erreur de compilation, voit une fonction, veut $
func $s;                # celui-ci va bien ; on a eu un vrai $
func $a[3];             # celui-ci aussi
func $h{bazar}[-1];     # et même ça
func 2+5;               # expression scalaire, toujours une erreur de
                        # compilation
func ${ \(2+5) };       # ça marche, mais le remède n'est-il pas pire que
                        # le mal ?
```

Si vous ne faites pas attention, vous pouvez vous attirer des ennuis avec les prototypes. Mais si vous faites attention, vous pouvez faire tout un tas de trucs classe avec. Tout cela est bien sûr très puissant, et ne devrait être utilisé qu'avec modération pour rendre le monde meilleur.

Attributs de sous-programmes

Une déclaration ou une définition de sous-programme peut se voir associer une liste d'attributs. Si une telle liste d'attributs est présente, elle sera découpée en fonction des blancs ou des deux-points et traitée comme si un `use attributes` avait été vu. Voir le pragma `use attributes` au chapitre 31 pour des détails internes. Il existe trois attributs standards pour les sous-programmes : `locked`, `method` et `lvalue`.

Les attibuts locked et method

```
# Un seul thread est autorisé dans cette fonction.
sub afunc : locked { ... }

# Un seul thread est autorisé dans cette fonction pour un objet donné.
sub afunc : locked method { ... }
```

Mettre l'attribut `locked` n'a de sens que lorsque la fonction ou la méthode est supposée être appelée par plusieurs threads simultanément. Quand il est appliqué à une fonction qui n'est pas une méthode, Perl s'assure qu'un verrou est mis sur la routine elle-même avant que l'on n'y entre. Quand il est appliqué à une fonction qui est une méthode (c'est-à-dire une fonction également marquée avec l'attribut `method`), Perl s'assure que toute invocation de cette méthode verrouille implicitement son premier argument (l'objet) avant l'exécution.

La sémantique de ce verrou est la même qu'en utilisant l'opérateur `lock` dans un sous-programme en première instruction de ce sous-programme. Pour en savoir plus sur les verrous, voir le chapitre 17, *Threads*.

L'attribut `method` peut être utilisé tout seul :

```
sub afunc : method { ... }
```

À l'heure actuelle, cela n'a pour seul effet que de marquer la routine de façon à ne pas déclencher l'avertissement « `Ambiguous call resolved as CORE::%s` ». (Nous pourrons lui donner une signification plus importante un de ces jours.)

Le système d'attributs est extensible par l'utilisateur, ce qui vous permet de créer vos propres noms d'attribut. Ces nouveaux attributs doivent porter des noms d'identificateur valides (sans autre caractère de ponctuation que « _ »). Une liste de paramètres peut y être ajoutée, pour laquelle n'est vérifiée pour le moment que le bon équilibrage des parenthèses.

Voici des exemples de syntaxe valide (bien que les attributs soient inconnus) :

```
sub fnord (&\%) : switch(10,foo(7,3))  :  expensive;
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

Voici des exemples de syntaxe incorrecte :

```
sub fnord : switch(10,toto(); # chaîne () déséquilibrée
sub snoid : Laid('(');        # chaîne () déséquilibrée
sub xyzzy : 5x5;              # "5x5" n'est pas un identificateur valide
sub plugh : Y2::nord;         # "Y2::nord" n'est pas un identificateur
                              # simple
sub snurt : toto + titi;      # "+" n'est ni un espace ni un deux-points
```

La liste d'attributs est passée comme une liste de chaînes constantes au code qui les associe avec le sous-programme. La façon exacte dont cela fonctionne (ou ne fonctionne pas) est hautement expérimentale. Pour des détails actuels sur les listes d'attributs et leur manipulation, voir *attributes*(3).

L'attribut lvalue

Il est possible de renvoyer un scalaire modifiable depuis un sous-programme, mais seulement si vous déclarez que le sous-programme retourne une *lvalue* :

```
my $val;
sub modifiable : lvalue {
    $val;
}
sub nonmodifiable {
    $val;
}

modifiable()    = 5;    # Affecte à $valeur.
nonmodifiable() = 5;    # ERREUR
```

Si vous passez des paramètres à un sous-programme lvalué, vous aurez besoin de parenthèses pour lever l'ambiguïté sur ce qui est affecté :

```
modifiable $x   = 5;    # affecte 5 à $x avant !
modifiable 42   = 5;    # impossible de modifier une constante : erreur
                        # de compilation
modifiable($x)  = 5;    # ceci fonctionne
modifiable(42)  = 5;    # et cela aussi
```

Si vous voulez finasser, vous pouvez contourner cela dans le cas particulier d'une fonction ne prenant qu'un seul argument. Déclarer la fonction avec le prototype `($)` la fait analyser avec la précédence d'un opérateur unaire nommé. Comme les opérateurs unaires nommés ont une précédence supérieure à celle de l'affectation, vous n'avez plus besoin de parenthèses. (La question de savoir si cela est souhaitable ou pas est laissée à l'appréciation de la milice du style.)

En revanche, vous n'avez pas besoin de finasser dans le cas d'un sous-programme qui ne prend aucun argument (c'est-à-dire avec un prototype `()`). Vous pouvez sans ambiguïté écrire :

```
modifiable = 5;
```

Cela fonctionne car aucun terme valide ne commence par `=`. De même, vous pouvez omettre les parenthèses dans les appels de méthodes lvaluées quand vous ne passez aucun argument :

```
$obj->modifiable = 5;
```

Nous promettons de ne pas casser ces deux constructions dans les prochaines versions de Perl. Elles sont pratiques quand vous voulez encapsuler des attributs d'objet dans des appels de méthodes (de façon à ce qu'ils soient hérités comme des appels de méthodes, mais accessibles comme des variables).

Le contexte scalaire ou de liste du sous-programme avec l'attribut lvalue et de la partie droite d'une affectation à ce sous-programme est dans les deux cas déterminé en remplaçant l'appel de fonction par un scalaire. Considérez par exemple :

```
data(2,3) = get_data(3,4);
```

Les deux sous-programmes ont été appelés en contexte scalaire, tandis que dans :

```
(data(2,3)) = get_data(3,4);
```

et dans :

```
(data(2),data(3)) = get_data(3,4);
```

tous les sous-programmes sont appelés en contexte de liste.

L'implémentation actuelle ne permet pas à des tableaux ou des hachages d'être retournés directement par des sous-programme lvalués. Vous pouvez toujours renvoyer une référence à la place.

7

Formats

Perl dispose d'un mécanisme pour vous aider à générer des rapports et des tableaux simples. Ainsi, Perl vous aide à coder votre page de sortie d'une façon proche de ce à quoi elle ressemblera quand elle s'affichera. Il garde trace du nombre de lignes sur la page, du numéro de page courant, de quand imprimer les en-têtes de page et ainsi de suite. Les mots-clés sont empruntés à FORTRAN : `format` pour les déclarer et `write` pour les exécuter. Voir les sections correspondantes au chapitre 29, *Fonctions*. Heureusement la disposition est beaucoup plus lisible, plus proche du `PRINT USING` de BASIC. Vous pouvez le voir comme le *nroff*(1) du pauvre. (Si vous connaissez *nroff*, ça ne ressemble peut-être pas à un encouragement.)

Les formats, comme les paquetages et les sous-programmes, sont déclarés plutôt qu'exécutés, et peuvent donc apparaître n'importe où dans votre programme. (Bien qu'en général il vaille mieux les regrouper tous ensemble.) Ils ont leur propre espace de nommage différent de ceux de tous les autres types de Perl. Cela signifie que si vous avez une fonction nommée « `toto` », elle n'a rien à voir avec le format nommé « `toto` ». Cependant le nom par défaut du format associé à un handle de fichier donné est le même que le nom de ce handle. Le format par défaut de `STDOUT` est donc nommé « `STDOUT` », et le format par défaut du handle de fichier `TEMP` est également appelé « `TEMP` ». Ils ont l'air identiques. Mais ils ne le sont pas.

Les formats d'enregistrement de sortie sont déclarés de la manière suivante :

```
format NOM =
FORMLISTE
.
```

If *`NOM`* est omis, c'est le format `STDOUT` qui est défini. *`FORMLISTE`* est composée d'une série de lignes, chacun pouvant être de trois types :

- un commentaire, indiqué par une # en première colonne ;
- une ligne « image », donnant le format pour une ligne de sortie ;
- une ligne d'arguments fournissant les valeurs à insérer dans la ligne d'image précédente.

Les lignes images sont imprimées exactement comme elles apparaissent hormis certains champs auxquels sont substituées des valeurs.[1] Chaque champ de substitution dans une ligne image commence, soit par un signe `@`, soit par un `^`. Ces lignes ne subissent aucune forme d'interpolation de variable. Le champ `@` (à ne pas confondre avec le marqueur de tableau `@`) représente le type normal de champ ; l'autre type, `^`, est utilisé pour un remplissage rudimentaire de blocs de texte multilignes. La longueur du champ est indiquée en remplissant le champ avec plusieurs caractères `<`, `>` ou `|` pour indiquer respectivement la justification à gauche, à droite ou au centre. Si la variable excède la longueur spécifiée, elle est tronquée.

Vous pouvez également utiliser, comme autre forme de justification à droite, des caractères `#` (après un caractère `@` ou `^` initial) pour spécifier un champ numérique. Vous pouvez insérer un `.` à la place de l'un des caractères `#` pour aligner les points décimaux. Si l'une des valeurs fournies pour ces champs contient un saut de ligne, seul le texte avant ce saut de ligne est imprimé. Enfin, le champ spécial `@*` peut être utilisé pour imprimer des valeurs multilignes de manière non tronquée ; il devrait généralement apparaître seul sur une ligne image.

Les valeurs sont spécifiées sur la ligne suivante dans le même ordre que les champs images. Les expressions fournissant les valeurs doivent être séparées par des virgules. Les expressions sont toutes évaluées en contexte de liste avant que la ligne ne soit traitée, et une seule expression peut donc produire plusieurs éléments de liste. Les expressions peuvent s'étendre sur plusieurs lignes, si elles sont mises entre accolades. (Dans ce cas, l'accolade ouvrante doit être le premier token sur la première ligne.) Cela vous permet d'aligner les valeurs sous leurs champs de format respectifs pour faciliter la lecture.

Si l'évaluation d'une expression donne un nombre avec une partie décimale, et si l'image correspondante spécifie que la partie décimale doit apparaître dans la sortie (c'est-à-dire n'importe quelle image sauf une série de `#` sans `.` inclus), le caractère pour le point décimal est toujours déterminé par la valeur de la locale `LC_NUMERIC`. Cela signifie que si l'environnement d'exécution spécifie par exemple une locale française, une virgule sera utilisée plutôt qu'un point. Pour plus d'information, voir la page de manuel *perllocale*.

À l'intérieur d'une expression, les caractères blancs `\n`, `\t` et `\f` sont tous considérés comme équivalents à un espace seul. Vous pourriez voir cela comme si le filtre suivant était appliqué à chaque valeur dans le format :

```
$valeur =~ tr/\n\t\f/ /;
```

Le caractère blanc restant, `\r`, force l'affichage d'une nouvelle ligne si la ligne image le permet.

Les champs images qui commencent par `^` plutôt que `@` sont traités de manière spéciale. Avec un champ `#`, celui-ci est vide si la valeur est indéfinie. Pour les autres types de champ, le chapeau permet un genre de mode de remplissage. Au lieu d'une expression arbitraire, la valeur fournie doit être un nom de variable scalaire contenant une chaîne

1. Même ces champs conservent l'intégrité des colonnes où vous les mettez. Il n'y a rien dans une ligne image qui peut faire croître ou rétrécir ou se décaler un champ. Les colonnes que vous voyez sont sacrées au sens WYSIWYG (si vous utilisez une fonte à chasse fixe). Même les caractères de contrôle sont supposés avoir une largeur de un.

de caractères. Perl met tout le texte qu'il peut dans le champ, puis coupe le début de la chaîne de façon à ce que la fois suivante où la variable est référencée le reste du texte puisse être imprimé. (Oui, cela signifie que le contenu de la variable est altéré pendant l'exécution de l'appel à `write` et n'est pas préservé. Utilisez une variable temporaire si vous voulez préserver la valeur originale.) Normalement, vous devrez utiliser une séquence de champs alignés verticalement pour afficher un bloc de texte. Vous pouvez terminer le champ final par le texte « `...` », qui apparaîtra dans la sortie seulement si le texte était trop long pour être affiché entièrement. Vous pouvez modifier les caractères de « rupture » en modifiant la valeur de la variable `$:` (`$FORMAT_LINE_BREAK_CHARACTERS` si vous utilisez le module `English`) en une liste des caractères désirés.

Les champs ^ peuvent produire des enregistrements de longueur variable. Si le texte à formater est court, répétez quelques fois la ligne de format avec le champ ^. Si vous faites cela pour de courtes données, vous obtiendrez quelques lignes blanches. Pour supprimer les lignes qui finiraient blanches, mettez un caractère ~ (tilde) n'importe où sur la ligne. (Le tilde lui-même est transformé en espace à la sortie.) Si vous mettez un second tilde à la suite du premier, la ligne sera répétée jusqu'à ce que tout le texte des champs de cette ligne ait été affiché. (Cela fonctionne car tous les champs ^ avalent les chaînes qu'ils impriment. Mais si vous utilisez un champ de type `@` en conjonction avec deux tildes, vous avez intérêt à ce que l'expression fournie ne renvoie pas perpétuellement la même valeur ! Utilisez un `shift` ou un autre opérateur dont l'un des effets secondaires épuise toutes les valeurs.)

Le traitement des en-têtes de format est géré par défaut par un format de même nom que le handle de fichier courant, auquel `_TOP` a été concaténé. Il est déclenché au début de chaque page. Voir `write` au chapitre 29.

Voici quelques exemples :

```
# un rapport sur le fichier /etc/passwd
format STDOUT_TOP =
                        Fichier Passwd
Nom                 Login    Bureau   Uid   Gid Home
------------------------------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<<<<< @||||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
$nom,               $login,  $bureau,$uid,$gid, $home
.

# un rapport à partir d'un formulaire de rapport de bogues
format STDOUT_TOP =
                           Rapport de bogues
@<<<<<<<<<<<<<<<<<<<<<<<     @|||         @>>>>>>>>>>>>>>>>>>>>>>
$systeme,                     $%,         $date
------------------------------------------------------------------
.
format STDOUT =
Sujet: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
         $suject
```

```
Index: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
       $index,                       $description
Priorité: @<<<<<<<<<< Date: @<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
          $priorite,        $date,   $description
De: @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $de,                          $description
Assigné à: @<<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
             $programmeur,          $description
~                                   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                    $description
~                                   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                    $description
~                                   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                    $description
~                                   ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                    $description
~                                   ^<<<<<<<<<<<<<<<<<<<<<<...
                                    $description
.
```

Les variables lexicales ne sont pas visibles dans un format sauf si le format est déclaré dans la portée de la variable lexicale.

Il est possible de mélanger des appels à `print` et à `write` sur le même canal de sortie, mais il vous faudra gérer la variable spéciale `$-` (`$FORMAT_LINES_LEFT` si vous utilisez le module `English`) vous-même.

Variables de format

Le nom de format en cours est stocké dans la variable `$~` (`$FORMAT_NAME`), et le nom de l'en-tête de format en cours dans `$^` (`$FORMAT_TOP_NAME`). Le numéro de page de sortie courant est stocké dans `$%` (`$FORMAT_PAGE_NUMBER`), et le nombre de lignes dans la page se trouve dans `$=` (`$FORMAT_LINES_PER_PAGE`). Le vidage automatique du tampon de sortie de ce handle est positionné dans `$|` (`$OUTPUT_AUTOFLUSH`). La chaîne à sortie avant chaque en-tête de page est stockée dans `$^L` (`$FORMAT_FORMFEED`). Ces variables sont positionnées pour chaque handle de fichier, aussi devrez-vous faire un `select` sur le handle de fichier associé à un format pour modifier ses variables de format :

```
select((select(OUTF),
        $~ = "Mon_Autre_Format",
        $^ = "Mon_Entete_de_Format"
       )[0]);
```

C'est bien moche, hein ? C'est cependant un idiome commun, aussi ne soyez pas trop surpris en le voyant. Vous pouvez au moins utiliser une variable temporaire pour conserver le handle de fichier précédent :

```
$ofh = select(OUTF);
$~ = "Mon_Autre_Format";
$^ = "Mon_Entete_de_Format";
select($ofh);
```

Non seulement c'est une meilleure approche en général, car cela améliore la lisibilité du code, mais vous avez maintenant en plus une instruction intermédiaire où s'arrêter dans le code avec un debogueur en pas à pas. Si vous utilisez le module `English`, vous pouvez même lire le nom des variables :

```
use English;
$ofh = select(OUTF);
$FORMAT_NAME     = "Mon_Autre_Format";
$FORMAT_TOP_NAME = "Mon_Entete_de_Format";
select($ofh);
```

Mais il vous reste quand même ces appels bizarres à `select`. Si vous voulez les éviter, utilisez le module `FileHandle` fourni avec Perl. Maintenant vous pouvez accéder à ces variables spéciales en utilisant des noms de méthodes en minuscules à la place :

```
use FileHandle;
OUTF->format_name("Mon_Autre_Format");
OUTF->format_top_name("Mon_Entete_de_Format");
```

C'est quand même mieux !

Comme la ligne de valeurs qui suit votre ligne image peut contenir des expressions arbitraires (pour les champs `@`, pas pour les champs `^`), vous pouvez passer les traitements plus compliqués à d'autres fonctions, comme `printf` ou une de vos propres fonctions. Par exemple pour insérer des virgules dans un nombre :

```
format Ident =
    @<<<<<<<<<<<<<<<
    commify($n)
.
```

Pour afficher un vrai `@`, `~` ou `^` dans un champ, faites comme ceci :

```
format Ident =
J'ai un @ ici.
       "@"
.
```

Pour centrer tout une ligne de texte, faites quelque chose comme cela :

```
format Ident =
@|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
                         "Une ligne de texte"
.
```

L'indicateur de longueur de champ `>` garantit que le texte sera justifié à droite dans le champ, mais le champ lui-même se place exactement où vous l'avez placé. Il n'existe pas de manière intégrée de dire « mets ce champ à droite de la page, quelle que soit sa largeur ». Vous devez spécifier où il se trouve relativement à la marge de gauche. Ceux qui sont vraiment désespérés peuvent générer leur propre format au vol, en s'appuyant sur le nombre courant de colonnes (non fourni) et puis faire un `eval` :

```
$format  = "format STDOUT = \n"
         . '^' . '<' x $cols . "\n"
         . '$entry' . "\n"
         . "\t^" . "<" x ($cols-8) . "~~\n"
         . '$entry' . "\n"
```

```
          . ".\n";
print $format if $Debug;
eval $format;
die $@ if $@;
```

La ligne la plus importante est probablement celle du `print`. Ce qu'il imprime ressemble à ceci :

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$entry
    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
$entry
.
```

Voici un petit programme qui se comporte comme l'utilitaire Unix *fmt*(1) :

```
format =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ~~
$_

.

$/ = "";
while (<>) {
    s/\s*\n\s*/ /g;
    write;
}
```

Pieds de page

Alors que `$^` (`$FORMAT_TOP_NAME`) contient le nom du format d'en-tête courant, il n'existe pas de mécanisme correspondant pour les pieds de page. L'un des problèmes majeurs est de ne pas savoir quelle sera la taille d'un format avant de l'avoir évalué. C'est dans la liste des choses à faire.[2]

Voici une stratégie : si vous avez un pied de page de taille fixe, vous pouvez l'afficher en vérifiant `$-` (`$FORMAT_LINES_LEFT`) avant chaque `write` puis afficher le pied de page si nécessaire.

Voici une autre stratégie : ouvrez un tube sur vous-même, en utilisant `open(MOIMEME, "|-")` (voir l'entrée `open` au chapitre 29) et faites toujours des `write` sur `MOIMEME` au lieu de `STDOUT`. Le processus fils devra traiter son `STDIN` après coup pour réarranger ses entêtes et pieds de page à votre guise. Pas très pratique, mais faisable.

2. Bien sûr, cela ne garantit en rien que ce sera fait un jour. Les formats sont quelque chose d'un peu dépassé en cet âge du WWW, de l'Unicode, du XML, du XSLT et toutes les autres choses qu'on inventera après tout ça.

Accès au mécanisme interne de formatage

Pour accéder au mécanisme interne de formatage, vous pouvez utiliser l'opérateur intégré `formline` et accéder à `$^A` (la variable `$ACCUMULATOR`) directement. Par exemple :

```
$str = formline <<'END', 1,2,3;
@<<<  @|||  @>>>
END

print "Wouah, je viens juste de mettre `$^A' dans l'accumulateur !\n";
```

Ou pour créer un sous-programme `swrite` qui serait à `write` ce que `sprintf` est à `printf` :

```
use Carp;
sub swrite {
    croak "usage: swrite IMAGE ARGS" unless @_;
    my $format = shift;
    $^A = "";
    formline($format, @_);
    return $^A;
}

$chaine = swrite(<<'END', 1, 2, 3);
Vérifiez-moi
@<<<  @|||  @>>>
END
print $chaine;
```

Si vous utilisiez le module `FileHandle`, vous pourriez utiliser `formline` comme suit pour passer à la ligne à la colonne 72 :

```
use FileHandle;
STDOUT->formline("^" . ("<" x 72) . "~~\n", $long_texte);
```

8

Références

Pour des raisons tant philosophiques que pratiques, Perl a toujours favorisé les structures de données plates et linéaires. Et pour beaucoup de problèmes, c'est exactement ce dont vous avez besoin.

Imaginez que vous vouliez construire une table simple (un tableau à deux dimensions) listant certaines informations anthropométriques — âge, couleur des yeux, poids — concernant un groupe de personnes. Vous pourriez faire cela en créant un tableau pour chaque individu :

```
@john = (47, "marron", 84);
@mary = (23, "noisette", 58);
@bill = (35, "bleu",  71);
```

Vous pourriez ensuite construire un simple tableau supplémentaire contenant les noms des autres tableaux :

```
@infos = ('john', 'mary', 'bill');
```

Pour changer la couleur des yeux de John en « rouge » après une virée en ville, il nous faut un moyen de changer le contenu de `@john` à partir de la simple chaîne « `john` ». C'est le problème classique des indirections, que différents langages résolvent de différentes manières. En C, la forme la plus courante d'indirection est le pointeur, qui permet à une variable de contenir l'adresse en mémoire d'une autre variable. En Perl, la forme la plus courante d'indirection est la *référence*.

Qu'est-ce qu'une référence ?

Dans notre exemple, `$infos[0]` contient la valeur « `john` ». C'est-à-dire qu'il contient une chaîne qui se trouve être le nom d'une autre variable (globale). On dit que la première variable se *réfère* à la seconde ; cette sorte de référence est appelée *symbolique*, puisque Perl doit rechercher `@john` dans une table de symboles pour le retrouver. (Vous pouvez voir les références symboliques comme analogues aux liens symboliques d'un système de fichiers.) Nous parlerons des références symboliques un peu plus loin dans ce chapitre.

L'autre type de références sont les références en dur (*hard references*), qui constituent ce que les programmeurs Perl utilisent le plus souvent pour accomplir leurs indirections (quand ce ne sont pas leurs indiscrétions). Nous les appelons références en dur non pas parce qu'elles sont compliquées, mais parce qu'elles sont réelles et solides. Si vous préférez, vous pouvez voir les références en dur comme de vraies références et les références symboliques comme de fausses références. C'est un peu comme la différence entre avoir des amis et avoir des relations. Quand nous ne précisons pas de quel type de référence nous parlons, il s'agit de références en dur. La *figure 8-1* présente une variable nommée `$titi` faisant référence au contenu d'une variable nommée `$toto` qui a la valeur « `robot` ».

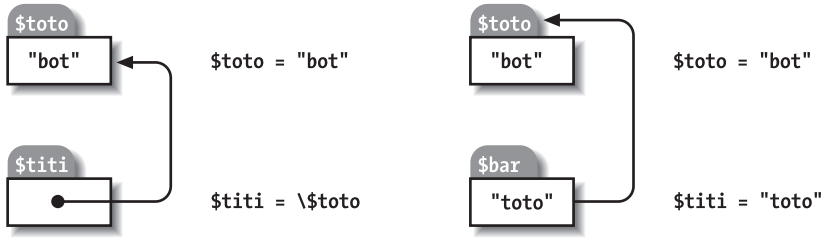


Figure 8-1. Une référence en dur et une référence symbolique

Contrairement aux références symboliques, une vraie référence ne se réfère non pas au nom d'une autre variable (qui n'est que le conteneur d'une valeur), mais à la valeur elle-même, un bout de donnée interne. Il n'y a pas de mot pour décrire cela, mais quand il le faut nous l'appellerons le *référent*. Supposez par exemple que vous créez une référence en dur vers le tableau à portée lexicale `@tableau`. Cette référence en dur, et le référent auquel elle se réfère, continueront d'exister même quand `@tableau` sera hors de portée. Un référent n'est détruit que quand toutes les références qui pointent sur lui ont été éliminées.

Un référent n'a pas vraiment de nom propre, en dehors des références qui s'y réfèrent. Pour le dire d'une autre façon, chaque variable Perl vit dans une sorte de table de symboles, qui contient une référence vers le référent sous-jacent (qui lui n'a pas de nom). Le référent peut être simple, comme un nombre ou une chaîne, ou complexe, comme un tableau ou un hachage. Dans tous les cas, il y a toujours exactement une référence de la variable à sa valeur. Vous pouvez éventuellement créer de nouvelles références en dur vers le même référent, mais si vous le faites, la variable elle-même n'en sait rien (et s'en moque).[1]

Une référence symbolique est juste une chaîne qui se trouve être le nom de quelque chose dans la table des symboles. Ce n'est pas tellement un type distinct, mais plutôt quelque chose que vous faites avec une chaîne. Alors qu'une référence en dur est une créature complètement différente. C'est le troisième des trois types scalaires fondamentaux, les deux autres étant les chaînes et les nombres. Une référence en dur n'a pas be-

1. Si vous êtes curieux, vous pouvez vous servir du module `Devel::Peek`, fourni avec Perl, pour déterminer la valeur du compteur de références sous-jacent.

soin de connaître le nom de quelque chose pour pointer dessus, et en fait il est complètement normal qu'il n'y ait *pas* de nom à utiliser du tout. De tels référents sans nom du tout sont *anonymes* ; nous en parlons dans la partie *Données anonymes* ci-dessous.

Référencer une valeur, dans la terminologie de ce chapitre, consiste à créer une référence en dur vers elle. (Il existe un opérateur spécial pour cette action créatrice.) La référence ainsi créée est un simple scalaire, qui se comporte dans les situations usuelles comme n'importe quel autre scalaire. *Déréférencer* ce scalaire signifie utiliser la référence pour obtenir le référent. Le référencement et le déréférencement se produisent quand vous invoquez des mécanismes explicites ; il ne se produit jamais de référencement ou de déréférencement implicites en Perl. Enfin, presque jamais.

Un appel de fonction *peut* passer implicitement des références — si elle a un prototype qui la déclare ainsi. Dans ce cas, l'appelant de la fonction ne passe pas explicitement une référence, bien qu'il vous faille toujours déréférencer explicitement celle-ci à l'intérieur de la fonction. Voir la section *Prototypes* au chapitre 6, *Sous-programmes*. Et pour être totalement honnête, il se produit aussi des déréférencements en coulisses quand vous utilisez certains types de handles de fichiers. Mais cela relève de la compatibilité antérieure et reste transparent pour l'utilisateur lambda. Finalement, deux fonctions internes, `bless` et `lock`, prennent toutes deux une référence en argument mais la déréférencent implicitement pour faire leurs manipulations sur ce qui se cache derrière. Ces indiscrétions admises, le principe selon lequel Perl n'a pas envie de toucher à vos niveaux d'indirections tient toujours.

Une référence peut pointer sur n'importe quelle structure de données. Comme les références sont des scalaires, vous pouvez les stocker dans des tableaux et des hachages, et ainsi construire des tableaux de tableaux, des tableaux de hachages, des hachages de tableaux, des tableaux de hachages de fonctions, et ainsi de suite. Vous trouverez des exemples de cela dans le chapitre 9, *Structures de données*.

Gardez cependant à l'esprit que les tableaux et les hachages de Perl sont à une dimension en interne. C'est-à-dire que leurs éléments ne peuvent contenir que des valeurs scalaires (chaînes, nombres et références). Quand nous disons quelque chose comme « tableau de tableaux », nous pensons en fait « tableau de références à des tableaux », tout comme nous pensons « hachage de références à des fonctions » quand nous disons « hachage de fonctions ». Mais comme les références sont la seule façon d'implémenter de telles structures en Perl, il s'ensuit que l'expression plus courte et moins précise n'est tout de même pas fausse, et ne devrait donc pas être totalement dénigrée, à moins que vous ne soyez maniaque.

Création de références

Il existe plusieurs façons de créer des références, dont nous décrirons la plupart avant d'expliquer comment utiliser (déréférencer) les références résultantes.

L'opérateur antislash

Vous pouvez créer une référence à une variable nommée ou à un sous-programme grâce à un antislash. (Vous pouvez aussi vous en servir sur une valeur scalaire anonymes

comme `7` ou `"chameau"`, bien que cela soit rarement nécessaire.) Cet opérateur fonctionne comme l'opérateur `&` (adresse de) en C — du moins à première vue.

Voici quelques exemples :

```
$refscalaire = \$toto;
$refconst    = \186_282.42;
$reftableau  = \@ARGV;
$refhash     = \%ENV;
$refcode     = \&handler;
$refglob     = \*STDOUT;
```

L'opérateur antislash peut faire bien plus que de créer de simples références. Il créera toute une liste de références s'il est appliqué à une liste. Pour plus de détails, voir la section *D'autres tours avec les références en dur*.

Données anonymes

Dans les exemples que nous venons de vous montrer, l'opérateur antislash se contente de dupliquer une référence qui est déjà contenue dans un nom de variable — avec une exception. Le `186_282.42` n'est référencé par aucune variable — c'est juste une valeur. C'est l'un de ces référents *anonymes* que nous avons mentionnés précédemment. On ne peut accéder aux référents anonymes qu'à travers des références. Celui-ci est un nombre, mais vous pouvez aussi bien créer des tableaux anonymes, des hachages anonymes et des sous-programmes anonymes.

Le composeur de tableaux anonymes

Vous pouvez créer une référence à un tableau anonyme en employant des crochets :

```
$reftableau = [1, 2, ['a', 'b', 'c', 'd']];
```

Ici nous avons construit un tableau anonyme de trois éléments, dont le dernier élément est une référence à un tableau anonyme à quatre éléments (décrit à la *figure 8-2*). (La syntaxe multidimensionnelle décrite plus loin peut être utilisée pour y accéder. Par exemple, `$reftableau->[2][1]` aurait la valeur « `b` ».)

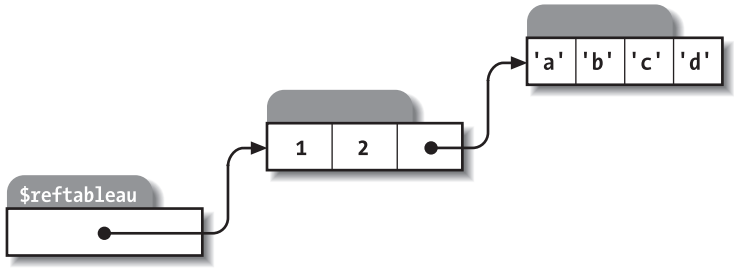


Figure 8-2. Une référence à un tableau, dont le troisième élément est lui-même une référence de tableau

Nous avons maintenant une méthode pour représenter la table du début de ce chapitre :

```
$table = [ [ "john", 47, "marron",   84],
           [ "mary", 23, "noisette", 58],
           [ "bill", 35, "bleu",     71] ];
```

Les crochets fonctionnent de cette façon seulement quand l'analyseur de Perl attend un terme dans une expression. Ils ne doivent pas être confondus avec les crochets dans une expression comme `$tableau[6]` — bien que l'association mnémotechnique avec les tableaux soit volontaire. À l'intérieur d'une chaîne entre guillemets, les crochets ne provoquent pas la création de tableaux anonymes ; ils deviennent simplement des caractères littéraux dans la chaîne. (Les crochets fonctionnent toujours pour l'indexation à l'intérieur de chaînes, sinon vous ne pourriez pas afficher des valeurs comme `"VAL=$tableau[6]\n"`. Et pour être tout à fait honnête, vous pouvez en fait glisser des composeurs de tableaux anonymes dans une chaîne, mais seulement s'ils sont inclus dans une expression plus grande qui est interpolée. Nous parlerons de cette fonctionnalité sympathique un peu plus loin dans ce chapitre, car elle implique du déréférencement aussi bien que du référencement.)

Le composeur de hachages anonymes

Vous pouvez créer une référence à un hachage anonyme avec des accolades :

```
$hashref = {
    'Adam'    => 'Eve',
    'Clyde'   => $bonnie,
    'Antoine' => 'Cléo' . 'patre',
};
```

Vous pouvez librement mélanger les composeurs d'autres tableaux, hachages ou sous-programmes anonymes pour les valeurs (mais pas pour les clés) afin de construire une structure aussi compliquée que vous le souhaitez.

Nous avons maintenant une nouvelle manière de représenter la table du début de ce chapitre :

```
$table = {
            "john" => [ 47, "marron",   84 ],
            "mary" => [ 23, "noisette", 58 ],
            "bill" => [ 35, "bleu",     71 ],
};
```

C'est un hachage de tableaux. Le choix de la meilleure structure est une affaire délicate, et le prochain chapitre y est entièrement dédié. Mais pour vous faire saliver, nous pouvons même utiliser un hachage de hachages pour notre table :

```
$table = {
           "john" => { age    => 47,
                       yeux   => "marron",
                       poids  => 84,
                     },
           "mary" => { age    => 23,
                       yeux   => "noisette",
                       poids  => 58,
                     },
```

```
            "bill" => { age    => 35,
                        yeux   => "bleu",
                        poids  => 71,
                      },
};
```

Comme avec les crochets, les accolades fonctionnent de cette façon uniquement quand Perl attend un terme dans une expression. Il ne faut pas les confondre avec les accolades dans une expression comme `$hash{cle}` — bien que l'association mnémotechnique soit (encore) volontaire. Les mêmes avertissements s'appliquent quant à l'utilisation des accolades à l'intérieur d'une chaîne.

Voici un avertissement supplémentaire qui ne s'appliquait pas aux crochets. Comme les accolades sont aussi utilisées pour plusieurs autres choses (y compris les blocs), vous pourrez occasionnellement avoir besoin de distinguer les accolades au début d'une instruction en mettant un `+` ou un `return` devant, afin que Perl réalise que ce n'est pas un bloc qui commence. Par exemple, si vous voulez une fonction qui crée un nouveau hachage et renvoie une référence sur lui, vous avez les possibilités suivantes :

```
sub hacheles {        { @_ } }   # Silencieusement FAUX -- renvoie @_.
sub hacheles {       +{ @_ } }   # Ok.
sub hacheles { return { @_ } }   # Ok.
```

Le composeur de sous-programmes anonymes

Vous pouvez créer une référence à un sous-programme anonyme en utilisant `sub` sans nom de sous-programme :

```
$refcode = sub { print "Boink!\n" };  # Maintenant &$refcode affiche
"Boink!"
```

Remarquez la présence du point-virgule, qui est obligatoire pour terminer l'expression. (Il ne le serait pas après la déclaration usuelle `sub`*`NOM`*`{}` qui déclare et définit un sous-programme nommé.) Hormis le fait que le code à l'intérieur n'est pas exécuté immédiatement, un `sub {}` sans nom est plus un opérateur qu'une déclaration — comme `do {}` ou `eval {}`. Il se contente de générer une référence au code, qui dans notre exemple est stockée dans `$refcode`. Cependant, quel que soit le nombre de fois que vous exécutez la ligne ci-dessus, `$refcode` se référera toujours au même sous-programme anonyme.[2]

Constructeurs d'objets

Les sous-programmes peuvent aussi renvoyer des références. Cela peut sembler banal, mais vous êtes parfois *censé* vous servir d'un sous-programme pour renvoyer une référence, plutôt que de créer la référence vous-même. En particulier, des sous-programmes spéciaux appelés *constructeurs* créent et renvoient des références sur des objets. Un objet est juste un type spécial de référence qui sait à quelle classe il est associé ; les constructeurs savent créer ce genre d'association. Ils les créent en prenant un référent ordinaire

2. Mais bien qu'il n'y ait qu'un seul sous-programme anonyme, il peut y avoir plusieurs copies des variables lexicales utilisées par le sous-programme, selon le moment où la référence a été générée. Cela sera abordé plus loin dans la section *Fermetures*.

et en le transformant en objet grâce à l'opérateur `bless` (qui signifie « bénir » en anglais), c'est pourquoi nous pouvons parler d'objets comme de références consacrées. Il n'y a rien de religieux dans tout cela ; étant donné qu'une classe fonctionne comme un type défini par l'utilisateur, consacrer un référent en fait un type utilisateur en plus d'un type prédéfini. Les constructeurs sont souvent nommés `new` — en particulier par les programmeurs C++ — mais en Perl on peut les appeler comme on veut.

Les constructeurs peuvent être appelés de n'importe laquelle de ces manières :

```
$refobj = Chien::->new(Queue => 'courte', Oreilles => 'longues');  #1
$refobj = new Chien:: Queue => 'courte', Oreilles => 'longues';    #2
$refobj = Chien->new(Queue => 'courte', Oreilles => 'longues');    #3
$refobj = new Chien Queue => 'courte', Oreilles => 'longues';      #4
```

La première et la deuxième invocations sont identiques. Toutes deux appelent une fonction nommée `new` qui est fournie par le module `Chien`. Les troisième et quatrième invocations sont identiques aux deux premières mais sont légèrement plus ambiguës : l'analyseur va se tromper si vous avez défini votre propre fonction `new` et que vous n'avez fait ni de `require`, ni de `use`, sur le module `Chien`, chacun des deux ayant pour effet de déclarer le module. Déclarez toujours vos modules si vous voulez utiliser la méthode numéro 4. (Et faites attention aux sous-programmes `Chien` errants.)

Voir le chapitre 12, *Objets*, pour une discussion sur les objets Perl.

Références de handles

Les références à un handle de fichier ou de répertoire peuvent être créées en prenant une référence à un typeglob de même nom :

```
bafouille(\*STDOUT);

sub bafouille {
    my $fh = shift;
    print $fh "heu hum hé bien a hmmm\n";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

Si vous faites circuler des handles de fichier, vous pouvez aussi utiliser le typeglob seul pour le faire : dans l'exemple ci-dessus, vous auriez pu utiliser `*STDOUT` ou `*STDIN` au lieu de `\*STDOUT` et `\*STDIN`.

Bien que vous puissiez en général indifféremment utiliser les typeglobs et les références de typeglob, il y a quelques cas où vous ne pouvez pas. Les typeglobs simples ne peuvent pas être consacrés (avec `bless`) en objets et les références de typeglobs ne peuvent être passées en dehors de la portée d'un typeglob localisé.

Pour générer de nouveaux handles de fichiers, du code ancien ferait plutôt des choses comme ceci pour ouvrir une liste de fichiers :

```
for $fichier (@noms) {
    local *FH;
    open(*FH, $fichier) || next;
    $handle{$fichier} = *FH;
}
```

Ça marche encore, mais de nos jours il est aussi facile de laisser une variable non définie s'autovivifier en typeglob anonyme :

```
for $fichier (@names) {
    my $fh;
    open($fh, $fichier) || next;
    $handle{$fichier} = $fh;
}
```

Avec les handles de fichiers indirects, que vous utilisiez des typeglobs, des références de typeglob ou l'un des objets d'E/S plus exotiques n'a aucune importance. Vous utilisez simplement un scalaire qui — d'une manière ou d'une autre — sera interprété comme un handle de fichier. Dans la plupart des cas, vous pouvez utiliser un typeglob ou une référence à un typeglob presque aléatoirement. Comme nous l'avons reconnu plus tôt, ce qui se passe ici, c'est la magie du déréférencement implicite.

Références de table de symboles

Dans des circonstances inhabituelles, vous pourriez ne pas savoir de quel type de référence vous aurez besoin au moment où vous écrivez votre programme. Une référence peut être créée en utilisant une syntaxe spéciale connue affectueusement sous le nom de syntaxe `*foo{`*`THING`*`}` (*en anglais dans le texte*). `*toto{`*`CHOSE`*`}` *(en français dans le texte)* retourne une référence à la case *`CHOSE`* de `*toto`, qui est l'entrée de la table de symbole contenant les valeurs de `$toto`, `@toto`, `%toto` et tous leurs amis.

```
$refscalaire = *toto{SCALAR};  # Même chose que \$toto
$reftableau  = *ARGV{ARRAY};   # Même chose que \@ARGV
$refhash     = *ENV{HASH};     # Même chose que \%ENV
$refcode     = *handler{CODE}; # Même chose que \&handler
$refglob     = *toto{GLOB};    # Même chose que \*toto
$refio       = *STDIN{IO};     # Euh...
```

Les noms parlent d'eux-mêmes, sauf dans le cas de `*STDIN{IO}`. Il contient l'objet interne `IO::Handle` que le typeglob contient, c'est-à-dire la partie du typeglob qui intéresse véritablement les différentes fonctions d'entrée/sortie. Pour assurer la compatibilité avec les anciennes versions de Perl, `*toto{FILEHANDLE}` est un synonyme de la notation `*toto{IO}` à la mode.

En théorie, vous pouvez utiliser un *`*HANDLE`*`{IO}` partout où vous auriez utilisé un *`*HANDLE`* ou un *`\*HANDLE`*, comme par exemple pour passer des handles à des fonctions ou pour en retourner, ou pour les stocker dans des structures plus grandes. (En pratique il reste encore quelques obstacles à aplanir.) Leur avantage est qu'ils accèdent uniquement au véritable objet d'E/S qui vous intéresse et pas à tout le typeglob, donc vous ne courez pas le risque de défaire plus de choses que vous ne voulez par une affectation de typeglob (mais si vous voulez assigner à une variable scalaire au lieu d'un typeglob, cela ne posera pas de problème). Un des inconvénients est qu'il n'est pas possible d'en autovivi-

fier à l'heure actuelle.[3]

```
bafouille(*STDOUT);
bafouille(*STDOUT{IO});

sub bafouille {
    my $fh = shift;
    print $fh "heu hum hé bien a hmmm\n";
}
```

Les deux invocations de `bafouille()` affichent « `heu hum hé bien a hmmm` ».

`*toto{`*`CHOSE`*`}` renvoie `undef` si cette *`CHOSE`* en particulier n'a pas encore été vue par le compilateur, sauf si *`CHOSE`* vaut *`SCALAR`*. Il se trouve que `*toto{SCALAR}` renvoie une référence à un scalaire anonyme même si `$toto` n'a pas encore été vu. (Perl ajoute toujours le scalaire au typeglob pour optimiser un bout de code ailleurs. Mais ne supposez pas que cela restera ainsi dans les prochaines versions.)

Création implicite de références

Une dernière méthode de création de références n'en est pas vraiment une. Les références d'un certain type se mettent simplement à exister quand vous les déréférencez dans un contexte de lvalue qui suppose qu'elle existent. Cela est extrêmement utile, et c'est aussi Ce à Quoi On s'Attend. Ce sujet est traité plus loin dans ce chapitre, quand nous verrons comment déréférencer toutes les références que nous avons créées jusqu'ici.

Utilisation des références en dur

Tout comme il existe de nombreuses manières de créer des références, il existe également plusieurs manières d'utiliser, ou de *déréférencer* une référence. Il existe un seul principe supérieur : Perl ne fait pas de référencement ou de déréférencement implicite.[4] Quand un scalaire contient une référence, il se comporte comme un simple scalaire. Il ne devient pas magiquement un tableau ou un hachage ou un sous-programme ; vous devez lui signifier explicitement de le faire, en le déréférençant.

Emploi d'une variable comme nom de variable

Quand vous rencontrez un scalaire comme `$toto`, vous devriez penser « la valeur scalaire associée à `toto` ». C'est-à-dire qu'il y a une entrée `toto` dans la table des symboles et que le drôle de caractère `$` est une manière de regarder quelle valeur scalaire se trouve à l'intérieur. Si ce qui se trouve à l'intérieur est une référence, vous pouvez regarder à l'intérieur de *cette* référence (déréférencer `$toto` en le faisant précéder d'un autre drôle de caractère. Ou en le regardant du point de vue opposé, vous pouvez remplacer le `toto` littéral dans `$toto` par une variable scalaire qui pointe sur le véritable référent. Cela est

3. Aujourd'hui, `open my $fh` autovivifie un typeglob au lieu d'un objet `IO::Handle` ; mais un des ces jours nous corrigerons cela, aussi ne devriez vous pas vous fier au fait qu'`open` autovivifie actuellement un typeglob.

4. Nous avons déjà reconnu que c'était un petit mensonge. Nous n'allons pas recommencer.

vrai pour tout type de variable, donc non seulement `$$toto` est la valeur scalaire de ce à quoi `$toto` se réfère, mais `@$titi` est la valeur tableau de ce à quoi `$titi` se réfère, `%$tutu` est la valeur hachage de ce à quoi `$tutu` se réfère, et ainsi de suite. Le résultat est que vous pouvez rajouter un drôle de caractère devant n'importe quelle valeur scalaire pour la déréférencer :

```
$toto         = "trois bosses";
$refscalaire  = \$toto;         # $refscalaire est maintenant une
                                # référence à $toto
$type_chameau = $$refscalaire;  # $type_chameau vaut maintenant "trois
                                # bosses"
```

Voici d'autres déréférences :

```
$titi = $$refscalaire;

push(@$reftableau, $fichier);
$$reftableau[0] = "Janvier";                # Affecte le premier élément
                                            # de @$reftableau
@$reftableau[4..6] = qw/Mai Juin Juillet/;  # Affecte plusieurs éléments
                                            # de @$reftableau

%$refhash = (CLE => "TROUSSEAU", OISEAU => "SIFFLER");  # Initialise tout
                                                        # le hash
$$refhash{CLE} = "VALEUR";                  # Change un couple clé/valeur
@$refhash{"CLE1","CLE2"} = ("VAL1","VAL2"); # Ajoute deux nouveaux couples

&$refcode(1,2,3);

print $refhandle "sortie\n";
```

Cette forme de déréférencement ne peut utiliser qu'une variable simple (sans indice). C'est-à-dire que le déréférencement se produit *avant* (ou lie plus fort que) toute recherche dans un tableau ou un hachage. Utilisons des accolades pour clarifier notre pensée : une expression comme `$$reftableau[0]` est équivalente à `${$reftableau}[0]` et signifie le premier élément du tableau référencé par `$reftableau`. Ce n'est pas du tout la même chose que `${$reftableau[0]}` qui déréférence le premier élément du tableau nommé `@reftableau` (et qui n'existe probablement pas). De même, `$$refhash{CLE}` est équivalent à `${$refhash}{CLE}` et n'a rien à voir avec `${$refhash{CLE}}` qui déréférencerait une entrée du hachage nommé `%refhash` (qui n'existe probalement pas). Vous serez malheureux tant que vous n'aurez pas compris cela.

Vous pouvez réaliser plusieurs niveaux de référencement et de déréférencement en concaténant les drôles de caractères de façon appropriée. Ce qui suit affiche « `salut` » :

```
$refrefref = \\\"salut";
print $$$$refrefref;
```

Vous pouvez voir les dollars comme opérant de droite à gauche. Mais le début de la chaîne doit toujours être un simple scalaire sans indice. Il existe cependant une technique qui met un peu plus de fantaisie, que nous avons déjà subrepticement utilisée un peu plus tôt, et que nous allons expliquer dans la section suivante.

Emploi d'un BLOC comme nom de variable

Non seulement vous pouvez déréférencer un simple nom de variable, mais vous pouvez aussi déréférencer le contenu d'un *BLOC*. Partout où vous mettriez un identificateur comme partie d'une variable ou d'un nom de sous-programme, vous pouvez remplacer l'identificateur par un *BLOC* qui renvoie une référence du bon type. En d'autres termes, tous les exemple précédents peuvent être clarifiés comme suit :

```
$titi = ${$refscalaire};
push(@{$reftableau}, $fichier);
${$reftableau}[0] = "Janvier";
@{$reftableau}[4..6] = qw/Mai Juin Juillet/;
${$refhash}{"CLE"} = "VALEUR";
@{$refhash}{"CLE1","CLE2"} = ("VAL1","VAL2");
&{$refcode}(1,2,3);
```

sans oublier :

```
$refrefref = \\\"salut";
print ${${${$refrefref}}};
```

Nous sommes d'accord qu'il est idiot de mettre des accolades dans des cas aussi simples, mais le *BLOC* peut contenir n'importe quelle expression. En particulier, il peut contenir une expression avec des indices. Dans l'exemple suivant, on suppose que `$dispatch{$index}` contient une référence à un sous-programme (parfois appelé « coderef »). L'exemple appelle le sous-programme avec trois arguments.

```
&{ $dispatch{$index} }(1, 2, 3);
```

Ici le *BLOC* est indispensable. Sans le couple d'accolades extérieures, Perl aurait traité `$dispatch` comme le coderef à la place de `$dispatch{$index}`.

Emploi de l'opérateur flèche

Dans le cas de hachages, de tableaux ou de sous-programmes, une troisième méthode de déréférencement implique l'utilisation de l'opérateur infixe `->`. Cette forme de « sucre syntaxique » facilite l'accès à un élément individuel d'un tableau ou d'un hachage, ainsi que l'appel indirect à un sous-programme.

Le type de déréférencement est déterminé par l'opérande de droite, c'est-à-dire par ce qui suit directement la flèche. Si ce qui suit la flèche est un crochet ou une accolade, l'opérande de gauche est traité respectivement comme une référence à un tableau ou un hachage, indexé par l'expression à droite. Si ce qui suit est une parenthèse ouvrante, l'opérande de gauche est traité comme un sous-programme à appeler avec les paramètres fournis entre les parenthèses à droite.

Chacun des trios qui suivent sont équivalents et correspondent aux trois notations que nous avons présentées. (Nous avons rajouté des espaces pour que les éléments qui se correspondent soient alignés.)

```
$  $reftableau  [2] = "Dorian";         #1
${ $reftableau }[2] = "Dorian";         #2
   $reftableau->[2] = "Dorian";         #3
```

```
$  $refhash  {TON} = "Fa#majeur";       #1
${ $refhash }{TON} = "Fa#majeur";       #2
   $refhash->{TON} = "Fa#majeur";       #3

&  $refcode  (Presto => 192);           #1
&{ $refcode }(Presto => 192);           #2
   $refcode->(Presto => 192);           #3
```

Vous pouvez voir que le premier drôle de caractère manque dans la troisième notation de chaque série. Le drôle de caractère est deviné par Perl, ce qui fait que cette notation ne peut servir pour déréférencer des tableaux ou des hachages entiers, ni même des tranches de ceux-ci. Cependant, tant que vous vous en tenez à des valeurs scalaires, vous pouvez utiliser n'importe quelle expression à gauche de `->`, y compris un autre déréférencement, car plusieurs opérateurs flèches s'associent de gauche à droite :

```
print $tableau[3]->{"Anglais"}->[0];
```

Vous pouvez déduire de cette expression que le quatrième élément de `@tableau` est supposé être une référence à un hachage, et que la valeur associée à l'entrée « `Anglais` » de ce hachage est supposée être une référence à un tableau.

Remarquez que `$tableau[3]` et `$tableau->[3]` ne sont pas les mêmes. Le premier donne le quatrième élément de `@tableau`, tandis que le second donne le quatrième élément du tableau (peut-être anonyme) dont la référence se trouve dans `$tableau`.

Supposez maintenant que `$tableau[3]` soit indéfini. L'instruction suivante est toujours légale :

```
$tableau[3]->{"Anglais"}->[0] = "January";
```

C'est l'un des cas que nous avons mentionné précédemment où les références se mettent à exister (ou sont « autovivifiées ») quand on les utilise comme des lvalues (c'est-à-dire quand une valeur leur est affectée). Si `$tableau[3]` était indéfini, il est automatiquement défini comme une référence à un hachage afin que nous puissions affecter une valeur à `$tableau[3]->{"Anglais"}` à l'intérieur. Une fois que c'est fait, `$tableau[3]->{"Anglais"}` est automatiquement défini comme une référence à un tableau, afin que nous puissions affecter une valeur au premier élément de ce tableau. Remarquez que les *rvalues* (les valeurs à droite d'une affectation) sont légèrement différentes : `print $tableau[3]->{"Anglais"}->[0]` ne crée que `$tableau[3]` et `$tableau[3]->{"Anglais"}`, et pas `$tableau[3]->{"Anglais"}->[0]`, puisque le dernier élément n'est pas une lvalue. (Le fait que cela définisse les deux premiers dans un contexte de rvalue devrait être considéré comme une bogue. Nous corrigerons peut-être cela un jour.)

La flèche est optionnelle entre les indices délimités par des accolades ou des crochets, et entre une accolade ou un crochet fermant et une parenthèse destinée à un appel de fonction indirect. Vous pouvez donc réduire le code précédent à :

```
$dispatch{$index}(1, 2, 3);
$tableau[3]{"Anglais"}[0] = "January";
```

Dans le cas des tableaux ordinaires, cela vous permet d'avoir des tableaux multidimensionnels exactement comme ceux de C :

```
$reponse[$x][$y][$z] += 42;
```

Bon d'accord, pas *exactement* comme ceux de C. D'abord C ne sait pas comment faire grossir ses tableaux à la demande, comme Perl. Ensuite certaines constructions similai-

res dans les deux langages sont analysées différemment. En Perl, les deux instructions suivantes font la même chose :

```
$refliste->[2][2] = "coucou";    # Assez clair
$$refliste[2][2]  = "coucou";    # Prête à confusion
```

La seconde instruction peut choquer le programmeur C, qui est habitué à utiliser `*a[i]` pour signifier « ce qui est pointé par le *i*e élément de `a` ». Mais en Perl, les cinq caractères (`$ @ * % &`) lient plus étroitement que les accolades ou les crochets.[5] C'est donc `$$refliste` et non `$refliste[2]` qui est pris comme référence à un tableau. Si vous voulez le comportement de C, soit vous devez écrire `${$refliste[2]}` pour forcer l'évaluation de `$refliste[2]` avant le premier `$`déréférenceur, ou bien utiliser la notation `->` :

```
$refliste[2]->[$salutation] = "coucou";
```

Emploi des méthodes d'objets

Si une référence se trouve être une référence à un objet, alors la classe qui définit cet objet fournit probablement des méthodes pour accéder à ce qu'il contient, et vous devriez généralement vous en tenir à ces méthodes si vous ne faites qu'utiliser la classe (par opposition à ceux qui l'implémentent). En d'autres termes, soyez correct et ne traitez pas un objet comme une simple référence, bien que Perl vous laisse faire si vous le devez vraiment. Perl n'impose pas l'encapsulation. Nous ne sommes pas des dictateurs. Mais nous espérons cependant un peu de civilité de votre part.

En retour de cette civilité, vous gagnez un complète orthogonalité entre les objets et les structures de données. Toute structure de données peut se comporter comme un objet, si vous le voulez. Et ne pas le faire, si vous ne voulez pas.

Pseudo-hachages

Un *pseudo-hachage* est une référence à un tableau dont le premier élément est une référence à un hachage. Vous pouvez traiter le pseudo-hachage soit comme une référence de tableau (comme vous pouviez vous y attendre) ou comme une référence de hachage (ce à quoi vous ne vous attendiez peut-être pas). Voici un exemple de pseudo-hachage :

```
$john = [ {age => 1, yeux => 2, poids => 3}, 47, "marron", 84 ];
```

Le hachage sous-jacent dans `$john->[0]` définit les noms `"age"`, `"yeux"` et `"poids"` des éléments de tableau qui suivent (`47`, `"marron"` et `84`). Maintenant vous pouvez accéder à un élément à la fois par la notation de hachage et par celle de tableau :

```
$john->{poids}              # Traite $john comme une référence de hachage
$john->[3]                  # Traite $john comme une référence de tableau
```

La magie des pseudo-hachages n'est pas très puissante ; ils ne connaissent qu'un seul « tour » : comment transformer un déréférencement de hachage en déréférencement de tableau. Quand vous ajoutez un nouvel élément au pseudo-hachage, vous devez ex-

5. Mais pas à cause de la précédence des opérateurs. Les « drôles de caractères » de Perl ne sont pas des opérateurs dans ce sens-là. La grammaire de Perl interdit simplement à tout ce qui est plus compliqué qu'une simple variable ou un bloc de suivre le drôle de caractère initial, pour plein de drôles de raisons.

plicitement préciser au hachage de correspondance sous-jacent où l'élément va résider avant d'utiliser la notation de hachage :

```
$john->[0]{hauteur} = 4;      # hauteur sera l'élément 4
$john->{hauteur} = "grand";   # Ou $john->[4] = "grand"
```

Perl lève une exception si vous essayez d'effacer une clé de pseudo-hachage, bien que vous puissiez toujours effacer une clé du hachage de correspondance. Perl lève également une exception si vous essayez d'accéder à une clé qui n'existe pas (« existence » signifiant ici « présence dans le hachage de correspondance »).

```
delete $john->[0]{hauteur};  # Supprime seulement dans le hachage
                             # sous-jacent
$john->{hauteur};            # Ceci lève désormais une exception
$john->[4];                  # Affiche toujours "grand"
```

Ne tentez pas de `splice` du tableau sans savoir ce que vous faites. Si les éléments du tableau se déplacent, les valeurs de correspondance du hachage feront toujours référence aux *anciennes* positions des éléments, à moins que vous ne les changiez elles aussi explicitement. La magie des pseudo-hachages n'est pas très puissante.

Pour éviter les incohérences, vous pouvez utiliser la fonction `fields::phash` fournie par le pragma `use fields` pour créer un pseudo-hachage :

```
use fields;
$ph = fields::phash(age => 47, yeux => "marron", poids => 84);
print $ph->{age};
```

Il existe deux manières de vérifier l'existence d'une clé dans un pseudo-hachage. La première est d'utiliser `exists`, qui vérifie si le champ fourni a jamais été affecté. Il fonctionne de cette façon pour reproduire le comportement d'un véritable hachage. Par exemple :

```
use fields;
$ph= fields::phash([qw(age yeux poids)], [47]);
$ph->{yeux} = undef;

print exists $ph->{age};   # Vrai, 'age' a été modifié dans la déclaration.
print exists $ph->{poids}; # Faux, 'poids' n'a pas été utilisé.
print exists $ph->{yeux};  # Vrai, les 'yeux' ont été modifiés.
```

La seconde manière est d'utiliser `exists` sur le hachage de correspondance installé dans le premier élément. Ceci vérifie que la clé donnée est un champ valide pour ce pseudo-hachage :

```
print exists $ph->[0]{age}; # Vrai, 'age' est un champ valide
print exists $ph->[0]{nom}; # Faux, 'nom' ne peut pas être utilisé
```

Contrairement à ce qui se produit pour un véritable hachage, appeler `delete` sur un élément de pseudo-hachage n'efface que la valeur dans le tableau, et pas la clé dans le hachage de correspondance. Pour effacer la clé, vous devrez l'effacer explicitement du hachage de correspondance. Une fois que cela est fait, vous ne pouvez plus utiliser ce nom de clé comme indice du pseudo-hachage :

```
print delete $ph->{age};     # Détruit $ph->[1], 47 et le renvoie
print exists $ph->{age};     # Maintenant c'est faux.
```

```
print exists $ph->[0]{age};  # Vrai, 'age' est encore utilisable
print delete $ph->[0]{age};  # Maintenant 'age' n'existe plus
print $ph->{age};            # Exception à l'exécution
```

Vous commencez sûrement à vous demander ce qui a pu motiver ce carnaval de tableaux se promenant dans un habit de hachage. Les tableaux fournissent un accès plus rapide et un stockage plus efficace, tandis que les hachages offrent la possibilité de nommer (au lieu de numéroter) vos données ; les pseudo-hachages offrent le meilleur des deux mondes. Mais c'est en considérant la phase de compilation de Perl que vous verrez apparaître le meilleur bénéfice de l'opération. Avec l'aide d'un pragma ou deux, le compilateur peut vérifier l'accès correct aux champs valides, aussi pouvez-vous vérifier l'inexistence d'un champ (ou les fautes d'orthographe) avant le début de votre programme.

Les propriétés de vitesse, d'efficacité, de vérification d'accès à la compilation (vous pouvez même le voir comme de la validation de type) sont particulièrement pratiques pour créer des modules de classes robustes et efficaces. Voir la discussion du pragma `use fields` au chapitre 12 et au chapitre 31, *Modules de pragmas*.

Les pseudo-hachages sont une fonctionnalité nouvelle et relativement expérimentale ; en tant que telle, l'implémentation en interne est susceptible de changer à l'avenir. Afin de vous protéger de tels changements, utilisez toujours l'interface documentée du module `fields`, en utilisant ses fonctions `phash` et `new`.

D'autres tours avec les références en dur

Comme mentionné précédemment, l'opérateur antislash est en général utilisé sur un seul référent pour générer une référence unique, mais ce n'est pas obligatoire. Quand il est utilisé sur une liste de référents, il produit une liste de références correspondantes. La seconde ligne de l'exemple suivant produit exactement le même resultat que la première, car l'antislash est automatiquement distribué sur toute la liste.

```
@listeref = (\$s, \@a, \%h, \&f);    # Liste de quatre références
@listeref = \($s,  @a   %h,  &f);    # Même chose
```

Si une expression entre parenthèses contient exactement un tableau ou un hachage, toutes ses valeurs sont interpolées et des références sont renvoyées pour chacune :

```
@listeref = \(@x);                   # Interpole le tableau puis prend
                                     # les références
@listeref = map { \$_ } @x;          # Même chose
```

Cela se produit également quand il y a des parenthèses à l'intérieur :

```
@listeref = \(@x, (@y));             # Seuls les aggrégats simples sont
                                     # interpolés
@listeref = (\@x, map { \$_ } @y);   # Même chose
```

Si vous essayez cela avec un hachage, le résultat contiendra des références aux valeurs (comme vous vous y attendez), mais aussi des références à des *copies* des clés (comme vous en vous y attendez peut-être pas).

Comme les tranches de tableaux et de hachages sont seulement des listes, vous pouvez antislasher des tranches de l'un ou l'autre type pour obtenir une liste de références. Chacune des trois listes suivantes fait exactement la même chose :

```
@envrefs = \@ENV{'HOME', 'TERM'};         # Antislashe une tranche
@envrefs = \( $ENV{HOME},  $ENV{TERM} );  # Antislashe une liste
@envrefs = ( \$ENV{HOME}, \$ENV{TERM} );  # Une liste de deux références
```

Comme les fonctions peuvent retourner des listes, vous pouvez leur appliquer l'opérateur antislash. Si vous avez plusieurs appels de fonction, interpolez d'abord la valeur de retour de chaque fonction dans une liste plus grande avant d'antislasher l'ensemble :

```
@listeref = \fx();
@listeref = map { \$_ } fx();                # Même chose

@listeref = \( fx(), fy(), fz() );
@listeref = ( \fx(), \fy(), \fz() );         # Même chose
@listeref = map { \$_ } fx(), fy(), fz();    # Même chose
```

Comme l'opérateur antislash fourni un contexte de liste à ses opérandes, ces fonctions sont toutes appelées en contexte de liste. Si l'antislash est lui-même en contexte scalaire, vous finirez avec une référence à la dernière valeur de la liste renvoyée par la fonction :

```
@listeref  = \localtime();     # Ref à chacun des neuf éléments de la date
$reffinale = \localtime();     # Ref à l'indicateur de l'heure d'été
```

À cet égard, l'opérateur antislash se comporte comme les opérateurs de liste nommés, comme `print`, `reverse` et `sort`, qui fournissent toujours un contexte de liste à leur droite, quoiqu'il se passe à leur gauche. Comme avec les opérateurs de liste nommés, utilisez un `scalar` explicite pour forcer ce qui suit à être en contexte de liste :

```
$refdate = \scalar localtime();    # \"Thu Jul 26 11:37:38 2001"
```

Vous pouvez vous servir de l'opérateur `ref` pour déterminer vers quoi une référence pointe. Vous pouvez voir l'opérateur `ref` comme un opérateur « typeof » qui renvoie vrai si son argument est une référence et faux dans le cas contraire. La valeur renvoyée dépend du type de chose référencée. Les types prédéfinis comprennent `SCALAR`, `ARRAY`, `HASH`, `CODE`, `GLOB`, `REF`, `LVALUE`, `IO`, `IO::Handle` et `Regexp`. Ici, nous nous en servons pour valider les arguments d'un sous-programme :

```
sub somme {
    my $reftableau = shift;
    warn "Ce n'est pas une référence de tableau"
         if ref($reftableau) ne "ARRAY";
    return eval join("+", @$reftableau);
}
```

Si vous utilisez une référence en dur dans un contexte de chaîne, elle sera convertie en une chaîne contenant le type et l'adresse : `SCALAR(0x1fc0e)`. (La conversion inverse ne peut pas être faite, puisque l'information de décompte de références est perdue pendant l'opération de conversion en chaîne — et aussi car il serait dangereux de laisser des programmes accéder à une adresse mémoire donnée par une chaîne arbitraire.)

Vous pouvez utiliser l'opérateur `bless` pour associer un référent à un paquetage fonctionnant comme une classe d'objet. Quand vous faites cela, `ref` renvoie le nom de la classe au lieu du type interne. Une référence d'objet utilisée en contexte de chaîne renvoie une chaîne avec les types interne et externe, ainsi que l'adresse en mémoire : `MonType=HASH(0x20d10)` ou `IO::Handle=IO(0x186904)`. Pour plus de détails sur les objets, voir le chapitre 12.

Puisque le type de référent que vous désirez est indiqué par la façon dont vous le déréférencez, un typeglob peut être utilisé de la même façon qu'une référence, malgré le fait qu'un référent contienne plusieurs référents de types divers. `${*main::toto}` et `${\$main::toto}` accèdent donc tous deux à la même variable scalaire, même si le second est plus efficace.

Voici une astuce pour interpoler la valeur de retour d'un appel de fonction dans une chaîne :

```
print "Ma fonction a renvoyé @{[ mafonction(1,2,3) ]} cette fois.\n";
```

Voici comment cela fonctionne. À la compilation, quand le `@{...}` est vu à l'intérieur de la chaîne entre guillemets doubles, il est analysé comme un bloc qui renvoie une référence. À l'intérieur du bloc, les crochets créent une référence à un tableau anonyme à partir de ce qui se trouve entre eux. À l'exécution, `mafonction(1,2,3)` est donc appelée en contexte de liste, et les résultats sont chargés dans un tableau anonyme dont une référence est renvoyée à l'intérieur du bloc. Cette référence de tableau est immédiatement déréférencée par le `@{...}` qui l'entoure, et la valeur de tableau est interpolée dans la chaîne à doubles guillemets tout comme un tableau ordinaire le serait. Ce stratagème s'avère également utile pour des expressions quelconques, comme :

```
print "Il nous faut @{ [$n + 5] } machins !\n";
```

Cependant faites attention : les crochets fournissent un contexte de liste à leur expression. Dans ce cas, cela n'a aucune importance, mais l'appel précédent à `mafonction` aurait pu y être sensible. Quand il le faut, utilisez un `scalar` explicite pour forcer le contexte :

```
print "mafonction renvoie @{ [scalar mafonction(1,2,3)] } maintenant.\n";
```

Fermetures

Nous avons parlé plus haut de la création de sous-programmes anonymes avec un `sub {}` sans nom. Vous pouvez voir ces sous-programmes comme s'ils étaient définis à l'exécution, ce qui signifie qu'ils ont un instant de génération en plus d'une position de définition. Certaines variables pourraient être à portée quand le sous-programme est créé, et d'autres variables pourraient être à portée quand le sous-programme est appelé.

Oubliez les sous-programmes pour le moment et considérez une référence qui se réfère à une variable lexicale :

```
{
    my $bestiau = "chameau";
    $refbestiau = \$creature;
}
```

La valeur de `$$refbestiau` restera « `chameau` » bien que `$bestiau` disparaisse après l'accolade fermante. Mais `$refbestiau` aurait très bien pu faire référence à un sous-programme qui se réfère à `$bestiau` :

```
{
    my $bestiau = "chameau";
    $refbestiau = sub { return $bestiau };
}
```

C'est une *fermeture*, qui est une notion sortie du monde de la programmation fonctionnelle de LISP et Scheme.[6] Cela signifie que quand vous définissez un sous-programme anonyme dans une portée lexicale particulière à un moment donné, il prétend continuer à fonctionner dans cette portée particulière, même si vous l'appelez plus tard depuis l'extérieur de cette portée. (Un puriste dirait qu'il n'a pas besoin de prétendre — il *fonctionne* réellement dans cette portée.)

En d'autres termes, vous êtes assuré de garder la même copie d'une variable lexicale à chaque fois, même si d'autres instances de cette variable lexicale ont été créées avant ou après pour d'autres instances de cette fermeture. Cela vous permet de d'ajuster les valeurs utilisées par un sous-programme quand vous le définissez, et pas seulement quand vous l'appelez.

Vous pouvez aussi voir les fermetures comme une manière d'écrire un patron de sous-programme sans utiliser `eval`. Les variables lexicales servent de paramètres pour remplir le modèle, ce qui est utile pour configurer des petits bouts de code à exécuter plus tard. On les appelle communément des *fonctions de rappel* en programmation événementielle, quand vous associez un bout de code à la pression d'une touche, à un clic de souris, à l'exposition d'une fenêtre et ainsi de suite. Quand on les utilise comme des fonctions de rappel, les fermetures font exactement ce à quoi vous vous attendez, même si vous ne connaissez rien à la programmation fonctionnelle. (Remarquez que cette histoire de fermeture ne s'applique qu'aux variables `my`. Les variables globales fonctionnent comme elles ont toujours fonctionné, étant créées et détruites de façon complètement différente des variables lexicales.

Une autre utilisation des fermetures se fait à l'intérieur de *générateurs de fonctions* ; c'est-à-dire de fonctions qui créent et retournent de nouvelles fonctions. Voici un exemple de générateur de fonction implémenté avec des fermetures :

```
sub faitbonjour  {
    my $salut = shift;
    my $newfunc = sub {
        my $cible = shift;
        print "$salut, $cible !\n";
    };
    return $newfunc;              # Renvoie une fermeture
}

$f = faitbonjour("Bonjour");      # Crée une fermeture
$g = faitbonjour("Salutations");  # Crée une autre fermeture

# Le temps passe...

$f->("monde");
$g->("Terriens");
```

Cela affiche :

```
Bonjour, monde !
Salutations, Terriens !
```

6. Dans ce contexte, « fonctionnel » ne doit pas être interprété comme le contraire de « dysfonctionnel ».

Remarquez en particulier comme `$salut` continue de faire référence à la valeur effectivement passée à `faitbonjour`, malgré le fait que le `my $salut` soit hors de portée quand le sous-programme anonyme tourne. C'est tout l'intérêt des fermetures. Comme `$f` et `$g` contiennent des références à des fonctions qui, quand on les appelle, ont toujours besoin d'accéder à des versions distinctes de `$salut`, ces versions restent automatiquement dans les environs. Si maintenant vous effacez `$f`, *sa* version de `$salut` disparaitra automatiquement. (Perl nettoie quand vous avez le dos tourné.)

Perl ne fournit pas de références aux méthodes d'objets (décrites au chapitre 12) mais vous pouvez obtenir un effet similaire en utilisant une fermeture. Supposons que vous voulez une référence, pas seulement à la fonction que la méthode représente, mais aussi une référence qui, quand vous l'invoquez, appelle cette méthode sur un objet particulier. Vous pouvez commodément vous souvenir à la fois de l'objet et de la méthode en en faisant des variables lexicales liées à une fermeture :

```
sub get_ref_methode {
    my ($self, $nommethode) = @_;
    my $refmeth = sub {
        # le @_ ci-dessous est différent de celui au-dessus !
        return $self->$nommethode(@_);
    };
    return $refmeth;
}

my $toutou = new Chien::
            Nom    => "Lucky",
            Pattes => 3,
            Queue  => "coupée";

our $remueur = get_ref_methode($toutou, 'remuer');
$remueur->("queue");        # Appelle $dog->remuer('queue').
```

Vous pouvez donc demander à Lucky de remuer ce qui lui reste de queue même quand la variable lexicale `$toutou` est hors de portée et que Lucky n'est plus visible. La variable globale `$remueur` peut toujours lui faire remuer la queue, où qu'il se trouve.

Fermetures comme modèles de fonctions

Utiliser une fermeture comme modèle de fonctions vous permet de générer plusieurs fonctions qui agissent de façon similaire. Supposons que vous voulez une suite de fonctions qui génèrent des changements de fonte HTML pour diverses couleurs :

```
print "Soyez ", rouge("prudent"), "avec cette ", vert("lumière"), " !!!";
```

Les fonctions `rouge` et `vert` seraient très similaires. Nous aimerions bien nommer nos fonctions, mais les fermetures n'ont pas de nom puisqu'il s'agit juste de routines qui s'y croient. Pour contourner cela, nous allons utiliser une jolie astuce pour nommer nos sous-programmes anonymes. Vous pouvez lier une référence de code à un nom existant en l'affectant au typeglob du nom de la fonction que vous voulez. (Voir la section *Tables de symboles* au chapitre 10, *Paquetages*). Dans ce cas, nous allons la lier à deux noms différents, l'un en majuscules et l'autres en minuscules :

```
@couleurs = qw(red blue green yellow orange purple violet);
```

```
for my $nom (@couleurs) {
    no strict 'refs';       # Autorise les références symboliques
    *$nom = *{uc $nom} = sub { "<FONT COLOR='$nom'>;@_</FONT>" };
}
```

Vous pouvez maintenant appeler les fonctions nommées `red`, `RED`, `blue`, `BLUE` et ainsi de suite, et la fermeture appropriée sera invoquée. Cette technique réduit la durée de compilation et conserve la mémoire ; elle est aussi moins susceptible d'erreur, puisque la vérification de syntaxe se fait à la compilation. Il est impératif que les variables dans le sous-programme anonyme soient lexicales afin de créer une fermeture. C'est la raison du `my` dans l'exemple ci-dessus.

Voici l'un des rares cas où donner un prototype à une fermeture sert à quelque chose. Si vous aviez voulu imposer un contexte scalaire aux arguments de ces fonctions (ce n'est probablement pas une bonne idée pour cet exemple), vous auriez plutôt écrit ceci :

```
*$nom = sub ($) { "<FONT COLOR='$nom'>$_[0]</FONT>" };
```

C'est presque suffisant. Cependant comme la vérification de prototypes se produit à la compilation, l'affectation à l'exécution se produit trop tard pour servir à quelque chose. Vous pouvez corriger cela en mettant toute la boucle d'affectations dans un bloc `BEGIN` qui la forcera à se produire à la compilation. (Plus probalement, vous la mettriez dans un module sur lequel vous ferez un `use` à la compilation.) Les prototypes seront alors visibles pour le reste de la compilation.

Sous-programmes emboîtés

Si vous avez l'habitude (prise dans d'autres langages de programmation) d'utiliser des procédures emboîtées dans d'autres procédures, chacune avec ses propres variables privées, il va vous falloir travailler un peu en Perl. Les sous-programmes nommés ne s'emboîtent pas bien, alors que les sous-programmes anonymes le peuvent.[7] De toute façon, nous pouvons émuler des sous-programmes emboîtés à portée lexicale à l'aide de fermetures. En voici un exemple :

```
sub exterieur {
    my $x = $_[0] + 35;
    local *interieur = sub { return $x * 19 };
    return $x + interieur();
}
```

Maintenant `interieur` ne peut être appelé que depuis `exterieur` à cause de l'affectation temporaire de la fermeture. Mais quand il l'est, il a accès normalement à la variable lexicale `$x` depuis la portée de `exterieur`.

Ceci a l'intéressant effet de créer une fonction locale à une autre, ce qui est quelque chose qui n'est normalement pas supporté en Perl. Comme `local` est à portée dynamique, et parce que les noms de fonctions sont globaux à leur paquetage, toute autre fonction qui serait appelée par `exterieur` pourrait aussi appeler la version temporaire

7. Pour être plus précis, les sous-programmes ayant un nom global ne s'emboîtent pas. Malheureusement c'est le seul type de déclaration de sous-programmes nommés que nous ayons. Nous n'avons pas encore implémenté les sous-programmes nommés à portée lexicale (connus sous le nom de `my sub`), mais quand nous le ferons, ils s'emboîteront correctement.

d'interieur. Pour empêcher cela, il vous faudra un niveau supplémentaire d'indirection :

```
sub exterieur {
    my $x = $_[0] + 35;
    my $interieur = sub { return $x * 19 };
    return $x + $interieur->();
}
```

Références symboliques

Que se passe-t-il si vous essayez de déréférencer une valeur qui n'est pas une référence en dur ? La valeur est alors traitée comme une *référence symbolique*. C'est-à-dire que la référence est interprétée comme une chaîne représentant le *nom* d'une variable globale.

Voici comment cela fonctionne :

```
$nom = "bam";
$$nom = 1;                # Définit $bam
$nom->[0] = 4;            # Définit le premier élément de @bam
$nom->{X} = "Y";          # Définit l'élément X de %bam à Y
@$nom = ();               # Vide @bam
keys %$nom;               # Renvoie les clés de %bam
&$nom;                    # Appelle &bam
```

C'est très puissant et légèrement dangereux, en ce qu'il est possible de vouloir (avec la plus grande sincérité) utiliser une référence en dur, mais d'accidentellement utiliser une référence symbolique à la place. Pour vous protéger contre cela, vous pouvez écrire :

```
use strict 'refs';
```

et seules les références en dur seront autorisées dans le reste du bloc englobant. Un bloc intérieur peut suspendre cette restriction par :

```
no strict 'refs';
```

Il est aussi important de comprendre la différence entre les deux lignes de code suivantes :

```
${identificateur};    # Comme $identificateur.
${"identificateur"};  # $identificateur aussi, mais une référence
symbolique.
```

Parce que la seconde forme est entre guillemets, elle est traitée comme une référence symbolique, et générera une erreur avec `use strict 'refs'`. Même si `use strict 'refs'` n'est pas activé, elle peut seulement faire référence à une variable de paquetage. Mais la première forme est identique à la forme sans accolades, et fera même référence à une variable à portée lexicale s'il y en a de déclarée. L'exemple suivant le montre (et la section suivante en parle).

Seules les variables de paquetage sont accessibles par des références symboliques, car les références symboliques passent toujours par la table de symboles du paquetage. Comme les variables lexicales ne se trouvent pas la table de symboles, elles sont invisibles pour ce mécanisme. Par exemple :

```
our $valeur  = "globale";
```

```
{
    my $valeur = "privée";
    print "À l'intérieur, my est ${valeur}, ";
    print "mais our est ${'valeur'}.\n";
}
print "À l'extérieur, ${valeur} est de nouveau ${'valeur'}.\n";
```

ce qui affiche :

```
À l'intérieur, my est privée, mais our est globale.
À l'extérieur, globale est de nouveau globale.
```

Accolades, crochets et guillemets

Dans la section précédente nous avons vu que `${identificateur}` n'est pas traité comme une référence symbolique. Vous vous demandez sûrement comment cela peut interagir avec les mots réservés. La réponse est que cela n'interagit pas. Bien que `push` soit un mot réservé, ces deux instructions affichent « `pop par dessus` » :

```
$push = "pop par ";
print "${push}dessus";
```

La raison en est qu'historiquement, cette utilisation des accolades est celle des shells Unix pour isoler un nom de variable du texte alphanumérique qui le suit, et qui aurait sinon été interprété comme faisant partie du nom. C'est la façon dont beaucoup de gens s'attendent à voir fonctionner l'interpolation, aussi l'avons nous fait fonctionner de la même façon en Perl. Mais en Perl cette notion s'étend plus loin, et s'applique également aux accolades utilisées pour générer les références, qu'elles soient ou non entre guillemets. Cela signifie que :

```
print ${push} . 'dessus';
```

ou même (puisque les blancs ne changent rien) :

```
print ${ push } . 'dessus';
```

affichent tous deux « `pop par dessus` », même si les accolades sont en dehors des guillemets doubles. La même règle s'applique à tout identificateur utilisé pour indexer un hachage. Ainsi, au lieu d'écrire :

```
$hash{ "aaa" }{ "bbb" }{ "ccc" }
```

vous pouvez simplement écrire :

```
$hash{ aaa }{ bbb }{ ccc }
```

ou :

```
$hash{aaa}{bbb}{ccc}
```

sans vous soucier de savoir si les indices sont des mots réservés. Donc ceci :

```
$hash{ shift }
```

est interprété comme `$hash{"shift"}`. Vous pouvez forcer l'interprétation comme un mot réservé en ajoutant quelque chose qui le différencie d'un identificateur :

```
$hash{  shift()  }
$hash{ +shift    }
$hash{  shift @_ }
```

Les références ne fonctionnent pas comme clés de hachages

Les clés de hachage sont stockées en interne comme des chaînes.[8] Si vous essayez de stocker une référence dans un hachage, la valeur de la clé sera convertie en chaîne :

```
$x{ \$a } = $a;
($cle, $valeur) = each %x;
print $$cle;                    # FAUX
```

Nous avons mentionné plus tôt que vous ne pouvez pas reconvertir une chaîne en référence en dur. Si vous essayez de déréférencer `$cle` qui contient une simple chaîne, vous ne récupèrerez pas une référence en dur, mais simplement un déréférencement symbolique. Et comme vous n'avez pas de variable nommée `SCALAR(0x1fc0e)`, vous n'obtiendrez pas ce que vous voulez. Vous devriez plutôt essayer quelque chose comme ceci :

```
$r = \@a;
$x{ $r } = $r;
```

Comme cela vous pourrez au moins utiliser la *valeur* du hachage, qui sera une référence en dur, au lieu de la clé qui n'en est pas une.

Bien que vous ne puissiez pas stocker une référence en tant que clé, si vous utilisez une référence en dur dans un contexte de chaîne (comme dans l'exemple précédent), vous êtes *certain* que cela produira une chaîne unique, puisque l'adresse de la référence fait partie de la chaîne retournée. Vous pouvez donc utiliser une référence comme clé d'un hachage. Seulement vous ne pourrez pas la déréférencer par la suite.

Il existe un cas particulier de hachage dans lequel vous *pouvez* utiliser des références comme clés. Par la magie[9] du module `Tie::RefHash` fourni avec Perl, vous pouvez faire ce que nous venons juste de dire que vous ne pouviez pas faire :

```
use Tie::RefHash;
tie my %h, 'Tie::RefHash';
%h = (
    ["ceci", "ici"]    => "à la maison",
    ["cela", "là-bas"] => "ailleurs",
);
while ( my($refcle, $valeur) = each %h ) {
    print "@$refcle is $valeur\n";
}
```

En fait, en liant des implémentations différentes aux types prédéfinis, vous pouvez faire se comporter des scalaires, des hachages et des tableaux de beaucoup de manières dont nous vous avions dit que c'était impossible. Ça nous apprendra, imbéciles d'auteurs...

Pour en savoir plus sur les liens avec `tie`, voir le chapitre 14, *Variables liées*.

---

8. Elles sont également stockées en *externe* comme des chaînes, par exemple quand vous les stockez dans des fichiers DBM. En fait, les fichiers DBM *imposent* que leurs clés (et valeurs) soient des chaînes.

9. Oui, c'est un terme *technique*, comme vous vous en rendrez compte si vous fouillez un peu dans le fichier *mg.c* dans les sources de Perl.

Ramasse-miettes, références circulaires et références faibles

Les langages de haut niveau permettent aux programmeurs de ne pas s'inquiéter de la libération de la mémoire quand ils ont fini de l'utiliser. Ce procédé de récupération automatique de la mémoire s'appelle en français le *ramassage des miettes* (en anglais *garbage collecting* ou collecte des ordures). Dans la plupart des cas, Perl se sert d'un mécanisme simple et rapide de ramasse miettes fondé sur le comptage de références.

Quand on sort d'un bloc, ses variables à portée locale sont normalement libérées, mais il est possible de cacher vos miettes de façon à ce que le ramasse miettes de Perl ne puisse les trouver. Un problème sérieux est que de la mémoire inatteignable avec un décompte de références non nul ne sera normalement pas libérée. C'est pourquoi les références circulaires sont une mauvaise idée :

```
{               # fait pointer $a et $b l'un sur l'autre
    my ($a, $b);
    $a = \$b;
    $b = \$a;
}
```

ou plus simplement :

```
{               # fait pointer $a sur lui-même
    my $a;
    $a = \$a;
}
```

Bien que `$a` doive être désalloué à la fin du bloc, en fait il ne l'est pas. Quand vous construisez des structures de données récursives, il vous faudra casser (ou affaiblir, voir plus loin) l'autoréférence vous-même si vous voulez que la mémoire soit récupérée avant la fin de votre programme (ou de votre thread). (À la sortie, la mémoire sera automatiquement récupérée pour vous par un mécanisme coûteux mais complet de ramasse-miettes par marquage et balayage.) Si la structure de données est un objet, vous pouvez utiliser la méthode `DESTROY` pour casser automatiquement la référence ; voir *Ramasse-miettes avec les méthodes DESTROY* au chapitre 12.

Une situation similaire peut se produire avec les *caches* — des entrepôts de données conçus pour une récupération plus rapide que la normale. Hors du cache on trouve des références à données à l'intérieur du cache. Le problème se produit quand toutes ces références sont effacées, mais que les données du cache et les références internes restent. L'existence d'une référence empêche Perl de récupérer le référent, même si nous voulons que les données du cache disparaissent dès qu'elles ne sont plus utiles. Comme pour les références circulaires, nous vous une référence qui n'affecte pas le compte de références, et donc ne retarde pas le ramasse-miettes.

Les *références faibles* résolvent les problèmes causés par les références circulaires en vous permettant d'« affaiblir » n'importe quelle référence ; c'est-à-dire lui permettre de ne pas modifier le compte des références. Quand la dernière référence non faible à un objet est détruite, l'objet est détruit et toutes les références faibles sont automatiquement libérées.

Pour utiliser cette fonctionnalité, vous aurez besoin du module `WeakRef` disponible sur CPAN qui contient une documentation complémentaire. Les références faibles sont une fonctionnalité expérimentale. Mais il faut bien que quelqu'un serve de cobaye.

9

Structures de données

Perl fournit gratuitement beaucoup des structures de données que vous devez construire vous-même dans d'autres langages de programmation. Les piles et et les files que les informaticiens en herbe apprennent en cours ne sont que des tableaux en Perl. Quand vous faites `push` et `pop` (ou `shift` et `unshift`) sur un tableau, c'est une pile ; quand vous faites `push` et `shift` (ou `unshift` et `pop`) sur un tableau, c'est une file. Beaucoup de structures arborescentes dans le monde sont construites pour fournir un accès rapide et dynamique à une table de consultation conceptuellement plate. Bien sûr, les hachages font partie de Perl et ils fournissent un accès rapide et dynamique à une table de consultation conceptuellement plate. Tout cela sans les abrutissantes structures de données que trouvent magnifiques ceux dont les esprits ont déjà été convenablement abrutis.

Mais parfois vous avez besoin de structures de données imbriquées parce qu'elle modélisent naturellement le problème que vous essayez de résoudre. Perl vous permet alors de combiner et d'emboîter des tableaux et des hachages de manière à créer des structures de données aussi complexes que vous voulez. Quand ils sont utilisés correctement, ils peuvent servir à créer des listes chaînées, des arbres binaires, des tas, des arbres-B, des ensembles, des graphes et tout ce que vous pouvez imaginer d'autre. Voir *Mastering Algorithms with Perl* (O'Reilly, 1999), *Perl en action* (O'Reilly, France, 1999)[1], ou CPAN qui sert de dépôt central pour tout ce genre de modules. Mais vous n'aurez peut-être jamais besoin que de simples combinaisons de tableaux et de hachages, aussi est-ce ce dont nous allons parler dans ce chapitre.

Tableaux de tableaux

Il existe de nombreuses sortes de structures de données. La plus simple à construire est le tableau de tableaux, aussi appelé matrice à deux dimensions. (Ceci se généralise de manière évidente : un tableau de tableaux de tableaux est une matrice à trois dimensions, et ainsi de suite pour les dimensions supérieures.) C'est relativement simple à

1. *Perl Cookbook* (O'Reilly, 1998) pour la version originale.

comprendre, et presque tout ce qui s'applique dans ce cas s'appliquera aussi aux structures plus compliquées que nous explorerons dans les sections suivantes.

Création et accès à un tableau à deux dimensions

Voici comment construire un tableau à deux dimensions :

```
# Affecte une liste de références de tableaux à un tableau
@TdT = (
         [ "fred", "barney" ],
         [ "george", "jane", "elroy" ],
         [ "homer", "marge", "bart" ],
);

print $TdT[2][1];   # affiche "marge"
```

La liste complète est entourée par des parenthèses et non par des crochets car vous faites une affectation à une liste et non à une référence. Si vous préfériez une référence, vous utiliseriez des crochets :

```
# Crée une référence à un tableau de références.
$ref_a_TdT = [
    [ "fred", "barney", "pebbles", "bamm bamm", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_a_TdT->[2][3];   # affiche "judy"
```

Rappelez-vous qu'il y a un `->` implicite entre deux accolades ou crochets adjacents. Ces deux lignes :

```
$TdT[2][3]
$ref_a_TdT->[2][3]
```

sont donc équivalentes aux deux lignes suivantes :

```
$TdT[2]->[3]
$ref_a_TdT->[2]->[3]
```

Il n'y a cependant aucun `->` implicite avant la première paire de crochets, c'est pourquoi le déréférencement de `$ref_a_TdT` nécessite le `->` initial. N'oubliez pas que vous pouvez aussi compter à partir de la fin d'un tableau avec un indice négatif, donc :

```
$TdT[0][-2]
```

est l'avant-dernier élément de la première ligne.

Volez de vos propres ailes

Ces grosses affectations de listes sont belles et bonnes pour créer une structure de données fixe, mais comment faire si vous voulez construire chaque élément au vol ou construire la structure au coup par coup ?

Lisons une structure de données à partir d'un fichier. Nous supposerons qu'il s'agit d'un fichier texte, où chaque ligne est une rangée de la structure et est composée d'éléments

séparés par des blancs. Voici comment procéder :[2]

```
while (<>) {
    @tmp = split;         # Splite les éléments dans un tableau
    push @TdT, [ @tmp ];  # Ajoute une référence de tableau anonyme à @TdT
}
```

Bien sûr, vous n'avez pas besoin de donner un nom au tableau temporaire et vous pourriez donc aussi écrire :

```
while (<>) {
    push @TdT, [ split ];
}
```

Si vous voulez une référence à un tableau de tableaux, vous pouvez procéder ainsi :

```
while (<>) {
    push @$ref_a_TdT, [ split ];
}
```

Ces deux exemples ajoutent de nouvelles lignes au tableau de tableaux. Comment faire pour ajouter de nouvelles colonnes ? Si vous ne travaillez qu'avec des tableaux à deux dimensions, il est souvent plus facile d'utiliser une affectation simple :[3]

```
for $x (0 .. 9) {                       # Pour chaque ligne...
    for $y (0 .. 9) {                   # Pour chaque colonne...
        $TdT[$x][$y] = func($x, $y);    # ...affecte cette valeur
    }
}

for $x ( 0..9 ) {                       # Pour chaque ligne...
    $ref_a_TdT->[$x][3] = func2($x);    # ...affecte la quatrième colonne
}
```

L'ordre dans lequel vous affectez les éléments n'a pas d'importance, pas plus que le fait que les éléments indicés de `@TdT` existent déjà ou non ; Perl les créera pour vous si besoin est en mettant les éléments intermédiaires à la valeur indéfinie. (Perl créera même la référence initiale dans `$ref_a_TdT` pour vous s'il le faut.) Si vous voulez juste ajouter des éléments au bout d'une ligne, vous devrez faire quelque chose d'un peu plus étrange :

```
# Ajoute de nouvelles colonnes à une ligne existante.
push @{ $TdT[0] }, "wilma", "betty";
```

Remarquez que ceci ne marcherait pas :

```
push $TdT[0], "wilma", "betty";  # FAUX !
```

Cela ne compilerait même pas, car l'argument de `push` doit être un véritable tableau, et pas seulement une référence à un tableau. C'est pourquoi le premier argument doit absolument commencer par un caractère `@`. Ce qui suit le `@` est assez négociable.

---

2. Ici comme dans d'autres chapitres, nous omettons (pour la clarté des exemples) les déclarations `my` que vous devriez normalement mettre. Dans cet exemple, vous devriez écrire `my @tmp = split`.

3. Comme avec l'affectation temporaire précédente, nous avons simplifié ; les boucles de ce chapitre seraient très probablement écrites `for my $x` dans du code réel.

Utilisation et affichage

Affichons maintenant la structure de données. Si vous ne voulez qu'un seul élément, ceci suffit :

```
print $TdT[3][2];
```

Mais si vous voulez tout imprimer, vous ne pouvez pas vous contenter d'écrire :

```
print @TdT;          # FAUX
```

C'est incorrect car vous aurez des références sous forme de chaînes de caractères au lieu de vos données. Perl ne déréférence jamais automatiquement à votre place. Il vous faut donc construire une boucle ou deux. Le code qui suit affiche la structure complète, en bouclant sur les éléments de `@TdT` et en les déréférençant chacun à l'intérieur de l'instruction `print` :

```
for $ligne ( @TdT ) {
    print "@$ligne\n";
}
```

Pour garder la trace des indices, vous pouvez faire comme ceci :

```
for $i ( 0 .. $#TdT ) {
    print "La ligne $i est : @{$TdT[$i]}\n";
}
```

ou même cela (remarquez la boucle intérieure) :

```
for $i ( 0 .. $#TdT ) {
    for $j ( 0 .. $#{$TdT[$i]} ) {
        print "L'élément $i $j est $AoA[$i][$j]\n";
    }
}
```

Comme vous pouvez le voir, les choses se compliquent. C'est pourquoi il est parfois plus simple d'utiliser une variable temporaire en cours de route :

```
for $i ( 0 .. $#TdT ) {
    $ligne = $TdT[$i];
    for $j ( 0 .. $#{$ligne} ) {
        print "L'élément $i $j est $ligne->[$j]\n";
    }
}
```

Tranches

Si vous voulez accéder à une tranche (une partie de ligne) d'un tableau multidimensionnel, vous allez devoir faire des indexations subtiles. L'opérateur flèche nous donne une façon simple d'accéder à un élément unique, mais rien de tel n'existe pour les tranches. Vous pouvez toujours extraire les éléments de votre tranche un par un avec une boucle :

```
@partie = ();
for ($y = 7; $y < 13; $y++) {
    push @partie, $TdT[4][$y];
}
```

Cette boucle particulière pourrait être remplacée par une tranche de tableau :

```
@partie = @{ $TdT[4] } [ 7..12 ];
```

Si vous voulez une tranche à deux dimensions, par exemple, avec `$x` dans l'intervalle `4..8` et `$y` dans `7..12`, voici une manière de procéder :

```
@nouveauTdT = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $nouveauTdT[$x - $startx][$y - $starty] = $TdT[$x][$y];
    }
}
```

Dans cet exemple, les valeurs individuelles dans notre tableau d'arrivée à deux dimensions, `@nouveauTdT`, sont affectées une par une à partir d'un sous-tableau à deux dimensions de `@TdT`. Une autre possibilité consiste à créer des tableaux anonymes, chacun d'entre eux étant constitué de la tranche désirée d'un sous-tableau de `@TdT`, puis à mettre des références à ces tableaux anonymes dans `@nouveauTdT`. Nous pourrions alors stocker les références dans `@nouveauTdT` (indexé une seule fois, pour ainsi dire) au lieu de stocker des valeurs de sous-tableau dans un `@nouveauTdT` indexé deux fois. Cette méthode élimine la boucle intérieure :

```
for ($x = 4; $x <= 8; $x++) {
    push @nouveauTdT, [ @{ $TdT[$x] } [ 7..12 ] ];
}
```

Bien sûr, si vous faites cela souvent, vous devriez probablement écrire un sous-programme appelé par exemple `extraire_rectangle`. Et si vous faites cela très souvent avec de grandes séries de données multidimensionnelles, vous devriez probablement utiliser le module `PDL` (*Perl Data Language*), disponible sur CPAN.

Erreurs fréquentes

Comme nous l'avons déjà mentionné, les tableaux et les hachages de Perl sont à une dimension. En Perl, même les tableaux « multidimensionnels » sont en fait à une dimension, mais les valeurs le long de cette dimension sont des références à d'autres tableaux, qui agrègent plusieurs éléments en un seul. Si vous affichez ces valeurs sans les déréférencer, vous obtiendrez les références converties en chaînes en lieu et place des données désirées. Par exemple ces deux lignes :

```
@TdT = ( [2, 3], [4, 5, 7], [0] );
print "@TdT";
```

donneront quelque chose comme :

```
ARRAY(0x83c38) ARRAY(0x8b194) ARRAY(0x8b1d0)
```

D'un autre côté, cette ligne affiche `7` :

```
print $TdT[1][2];
```

Quand vous construisez un tableau de tableaux, n'oubliez pas de créer de nouvelles références pour les sous-tableaux. Sinon, vous vous contenterez de créer un tableau contenant le nombre d'éléments des sous-tableaux, comme ceci :

```
for $i (1..10) {
    @tableau = fonction($i);
    $TdT[$i] = @tableau;       # FAUX !
}
```

Ici on accède à `@tableau` en contexte scalaire, ce qui renvoie donc le nombre de ses éléments, lequel est affecté comme il se doit à `$TdT[$i]`. La manière correcte d'affecter la référence vous sera montrée dans un moment.

Après avoir fait l'erreur précédente, les gens réalisent qu'ils doivent affecter une référence. L'erreur qu'ils font naturellement ensuite implique de prendre et reprendre une référence vers toujours le même emplacement mémoire :

```
for $i (1..10) {
    @tableau = fonction($i);
    $TdT[$i] = \@tableau;      # ENCORE FAUX !
}
```

Chaque référence générée par la seconde ligne de la boucle `for` est la même, c'est-à-dire une référence au seul tableau `@tableau`. Certes, ce tableau change effectivement à chaque passage dans la boucle, mais quand tout est fini, `$TdT` contient 10 références au même tableau, qui contient maintenant la dernière série de valeurs qui lui ont été affectées. `print @{$TdT[1]}` présentera les mêmes valeurs que `print @{$TdT[2]}`.

Voici une approche qui aura plus de succès :

```
for $i (1..10) {
    @tableau = fonction($i);
    $TdT[$i] = [ @tableau ];   # CORRECT !
}
```

Les crochets autour de `@tableau` créent un nouveau tableau anonyme, dans lequel les éléments de `@tableau` sont copiés. Nous stockons ensuite une référence à ce nouveau tableau.

Un résultat similaire, bien que beaucoup plus difficile à lire, pourra être obtenu par :

```
for $i (1..10) {
    @tableau = fonction($i);
    @{$TdT[$i]} = @tableau;
}
```

Comme `$TdT[$i]` doit être une nouvelle référence, la référence se met à exister. Le `@` qui précède déréférence alors cette nouvelle référence, avec pour résultat d'affecter (en contexte de liste) les valeurs de `@tableau` au tableau référencé par `$TdT[$i]`. Pour des raisons de clarté, vous pouvez préférer éviter cette construction.

Mais il *existe* une situation dans laquelle vous voudrez l'utiliser. Supposons que `@TdT` est déjà un tableau de références à des tableaux. C'est-à-dire que vous avez fait des affectations comme :

```
$TdT[3] = \@tableau_original;
```

Et supposons maintenant que vous vouliez modifier `@tableau_original` (c'est-à-dire que vous voulez modifier la quatrième ligne de `$TdT`) de façon à ce qu'il réfère aux éléments de `@tableau`. Ce code fonctionnera :

```
@{$TdT[3]} = @tableau;
```

Dans ce cas, la référence ne change pas, mais les éléments du tableau référencé changent. Cela écrase les valeurs de `@tableau_original`.

Finalement, ce code dangereux en apparence fonctionne parfaitement :

```
for $i (1..10) {
    my @tableau = fonction($i);
    $TdT[$i] = \@tableau;
}
```

C'est parce que la variable à portée lexicale `my @tableau` est recréée complètement à chaque passage dans la boucle. Bien que vous ayez l'impression d'avoir stocké une référence à la même variable à chaque fois, ce n'est en fait pas le cas. C'est une subtile distinction, mais cette technique peut produire du code plus efficace, au risque de tromper les programmeurs moins éclairés. (C'est plus efficace car il n'y a pas de copie dans la dernière affectation.) D'un autre côté, s'il vous faut de toute façon copier les valeurs (ce que fait la première affectation de la boucle), vous pouvez aussi bien utiliser la copie impliquée par les crochets et éviter la variable temporaire :

```
for $i (1..10) {
    $TdT[$i] = [ fonction($i) ];
}
```

En résumé :

```
$TdT[$i] = [ @tableau ];   # Plus sûr, parfois plus rapide
$TdT[$i] = \@tableau;      # Rapide mais risqué, dépend du côté "my"
                           # de @tableau
@{ $TdT[$i] } = @tableau;  # Un peu rusé
```

Une fois que vous maîtrisez les tableaux de tableaux, vous voudrez vous mesurer à des structures de données plus complexes. Si vous cherchez les structures (`struct`) de C ou les enregistrements (`record`) de Pascal, vous ne trouverez aucun mot réservé en Perl qui en créera pour vous. Ce que vous avez en revanche est un système plus flexible. Si votre idée d'une structure est moins flexible que cela, ou si vous voulez fournir à vos utilisateurs quelque chose de plus rigide et de plus opaque, alors vous pouvez utiliser les fonctionnalités orientées objet détaillées au chapitre 12, *Objets*.

Perl dispose de deux manières d'organiser les données : comme des listes ordonnées dans des tableaux, dont l'accès aux éléments se fait par leur position, ou comme des listes non ordonnées de couples clé/valeur stockés dans des hachages et auxquels on accède par leur nom. La meilleure façon de représenter un enregistrement en Perl consiste à utiliser une référence de hachage, mais la manière d'organiser de tels enregistrements peut varier. Si vous désirez conserver une liste ordonnée de ces enregistrements et y accéder par leur numéro, vous utiliserez un tableau de références de hachages pour stocker les enregistrements. Ou bien, si vous préférez accéder à ces enregistrements par leurs noms, vous maintiendrez un hachage de références à des hachages. Vous pourriez même faire les deux à la fois avec un pseudo-hachage.

Dans les sections suivantes, vous trouverez des exemples de code décrivant comment composer (à partir de rien), générer (depuis d'autres sources), accéder à et afficher plusieurs structures de données différentes. Nous montrerons d'abord trois combinaisons évidentes de hachages et de tableaux, suivies par un hachage de fonctions puis des structures de données plus irrégulières. Nous terminerons par la présentation d'une manière

de sauver ces structures de données. Ces exemples supposent que vous vous êtes déjà familiarisé avec les explications qui précèdent dans ce chapitre.

Hachages de tableaux

Utilisez un hachage de tableaux quand vous voulez accéder à chaque tableau par une chaîne particulière plutôt que par un numéro d'indice. Dans notre exemple de personnages d'émissions de télévision, au lieu de parcourir la liste de noms à partir de la zéroième émission, la première et ainsi de suite, nous allons mettre en place notre structure de façon à pouvoir obtenir la liste des personnages à partir du nom du dessin animé.

Comme notre structure de données est un hachage, nous ne pouvons pas en ordonner le contenu, mais nous pouvons utiliser la fonction `sort` pour spécifier un ordre de sortie particulier.

Composition d'un hachage de tableaux

Vous pouvez créer un hachage de tableaux anonymes comme suit :

```
# Nous omettons habituellement les guillemets quand les clés sont
# des identificateurs.
%HdT = (
    flintstones    => [ "fred", "barney" ],
    jetsons        => [ "george", "jane", "elroy" ],
    simpsons       => [ "homer", "marge", "bart" ],
);
```

Pour ajouter un nouveau tableau au hachage, vous pouvez simplement écrire :

```
$HdT{teletubbies} = [ "tinky winky", "dipsy", "laa-laa", "po" ];
```

Génération d'un hachage de tableaux

Voici quelques techniques pour remplir un hachage de tableaux. Pour le lire à partir d'un fichier avec le format suivant :

```
flintstones: fred barney wilma dino
jetsons:     george jane elroy
simpsons:    homer marge bart
```

vous pourriez utiliser l'une des deux boucles suivantes :

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HdT{$1} = [ split ];
}

while ( $ligne = <> ) {
    ($qui, $reste) = split /:\s*/, $ligne, 2;
    @champs = split ' ', $reste;
    $HdT{$qui} = [ @champs ];
}
```

Si vous avez un sous-programme `get_famille` qui renvoie un tableau, vous pouvez vous en servir pour remplir `%HdT` avec l'une de ces deux boucles :

```
for $groupe ( "simpsons", "jetsons", "flintstones" ) {
    $HdT{$groupe} = [ get_famille($groupe) ];
}

for $groupe ( "simpsons", "jetsons", "flintstones" ) {
    @membres = get_famille($groupe);
    $HdT{$groupe} = [ @membres ];
}
```

Vous pouvez ajouter de nouveaux membres à un tableau existant comme ceci :

```
push @{ $HdT{flintstones} }, "wilma", "pebbles";
```

Utilisation et affichage d'un hachage de tableaux

Vous pouvez fixer le premier élément d'un tableau en particulier comme suit :

```
$HdT{flintstones}[0] = "Fred";
```

Pour mettre en majuscule l'initiale du deuxième Simpson, appliquez une substitution à l'élément de tableau approprié :

```
$HdT{simpsons}[1] =~ s/(\w)/\u$1/;
```

Vous pouvez afficher toutes les familles en bouclant sur toutes les clés du hachage :

```
for $famille ( keys %HdT ) {
    print "$famille : @{ $HdT{$famille} }\n";
}
```

Avec un petit effort supplémentaire, vous pouvez également ajouter les indices de tableau :

```
for $famille ( keys %HoA ) {
    print "$famille : ";
    for $i ( 0 .. $#{ $HdT{$famille} } ) {
        print " $i = $HdT{$famille}[$i]";
    }
    print "\n";
}
```

Ou trier les tableaux par leur nombre d'éléments :

```
for $famille ( sort { @{$HdT{$b}} <=> @{$HdT{$a}} } keys %HdT ) {
    print "$famille : @{ $HdT{$famille} }\n"
}
```

Ou même trier les tableaux par leur nombre d'éléments puis trier les éléments ASCII-bétiquement (ou pour être précis, « utf8-tiquement ») :

```
# Affiche l'ensemble trié par le nombre de personnes et leurs noms.
for $famille ( sort { @{$HdT{$b}} <=> @{$HdT{$a}} } keys %HdT ) {
    print "$famille : ", join(", ", sort @{ $HdT{$famille} }), "\n";
}
```

Tableaux de hachages

Un tableau de hachages est utile quand vous avez un tas d'enregistrements auxquels vous aimeriez accéder séquentiellement et que chaque enregistrement comporte des couples clé/valeur. Les tableaux de hachages sont moins fréquemment utilisés que les autres structures de ce chapitre.

Composition d'un tableau de hachages

Vous pouvez créer un tableau de hachages anonymes comme suit :

```
@TdH = (
    {
       mari     => "barney",
       femme    => "betty",
       fils     => "bamm bamm",
    },
    {
       mari    => "george",
       femme   => "jane",
       fils    => "elroy",
    },
    {
       mari    => "homer",
       femme   => "marge",
       fils    => "bart",
    },
  );
```

Pour ajouter un nouveau hachage au tableau, vous pouvez simplement écrire :

```
push @TdH, { mari => "fred", femme => "wilma", fille => "pebbles" };
```

Génération d'un tableau de hachages

Voici quelques techniques pour remplir un tableau de hachages. Pour lire depuis un fichier écrit selon le format suivant :

```
mari=fred ami=barney
```

vous pourriez utiliser l'une des deux boucles suivantes :

```
while ( <> ) {
    $enrt = {};
    for $champ ( split ) {
        ($cle, $valeur) = split /=/, $champ;
        $enrt->{$cle} = $valeur;
    }
    push @TdH, $enrt;
}
```

```
while ( <> ) {
    push @TdH, { split /[\s=]+/ };
}
```

Si vous avez un sous-programme `get_paire_suivante` qui renvoie des couples clé/valeur, vous pouvez l'utiliser pour remplir `@TdH` avec l'un de ces deux boucles :

```
while ( @champs = get_paire_suivante() ) {
    push @TdH, { @champs };
}

while (<>) {
    push @TdH, { get_paire_suivante($_) };
}
```

Vous pouvez ajouter de nouveaux éléments à un hachage existant comme ceci :

```
$TdH[0]{animal} = "dino";
$TdH[2]{animal} = "petit papa Noël";
```

Utilisation et affichage d'un tableau de hachages

Vous pouvez modifier un couple clé/valeur d'un hachage particulier comme suit :

```
$TdH[0]{mari} = "fred";
```

Pour mettre en majuscule le mari du deuxième hachage appliquez une substitution :

```
$TdH[1]{mari} =~ s/(\w)/\u$1/;
```

Vous pouvez afficher toutes les données comme suit :

```
for $href ( @TdH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}
```

et avec les indices :

```
for $i ( 0 .. $#TdH ) {
    print "$i est { ";
    for $role ( keys %{ $TdH[$i] } ) {
        print "$role=$TdH[$i]{$role} ";
    }
    print "}\n";
}
```

Hachages de hachages

Le hachage multidimensionnel est la plus souple des structures emboîtées de Perl. Cela revient à construire un enregistrement qui contient d'autres enregistrement Vous indexez à chaque niveau avec des chaînes (entre guillemets si besoin est). Souvenez vous cependant que les couples clé/valeur du hachage ne sortiront dans aucun ordre particu-

lier ; vous pouvez utiliser la fonction `sort` pour récupérer les couples dans l'ordre qui vous intéresse.

Composition d'un hachage de hachage

Vous pouvez créer un hachage de hachages anonymes comme suit :

```
%HdH = (
    flintstones => {
        mari      => "fred",
        pote      => "barney",
    },
    jetsons => {
        mari      => "george",
        femme     => "jane",
        "son fils"=> "elroy",  # Guillemets nécessaires sur la clé
    },
    simpsons => {
        mari      => "homer",
        femme     => "marge",
        fiston    => "bart",
    },
);
```

Pour ajouter un autre hachage anonyme à `%HdH`, vous pouvez simplement écrire :

```
$HdH{ mash } = {
    capitaine => "pierce",
    major     => "burns",
    caporal   => "radar",
};
```

Génération d'un hachage de hachages

Voici quelques techniques pour remplir un hachage de hachages. Pour lire des données à partir d'un fichier dans le format suivant :

```
flintstones: mari=fred pote=barney femme=wilma animal=dino
```

vous pourriez utiliser l'une des deux boucles suivantes :

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $qui = $1;
    for $champ ( split ) {
        ($cle, $valeur) = split /=/, $champ;
        $HdH{$qui}{$cle} = $valeur;
    }
}

while ( <> ) {
    next unless s/^(.*?):\s*//;
    $qui = $1;
```

```
    $enrt = {};
    $HdH{$qui} = $enrt;
    for $champ ( split ) {
        ($cle, $valeur)  = split /=/, $champ;
        $enrt->{$cle} = $valeur;
    }
}
```

Si vous avez un sous-programme `get_famille` qui renvoie une liste de couples clé/valeur, vous pouvez vous en servir pour remplir `%HdH` avec l'un de ces trois bouts de code :

```
for $groupe ( "simpsons", "jetsons", "flintstones" ) {
    $HdH{$groupe} = { get_famille($groupe) };
}

for $groupe ( "simpsons", "jetsons", "flintstones" ) {
    @membres = get_famille($groupe);
    $HdH{$groupe} = { @membres };
}

sub hash_familles {
    my @ret;
    for $groupe ( @_ ) {
        push @ret, $groupe, { get_famille($groupe) };
    }
    return @ret;
}
%HdH = hash_familles( "simpsons", "jetsons", "flintstones" );
```

Vous pouvez ajouter de nouveaux membres à un hachage comme suit :

```
%nouveaux = (
    femme  => "wilma",
    animal => "dino";
);
for $quoi (keys %nouveaux) {
    $HdH{flintstones}{$quoi} = $nouveaux{$quoi};
}
```

Utilisation et affichage d'un hachage de hachages

Vous pouvez modifier un couple clé/valeur d'un hachage particulier comme suit :

```
$HdH{flintstones}{femme} = "wilma";
```

Pour mettre en majuscule la valeur associée à une clé particulière, appliquez une substitution à un élément :

```
$HdH{jetsons}{'son fils'} =~ s/(\w)/\u$1/;
```

Vous pouvez afficher toutes les familles en bouclant sur les clés du hachage externe, puis en bouclant sur les clés du hachage interne :

```
for $famille ( keys %HdH ) {
    print "$famille : ";
```

```
    for $role ( keys %{ $HdH{$famille} } ) {
         print "$role=$HdH{$famille}{$role} ";
    }
    print "\n";
}
```

Dans les hachages très grands, il peut être un peu plus rapide de récupérer à la fois les clés et les valeurs en utilisant each (ce qui exclut le tri) :

```
while ( ($famille, $roles) = each %HdH ) {
    print "$famille : ";
    while ( ($role, $personne) = each %$roles ) {
        print "$role=$personne ";
    }
    print "\n";
}
```

(Hélas, ce sont les grands hachages qui ont vraiment besoin d'être triés, sinon vous ne trouverez jamais ce que vous cherchez dans ce qui est affiché.) Vous pouvez trier les familles, puis les rôles, comme suit :

```
for $famille ( sort keys %HdH ) {
    print "$famille : ";
    for $role ( sort keys %{ $HdH{$famille} } ) {
         print "$role=$HdH{$famille}{$role} ";
    }
    print "\n";
}
```

Pour trier les familles par leur nombre de membres (plutôt qu'ASCII-bétiquement (ou utf8-tiquement)), vous pouvez utiliser keys en contexte scalaire :

```
for $famille ( sort { keys %{$HdH{$a}} <=> keys %{$HdH{$b}} } keys %HdH ) {
    print "$famille : ";
    for $role ( sort keys %{ $HdH{$famille} } ) {
         print "$role=$HdH{$family}{$role} ";
    }
    print "\n";
}
```

Pour trier les membres d'une famille dans un certain ordre, vous pouvez leur donner à chacun un rang :

```
$i = 0;
for ( qw(mari femme fils fille pote animal) ) { $rang{$_} = ++$i }

for $famille ( sort { keys %{$HdH{$a}} <=> keys %{$HdH{$b}} } keys %HdH ) {
    print "$famille : ";
    for $role ( sort { $rang{$a} <=> $rang{$b} } keys %{ $HdH{$famille} } ) {
        print "$role=$HdH{$famille}{$role} ";
    }
    print "\n";
}
```

Hachages de fonctions

Quand vous écrivez une application complexe ou un service réseau en Perl, vous voulez probablement mettre un grand nombre de commandes à la disposition de vos utilisateurs. Un tel programme pourrait avoir du code comme celui-ci pour examiner le choix de l'utilisateur et agir en conséquence :

```
if    ($cmd =~ /^exit$/i)     { exit }
elsif ($cmd =~ /^help$/i)     { show_help() }
elsif ($cmd =~ /^watch$/i)    { $watch = 1 }
elsif ($cmd =~ /^mail$/i)     { mail_msg($msg) }
elsif ($cmd =~ /^edit$/i)     { $edited++; editmsg($msg); }
elsif ($cmd =~ /^delete$/i)   { confirm_kill() }
else {
    warn "Commande inconnue : ‘$cmd’; Essayez ‘help’ la prochaine fois\n";
}
```

Vous pouvez également stocker des références à des fonctions dans vos structures de données, tout comme vous pouvez y stocker des tableaux ou des hachages :

```
%HdF = (                            # Compose un hachage de fonctions
    exit    =>  sub { exit },
    help    =>  \&show_help,
    watch   =>  sub { $watch = 1 },
    mail    =>  sub { mail_msg($msg) },
    edit    =>  sub { $edited++; editmsg($msg); },
    delete  =>  \&confirm_kill,
);

if   ($HdF{lc $cmd}) { $HdF{lc $cmd}->() }   # Appelle la fonction
else { warn "Commande inconnue : ‘$cmd’; Essayez ‘help’ la prochaine fois\n"
}
```

Dans l'avant-dernière ligne, nous vérifions si le nom de commande spécifié (en minuscules) existe dans notre « table de distribution » `%HdF`. Si c'est le cas, nous appelons la commande appropriée en déréférençant la valeur du hachage comme une fonction et passons à cette fonction une liste d'arguments vide. Nous pourrions aussi l'avoir déréférencée comme `&{ $HoF{lc $cmd} }()`, ou à partir de la version 5.6 de Perl, simplement par `$HoF{lc $cmd}()`.

Enregistrements plus élaborés

Jusqu'ici nous avons vu dans ce chapitre des structures de données simples, homogènes et à deux niveaux : chaque élément contient le même genre de référent que tous les autres éléments de ce niveau. Ce n'est pas indispensable. Tout élément peut contenir n'importe quel type de scalaire, ce qui signifie qu'il peut s'agir d'un chaîne, d'un nombre ou d'une référence à n'importe quoi d'autre. La référence peut être une référence à un tableau ou à un hachage, ou à un pseudo-hachage, ou une référence à une fonction nommée ou anonyme, ou un objet. La seule chose que vous ne pouvez pas faire est de mettre plusieurs référents dans un seul scalaire. Si vous vous retrouvez à essayer de faire ce genre de chose, c'est le signe que vous avez besoin d'une référence à un tableau ou un hachage pour superposer plusieurs valeurs en une seule.

Dans les sections qui suivent, vous trouverez des exemples de code destinés à montrer ce qu'il est possible de stocker dans un enregistrement, que nous implémenterons comme une référence à un hachage. Les clés sont en majuscules, une convention parfois employée quand le hachage est utilisé comme un type d'enregistrement spécifique.

Composition, utilisation et affichage d'enregistrements plus élaborés

Voici un enregistrement avec six champs de types différents :

```
$enrt = {
    TEXTE     => $chaine,
    SEQUENCE  => [ @anc_valeurs ],
    RECHERCHE => { %une_table },
    CODE1     => \&une_fonction,
    CODE2     => sub { $_[0] ** $_[1] },
    HANDLE    => \*STDOUT,
};
```

Le champ `TEXTE` est une simple chaîne, que vous pouvez donc afficher :

```
print $enrt->{TEXT};
```

`SEQUENCE` et `RECHERCHE` sont des références normales à un tableau et à un hachage :

```
print $enrt->{SEQUENCE}[0];
$dernier = pop @{ $enrt->{SEQUENCE} };

print $enrt->{RECHERCHE}{"cle"};
($premier_c, $premier_v) = each %{ $enrt->{RECHERCHE} };
```

`CODE1` est un sous-programme nommé et `CODE2` un sous-programme anonyme, mais sont appelés de la même façon :

```
$reponse1 = $enrt->{CODE1}->($arg1, $arg2);
$reponse2 = $enrt->{CODE2}->($arg1, $arg2);
```

Avec une paire d'accolades supplémentaires, pour pouvez traiter `$rec->{HANDLE}` comme un objet indirect :

```
print { $enrt->{HANDLE} } "une chaîne\n";
```

Si vous utilisez le module `FileHandle`, vous pouvez même traiter le handle comme un objet normal :

```
use FileHandle;
$enrt->{HANDLE}->autoflush(1);
$enrt->{HANDLE}->print("une chaîne\n");
```

Composition, utilisation et affichage d'enregistrements encore plus élaborés

Naturellement, les champs de vos structures de données peuvent eux-mêmes être des structures de données arbitrairement complexes :

```
%TV = (
    flintstones => {
        serie    => "flintstones",
        soirs    => [ "lundi", "jeudi", "vendredi" ],
        membres  => [
            { nom => "fred",    role => "mari",    age  => 36, },
            { nom => "wilma",   role => "femme",   age  => 31, },
            { nom => "pebbles", role => "enfant",  age  =>  4, },
        ],
    },

    jetsons     => {
        serie    => "jetsons",
        soirs    => [ "mercredi", "samedi" ],
        membres  => [
            { nom => "george",  role => "mari",    age  => 41, },
            { nom => "jane",    role => "femme",   age  => 39, },
            { nom => "elroy",   role => "enfant",  age  =>  9, },
        ],
    },

    simpsons    => {
        serie    => "simpsons",
        soirs    => [ "lundi" ],
        membres  => [
            { nom => "homer", role => "mari",      age => 34, },
            { nom => "marge", role => "femme",     age => 37, },
            { nom => "bart",  role => "enfant",    age => 11, },
        ],
    },
);
```

Génération d'un hachage d'enregistrements complexes

Comme Perl est assez bon pour analyser des structures de données complexes, pour pouvez aussi bien mettre vos déclarations de données dans un fichier séparé comme du code Perl normal, puis le charger avec les fonctions intégrées `do` ou `require`. Une autre approche courante est d'utiliser un module de CPAN (comme `XML::Parser`) pour charger des structures de données quelconques écrites dans un autre langage (comme XML).

Vous pouvez construire vos structures de données petit à petit :

```
$enrt = {};
$enrt->{serie} = "flintstones";
$enrt->{soirs} = [ quels_jours() ];
```

Ou les lire à partir d'un fichier (qu'on suppose ici écrit sous la forme `champ=valeur`) :

```
@membres = ();
while (<>) {
     %champs = split /[\s=]+/;
     push @membres, { %champs };
}
```

```
$enrt->{membres} = [ @membres ];
```

Puis les inclure dans des structures de données plus grandes, indexées par l'un des sous-champs :

```
$TV{ $enrt->{serie} } = $enrt;
```

Vous pouvez utiliser des champs supplémentaires pour éviter de dupliquer les données. Par exemple, vous pourriez vouloir inclure un champ `"enfants"` dans l'enregistrement d'une personne, lequel pourrait être une référence à un tableau contenant des références aux propres enregistrements des enfants. En ayant des parties de votre structure de données qui pointent sur d'autres, vous éviter les dérives résultant de la mise à jour des données à un endroit mais pas à un autre :

```
for $famille (keys %TV) {
    my $enrt = $TV{$famille};   # pointeur temporaire
    my @enfants = ();
    for $personne ( @{$enrt->{membres}} ) {
        if ($personne->{role} =~ /enfant|fils|fille/) {
            push @enfants, $personne;
        }
    }
    # $enrt et $TV{$famille} pointent sur les mêmes données !
    $enrt->{enfants} = [ @enfants ];
}
```

L'affectation `$enrt->{enfants} = [ @enfants ]` copie le contenu du tableau — mais ce sont simplement des références à des données non copiées. Cela signifie que si vous modifiez l'âge de Bart comme suit :

```
$TV{simpsons}{enfants}[0]{age}++;            # incrémente à 12
```

alors vous verrez le résultat suivant, car `$TV{simpsons}{enfants}[0]` et `$TV{simpsons}{membres}[2]` pointent tous deux vers le même hachage anonyme :

```
print $TV{simpsons}{membres}[2]{age};     # affiche 12 également
```

Et maintenant pour imprimer toute la structure de données `%TV` :

```
for $famille ( keys %TV ) {
    print "la famille $famille";
    print " passe ", join (" et ", @{ $TV{$famille}{nights} }), "\n";
    print "ses membres sont :\n";
    for $qui ( @{ $TV{$famille}{membres} } ) {
        print " $qui->{name} ($qui->{role}), age $qui->{age}\n";
    }
    print "enfants: ";
    print join (", ", map { $_->{nom} } @{ $TV{$famille}{enfants} } );
    print "\n\n";
}
```

Sauvegarde de structures de données

Il existe plusieurs manières de sauver vos structures de données pour les utiliser avec un autre programme par la suite. La manière la plus simple consiste à utiliser le module

`Data::Dumper` de Perl, qui convertit une structure de données (potientiellement autoréférente) en une chaîne qui peut être sauvée pour être reconstituée par la suite avec `eval` ou `do`.

```
use Data::Dumper;
$Data::Dumper::Purity = 1;       # car %TV est autoréférent
open (FILE, "> tvinfo.perldata") or die "impossible d'ouvrir tvinfo: $!";
print FILE Data::Dumper->Dump([\%TV], ['*TV']);
close FILE                       or die "impossible de fermer tvinfo: $!";
```

Un autre programme (ou le même programme) pourra lire le fichier plus tard :

```
open (FILE, "< tvinfo.perldata") or die "impossible d'ouvrir tvinfo: $!";
undef $/;                        # lit tout le fichier d'un coup
eval <FILE>;                     # recrée %TV
die "impossible de recréer les données depuis tvinfo.perldata: $@" if $@;
close FILE                       or die "impossible de fermer tvinfo: $!";
print $TV{simpsons}{membres}[2]{age};
```

ou simplement :

```
do "tvinfo.perldata"           or die "impossible de recréer tvinfo: $! $@";
print $TV{simpsons}{membres}[2]{age};
```

Beaucoup d'autres solutions sont disponibles, avec des formats de stockage qui vont des données binaires (très rapide) au XML (très interopérable). Visitez vite un miroir du CPAN près de chez vous !

10

Paquetages

Dans ce chapitre, nous allons pouvoir commencer à nous amuser, car nous allons aborder la bonne conception des logiciels. Pour ce faire, parlons de Paresse, d'Impatience et d'Orgueil, les bases de la bonne conception des logiciels.

Nous sommes tous tombés dans le piège du copier/coller alors qu'il fallait définir une abstraction de haut niveau, ne serait-ce qu'une boucle ou une routine simple.[1] Il est vrai que certains partent à l'autre extrême en définissant des tas croissants d'abstractions de plus haut niveau alors qu'il aurait suffi d'un copier/coller.[2] Mais en gé7néral, la plupart d'entre nous devraient penser à employer plus d'abstraction plutôt que moins.

Entre les deux se trouvent ceux qui ont une vision équilibrée du degré d'abstraction nécessaire, mais qui se lancent dans la conception de leurs propres abstractions alors qu'ils devraient utiliser un code existant.[3]

Chaque fois que vous êtes tenté d'adopter l'une ou l'autre de ces attitudes, il vous faut d'abord vous asseoir et penser à ce qui sera le mieux pour vous et votre prochain à long terme. Si vous allez déverser vos énergies créatives dans une masse de code, pourquoi ne pas rendre le monde meilleur pendant que vous y êtes ? (Même si vous ne visez qu'à *réussir* le programme, vous devez vous assurer qu'il convient à la bonne niche écologique.)

La première étape vers une programmation écologiquement soutenable est simple : pas de détritus dans le parc. Lorsque vous écrivez un bout de code, pensez à lui donner son propre espace de noms, de sorte que vos variables et vos fonctions n'écrasent pas celles de quelqu'un d'autre, ou vice-versa. Un espace de noms est un peu comme votre maison, dans laquelle il vous est permis d'être aussi désordonné que vous le voulez, tant que votre interface externe avec les autres citoyens reste un tant soit peu correcte. En Perl, un espace de noms s'appelle un *paquetage* (`package`). Les paquetages fournissent la bri-

1. Ce qui est une mauvaise forme de Paresse.

2. Ce qui est une forme d'Orgueil mal placé.

3. Ce qui, vous l'avez deviné, est une mauvaise forme d'Impatience. Mais si vous voulez vraiment réinventer la roue, essayez au moins d'en inventer une meilleure.

que de base sur laquelle les concepts de plus haut niveau de module et de classe sont élaborés.

Tout comme la notion de « maison », la notion de « paquetage » est un peu nébuleuse. Les paquetages sont indépendants des fichiers. On peut aussi bien avoir plusieurs paquetages dans un seul fichier, ou un unique paquetage réparti sur plusieurs fichiers, tout comme votre maison peut être soit une petite mansarde dans un grand bâtiment (si vous êtes un artiste sans le sou), soit un ensemble de bâtiments (si votre nom se trouve être Reine Elizabeth). Mais une maison est habituellement composée d'un seul bâtiment, comme un paquetage est habituellement composé d'un seul fichier. Avec Perl, vous pouvez mettre un paquetage dans un seul fichier, tant que vous acceptez de donner au fichier le même nom que le paquetage et d'utiliser l'extension *.pm*, qui est l'abréviation de « perl module ». Le *module* est l'unité fondamentale de réutilisation de Perl. En effet, pour utiliser un module, vous vous servez de la commande `use`, qui est une directive de compilation qui contrôle l'importation des fonctions et des variables d'un module. Chaque exemple de `use` que vous avez vu jusqu'à maintenant est un exemple de réutilisation de module.

Le *Comprehensive Perl Archive Network*, ou CPAN, est l'endroit où mettre vos modules si d'autres personnes pouvaient les trouver utiles. Perl a prospéré grâce à la volonté des programmeurs à partager les fruits de leur travail avec la communauté. Évidemment, CPAN est également l'endroit où vous pouvez trouver des modules que d'autres ont gentiment téléchargé pour l'usage de tous. Voir le chapitre 22, *CPAN*, et *www.cpan.org* plus de détails.

Depuis 25 ans environ, il est à la mode de concevoir des langages informatiques qui imposent un état de paranoïa. On attend de vous de programmer chaque module comme s'il était en état de siège. Il existe sans doute certaines cultures féodales au sein desquelles ceci est justifiable, mais toutes les cultures ne sont pas ainsi faites. Dans la culture Perl, par exemple, on attend de vous que vous restiez en dehors de la maison de quelqu'un parce que vous n'y avez pas été invité, et non parce qu'il y a des barreaux aux fenêtres.[4]

Ceci n'est pas un livre sur la programmation orientée objet, et nous ne sommes pas là pour vous convertir en fanatique délirant de l'orienté objet, même si justement vous souhaitez être converti. Il existe déjà de nombreux livres disponibles sur ce sujet. La philosophie Perl de la conception orientée objet correspond parfaitement à la philosophie Perl de tout le reste ; utilisez la conception orientée objet (OO) là où elle a un sens, et évitez-la là où elle n'en a pas. À vous de voir.

En parler OO, chaque objet appartient à un groupement appelé *classe*. En Perl, les classes, les paquetages et les modules sont tellement liés que les débutants peuvent souvent les considérer interchangeables. Typiquement, une classe est implémentée par un module qui définit un paquetage ayant le même nom que la classe. Nous expliquerons cela dans les quelques chapitres qui suivent.

Lorsque vous utilisez un module avec `use`, vous bénéficiez de la réutilisation directe de logiciel. Avec les classes, vous bénéficiez de la réutilisation indirecte de logiciel lorsqu'une classe en utilise une autre par héritage. Avec les classes, vous gagnez quelque chose de plus : une interface propre avec un autre espace de noms. On n'accède à quelque

4. Mais Perl fournit aussi des barreaux si vous le voulez. Voir *Gestion des données non sûres* au chapitre 23, *Sécurité*.

chose dans une classe qu'indirectement, isolant ainsi la classe du monde extérieur.

Comme nous l'avons indiqué dans le chapitre 8, *Références*, la programmation orientée objet est réalisée par des références dont les référents savent à quelle classe ils appartiennent. En fait, maintenant que vous connaissez les références, vous connaissez presque tout ce que les objets ont de difficile. Le reste « se trouve sous les doigts », comme dirait un pianiste. Mais cependant il va falloir pratiquer un peu.

Un de vos exercices de gammes de base consiste à apprendre comment protéger différents bouts de code pour qu'une partie n'altère pas par inadvertance les variables des autres parties. Chaque bout de code appartient à un *paquetage* bien défini, ce qui détermine quelles variables et quelles fonctions sont à sa disposition. Lorsque Perl rencontre un bout de code, il le compile dans le *paquetage courant*. Le paquetage courant initial est appelé « `main` », mais vous pouvez passer du paquetage courant à un autre à tout moment avec la déclaration `package`. Le paquetage courant détermine quelle table de symboles est utilisée pour trouver vos variables, fonctions, handles d'entrée/sortie, et formats.

Toute variable qui n'est pas déclarée avec `my` est associée à un paquetage — même des variables apparemment omniprésentes comme `$_` et `%SIG`. En fait, les variables globales n'existent pas vraiment en Perl, ce sont simplement des variables de paquetage. (Les identificateurs spéciaux comme `_` et `SIG` ne sont qu'apparemment globaux, car ils sont par défaut dans le paquetage `main` plutôt que dans le paquetage courant.)

La portée d'une déclaration `package` commence à la déclaration elle-même et se termine à la fin de la portée qui la contient (bloc, fichier ou `eval` — celle qui arrive en premier) ou jusqu'à une autre déclaration `package` au même niveau, qui remplace la précédente. (Ceci est une pratique courante.)

Tous les identificateurs ultérieurs (dont ceux qui sont déclarés avec `our`, mais non ceux qui sont déclarés avec `my` ou qualifiés avec le nom d'un autre paquetage) seront placés dans la table de symboles appartenant au paquetage courant. (Les variables déclarées avec `my` sont indépendantes des paquetages. Elles sont toujours visibles à l'intérieur, et uniquement à l'intérieur, de la portée les contenant, quelles que soient les déclarations de paquetage.)

Une déclaration de paquetage sera habituellement la première instruction d'un fichier destiné à être inclus par `require` ou `use`. Mais ce n'est qu'une convention. Vous pouvez mettre une déclaration `package` partout où vous pouvez mettre une instruction. Vous pouvez même la mettre à la fin d'un bloc, auquel cas elle n'aura pas le moindre effet. Vous pouvez passer d'un paquetage à un autre à plus d'un endroit, car une déclaration de paquetage ne fait que sélectionner la table de symboles à utiliser par le compilateur pour le reste de ce bloc. (C'est ainsi qu'un paquetage donné peut se répartir sur plusieurs fichiers.)

Vous pouvez faire référence aux identificateurs[5] appartenant à d'autres paquetages en

5. Par identificateurs, nous désignons les noms utilisés comme clefs de table de symboles pour accéder aux variables scalaires, variables de tableau, variables de hachage, fonctions, handles de fichier ou de répertoire et formats. Syntaxiquement parlant, les étiquettes sont aussi des identificateurs, mais elles ne sont pas placées dans une table de symboles précise ; elles sont plutôt rattachées directement aux instructions de votre programme. Les étiquettes ne peuvent être qualifiées par un paquetage.

les préfixant (« qualifiant ») avec leur nom de paquetage et un double deux-points : `$Paquetage::Variable`. Si le nom du paquetage est nul, c'est le paquetage principal (`main`) qui est pris par défaut. C'est-à-dire que `$::chaude` est équivalent à `$main::chaude`. [6]

L'ancien délimiteur de paquetage était l'apostrophe, ainsi dans d'anciens programmes Perl vous verrez des variables comme `$main'chaude` et `$unpack'debiere`. Mais le double deux-points est conseillé parce qu'il est plus lisible, d'une part pour les humains, et d'autre part pour les macros *emacs*. Il donne aussi aux programmeurs C++ le sentiment qu'ils savent ce qui se passe — alors que l'apostrophe était là pour donner aux programmeurs Ada le sentiment de savoir ce qui se passe. Comme la syntaxe démodée est encore supportée pour la compatibilité arrière, si vous essayez d'utiliser une chaîne comme `"This is $owner's house"`, vous accèderez à `$owner::s`, c'est à dire la variable `$s` dans le paquetage `owner`, ce qui n'est probablement pas ce que vous vouliez dire. Utilisez des accolades pour éliminer l'ambiguïté, par exemple `"This is ${owner}'s house"`.

Le double deux-points peut être utilisé pour mettre bout à bout des identificateurs dans un nom de paquetage : `$Rouge::Bleu::var`. Ce qui veut dire : le `$var` appartenant au paquetage `Rouge::Bleu`. Le paquetage `Rouge::Bleu` n'a rien à voir avec un quelconque paquetage `Rouge` ou `Bleu` existant éventuellement. Une relation entre `Rouge::Bleu` et `Rouge` ou `Bleu` peut avoir un sens pour la personne qui écrit ou utilise le programme, mais n'en a aucun pour Perl. (Enfin, à part le fait que, dans l'implémentation actuelle, la table de symboles `Rouge::Bleu` se trouve être rangée dans la table de symboles `Rouge`. Mais le langage Perl n'en fait aucun usage direct.)

Pour cette raison, chaque déclaration `package` doit déclarer un nom de paquetage complet. Aucun nom de paquetage ne sous-entend un quelconque « préfixe », même s'il est (apparemment) déclaré à l'intérieur de la portée d'une autre déclaration de paquetage.

Seuls les identificateurs (les noms commençant par une lettre ou un souligné) sont rangés dans la table de symboles d'un paquetage. Tous les autres symboles sont gardés dans le paquetage `main`, notamment toutes les variables non-alphabétiques comme `$!`, `$?`, et `$_`. De plus, lorqu'ils ne sont pas qualifiés, les identificateurs `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`, `ENV`, `INC` et `SIG` se trouvent obligatoirement dans le paquetage `main`, même lorsqu'ils sont utilisés à d'autres fins que celles auxquelles ils sont destinés par construction. Ne nommez pas votre paquetage `m`, `s`, `y`, `tr`, `q`, `qq`, `qr`, `qw`, ou `qx` sauf si vous cherchez beaucoup d'ennuis. Par exemple, vous ne pourrez pas utiliser la forme qualifiée d'un identificateur comme handle de fichier, car il sera interprété au contraire comme une recherche de motif, une substitution ou une translittération.

Il y a longtemps, les variables commençant avec un souligné appartenaient obligatoirement au paquetage `main`, mais nous avons décidé qu'il était plus utile pour les créateurs d'un paquetage de pouvoir utiliser un souligné en début de nom pour indiquer des identificateurs semi-privés destinés uniquement à l'usage interne de ce paquetage. (Des variables réellement privées peuvent être déclarées comme variables lexicales dans la portée d'un fichier, mais cela marche le mieux quand un paquetage et un module ont

6. Pour éviter un autre sujet de confusion éventuelle, dans le cas d'un nom de variable comme `$main::chaude`, nous utilisons le terme « identificateur » pour parler de `main` et de `chaude`, mais pas de `main::chaude`, que nous nommons plutôt nom de variable, car les identificateurs ne peuvent pas contenir de deux-points.

une relation bijective, ce qui est courant mais pas obligatoire.)

Le hachage `%SIG` (qui sert à intercepter les signaux, voir le chapitre 16, *Communication interprocessus*) est également spécial. Si vous définissez un gestionnaire de signal en tant que chaîne, il est sous-entendu qu'il fait référence à une fonction du paquetage `main`, sauf si le nom d'un autre paquetage est explicitement utilisé. Qualifiez complètement le nom d'un gestionnaire de signal si vous voulez utiliser un paquetage spécifique, ou évitez entièrement les chaînes en affectant plutôt un typeglob ou une référence à une fonction :

```
$SIG{QUIT} = "Pack::attrape_quit";     # nom de gestionnaire qualifié
$SIG{QUIT} = "attrape_quit";      # sous-entend "main::attrape_quit"
$SIG{QUIT} = *attrape_quit;       # impose la fonction du paquetage courant
$SIG{QUIT} = \&attrape_quit;      # impose la fonction du paquetage courant
$SIG{QUIT} = sub { print "Attrapé SIGQUIT\n" };    # fonction anonyme
```

La notion de « paquetage courant » est un concept concernant à la fois la compilation et l'exécution. La plupart des vérifications de nom de variable se font à la compilation, mais les vérifications à l'exécution se font lorsque des références symboliques sont déréférencées et quand de nouveaux bouts de code sont analysés avec `eval`. En particulier, quand vous appliquez `eval` à une chaîne, Perl sait dans quel paquetage `eval` a été invoqué et propage ce paquetage vers l'intérieur en évaluant la chaîne. (Bien sûr, vous pouvez toujours passer à un autre paquetage à l'intérieur de la chaîne `eval`, puisqu'une chaîne `eval` compte pour un bloc, tout comme un fichier chargé avec un `do`, un `require`, ou un `use`.)

D'un autre côté, si `eval` veut déterminer dans quel paquetage il se trouve, le symbole spécial `__PACKAGE__` contient le nom du paquetage courant. Puisque vous pouvez le traiter comme une chaîne, vous pourriez l'utiliser dans une référence symbolique pour accéder à une variable de paquetage. Mais si vous faites cela, il y a des chances que vous ayez plutôt intérêt à déclarer la variable avec `our`, de sorte qu'on puisse y accéder comme si c'était une variable lexicale.

Tables de symboles

Les éléments contenus dans un paquetage sont dans leur ensemble appelés *table de symboles*. Les tables de symboles sont rangées dans un hachage dont le nom est le même que celui du paquetage, avec un double deux-points ajouté à la fin. Ainsi le nom de la table de symboles principale, `main`, est `%main::`. Puisque `main` se trouve aussi être le paquetage par défaut, Perl fournit `%::`comme abbréviation pour `%main::`.

De même, la table de symboles du paquetage `Rouge::Bleu` est appelé `%Rouge::Bleu::`. Il se trouve que la table de symboles `main` contient toutes les autres tables de symboles de haut niveau, elle-même incluse, donc `%Rouge::Bleu::` est aussi `%main::Rouge::Bleu::`.

Quand nous disons que la table de symboles « contient » une autre table de symboles, nous voulons dire qu'elle contient une référence à une autre table de symboles. Puisque `main` est le paquetage de plus haut niveau, il contient une référence à lui-même, ainsi `%main::` est pareil que `%main::main::`, et `%main::main::main::`, et ainsi de suite, ad infinitum. Il est important de prévoir ce cas spécial si vous écrivez du code qui parcourt toutes les tables de symboles.

À l'intérieur du hachage d'une table de symboles, chaque couple clef/valeur fait correspondre à un nom de variable sa valeur. Les clefs sont les identificateurs de symbole et les valeurs les typeglobs correspondants. Donc quand vous utilisez la notation typeglob **NOM*, vous accédez en réalité à une valeur dans le hachage qui contient la table de symboles du paquetage courant. En fait, les instructions suivantes ont (quasiment) le même effet :

```
*sym = *main::variable;
*sym = $main::{"variable"};
```

La première est plus efficace car on accède à la table de symboles `main` au moment de la compilation. Elle créera aussi un nouveau typeglob de ce nom s'il n'en existait pas précédemment, alors que la deuxième version ne le fera pas.

Puisqu'un paquetage est un hachage, vous pouvez rechercher les clefs du paquetage et récupérer toutes les variables du paquetage. Puisque les valeurs du hachage sont des typeglobs, vous pouvez les déréférencer de diverses façons. Essayez ceci :

```
foreach $nomsym (sort keys %main::) {
    local *sym = $main::{$nomsym};
    print "\$$nomsym est défini\n" if defined $sym;
    print "\@$nomsym is non nul\n" if         @sym;
    print "\%$nomsym is non nul\n" if         %sym;
}
```

Puisque tous les paquetages sont accessibles (directement ou indirectement) par le paquetage `main`, vous pouvez écrire du code Perl pour visiter chaque variable de paquetage de votre programme. C'est précisément ce que fait le débogueur Perl lorsque vous lui demandez d'afficher toutes vos variables avec la commande `V`. Notez que si vous faites cela, vous ne verrez pas les variables déclarées avec `my`, puisqu'elles sont indépendantes des paquetages, bien que vous verrez les variables déclarées avec `our`. Voir le chapitre 20, *Le débogueur Perl*.

Précédemment nous avons dit que seuls les identificateurs étaient rangés dans des paquetages autres que `main`. C'était un tout petit mensonge : vous pouvez utiliser la chaîne que vous voulez comme clef dans le hachage de la table de symboles — mais ça ne serait pas du Perl valide si vous essayiez d'utiliser un non-identificateur directement :

```
$!@#$%           = 0;         # FAUX, erreur de syntaxe.
${'!@#$%'}       = 1;         # Ok, mais non qualifié.

${'main::!@#$%'} = 2;         # On peut qualifier dans la chaîne
print ${ $main::{'!@#$%'} }   # Ok, imprime  2!
```

Les typeglobs servent essentiellement d'alias :

```
*dick = *richard;
```

rend les variables, fonctions, formats, et handles de fichiers et répertoires accessibles par l'identificateur `richard` également accessibles par l'identificateur `dick`. Si vous ne souhaitez affecter un alias qu'à une seule variable ou fonction, affectez plutôt la référence :

```
*dick = \$richard;
```

Ce qui fait que `$richard` et `$dick` sont littéralement la même variable, mais que `@richard` et `@dick` restent des tableaux différents. Rusé, non ?

C'est ainsi que fonctionne le paquetage `Exporter` lorsque l'on importe des variables d'un paquetage vers un autre. Par exemple :

```
*UnPack::dick = \&AutrePack::richard;
```

importe la fonction `&richard` du paquetage `AutrePack` dans le paquetage `UnPack`, le rendant disponible comme fonction `&dick`. (Le module `Exporter` sera décrit dans le prochain chapitre.) Si vous précédez l'affectation avec un `local`, l'alias ne durera que le temps de la portée dynamique courante.

Ce mécanisme peut être utilisé pour récupérer une référence à partir d'une fonction, rendant le référent disponible avec le type de données approprié :

```
*unites = renseigner() ;   # Affecter \%nouveau_hachage au typeglob
print $unites{kg};         # Affiche 70; pas nécessaire de déréférencer!

sub renseigner {
    my %nouveau_hachage = (km => 10, kg => 70);
    return \%nouveau_hachage;
}
```

De même, vous pouvez passer une référence à une fonction et l'utiliser sans la déréférencer :

```
%unites = (kilometres => 6, kilos => 11);
faireleplein( \%unites );    # Passer une référence
print $unites{litres};       # Affiche  4

sub faireleplein {
    local *hashsym = shift;  # Affecter \%unites au typeglob
    $hashsym{litres} = 4;    # Modifie %unites; pas nécessaire
                             # de déréférencer !
}
```

Ce sont des moyens astucieux de manipuler des références quand vous ne voulez pas devoir les déréférencer explicitement. Notez que les deux techniques ne fonctionnent qu'avec des variables de paquetage. Elles ne marcheraient pas si on avait déclaré `%unites` avec `my`.

Les tables de symboles s'utilisent aussi pour créer des scalaires « constants » :

```
*PI = \3.14159265358979;
```

Il vous est maintenant impossible d'altérer `$PI`, ce qui probablement une bonne chose, somme toute. Ceci n'est pas pareil qu'une fonction constante, qui est optimisée à la compilation. Une fonction constante est une fonction dont le prototype indique qu'elle ne prend pas de paramètres et qu'elle retourne une expression constante ; voir pour plus de détails la section *Substitution en ligne de fonctions constantes*, chapitre 6, *Sous-programmes*. Le pragma `use constant` est un raccourci commode :

```
use constant PI => 3.14159;
```

Sous le capot, ceci utilise la case « fonction » de `*PI`, plutôt que la case « scalaire » utilisée ci-dessus. L'expression suivante, plus compacte (mais moins lisible), est équivalente :

```
*PI = sub () { 3.14159 };
```

En tout cas c'est un idiome pratique à connaître. Affecter un `sub {}` à un typeglob permet de donner un nom à une fonction anonyme lors de l'exécution.

Affecter une référence à un typeglob à un autre typeglob (`*sym = \*oldvar`) revient à affecter the typeglob en entier, car Perl déréférence automatiquement la référence au typeglob à votre place. Lorsque vous affectez simplement une chaîne à une typeglob, vous obtenez le typeglob entier nommé par cette chaîne, car Perl va chercher la chaîne dans la table de symboles courante. Les instructions suivantes sont toutes équivalentes entre elles, bien que les deux premières sont calculées à la compilation, alors que les deux dernières le sont à l'exécution :

```
*sym =    *anciennevar;
*sym =   \*anciennevar;    # déréférence auto
*sym = \*{"anciennevar"};  # recherche explicite dans la table de symboles
*sym =    "anciennevar";   # recherche implicite dans la table de symboles
```

Lorsque vous faites l'une des affectations suivantes, vous ne faites que remplacer l'une des références à l'intérieur du typeglob :

```
*sym = \$frodon;
*sym = \@sam;
*sym = \%merry;
*sym = \&pippin;
```

D'un autre point de vue, le typeglob lui-même peut être considéré comme une espèce de hachage, incluant des entrées pour les différents types de variables. Dans ce cas, les clefs sont fixées, puisqu'un typeglob contient exactement un scalaire, un tableau, un hachage, et ainsi de suite. Mais vous pouvez extraire les références individuelles, comme ceci :

```
*pkg::sym{SCALAR}   # pareil que \$pkg::sym
*pkg::sym{ARRAY}    # pareil que \@pkg::sym
*pkg::sym{HASH}     # pareil que \%pkg::sym
*pkg::sym{CODE}     # pareil que \&pkg::sym
*pkg::sym{GLOB}     # pareil que \*pkg::sym
*pkg::sym{IO}       # handle de fichier/rép interne, pas d'équivalent direct
*pkg::sym{NAME}     # "sym" (pas une référence)
*pkg::sym{PACKAGE}  # "pkg" (pas une référence)
```

Vous pouvez dire `*foo{PACKAGE}` et `*foo{NAME}` pour déterminer de quel nom et et de quel paquetage provient l'entrée `*foo` d'une table de symboles. Ceci peut être utile dans une fonction à laquelle des typeglobs sont passés en paramètres :

```
sub identifier_typeglob {
    my $glob = shift;
    print 'Vous m'avez donné ', *{$glob}{PACKAGE}, '::', *{$glob}{NAME},
"\n"
}

identifier_typeglob(*foo);
identifier_typeglob(*bar::glarch);
```

Ceci affiche :

```
Vous m'avez donné main::foo
Vous m'avez donné bar::glarch
```

La notation *foo{*TRUC*} peut être utilisée pour obtenir des références aux éléments individuels de *foo. Voir pour plus de détails la section *Références de table de symboles*, chapitre 8.

Cette syntaxe est principalement utilisée pour accéder au handle de fichier ou de répertoire interne, car les autres références internes sont déjà accessibles d'autres façons. (L'ancien *foo{FILEHANDLE} reste supporté et signifie *foo{IO}, mais ne laissez pas son nom vous faire croire qu'il peut distinguer entre des handles de fichier et de répertoire.) Nous avons pensé généraliser cela parce que quelque part, c'est joli. Quelque part. Vous n'avez probablement pas à vous souvenir de tout cela sauf si vous avez l'intention d'écrire un autre débogueur de Perl.

Autochargement

Normalement, vous ne pouvez pas appeler une fonction non définie. Cependant, s'il y a une fonction nommée AUTOLOAD dans le paquetage de la fonction non définie (ou dans le cas d'une méthode d'objet, dans le paquetage de l'une quelconque des classes de base de l'objet), alors la fonction AUTOLOAD est appelée avec les mêmes paramètres qui seraient passés à la fonction d'origine. Vous pouvez définir la fonction AUTOLOAD pour qu'elle retourne des valeurs exactement comme une fonction normale, ou vous pouvez définir la fonction qui n'existait pas et ensuite l'appeler comme si elle avait existé tout du long.

Le nom pleinement qualifié de la fonction d'origine apparaît magiquement dans la variable globale au paquetage $AUTOLOAD, dans le même paquetage que la fonction AUTOLOAD. Voici un exemple simple qui vous avertit gentiment lors des appels de fonctions non définies, au lieu d'arrêter le programme.

```
sub AUTOLOAD {
    our $AUTOLOAD;
    warn "la tentative d'appeler $AUTOLOAD a échoué.\n";
}

blarg(10);              # notre $AUTOLOAD aura la valeur to main::blarg
print "Encore vivant!\n"
```

Ou vous pouvez retourner une valeur pour le compte de la fonction non définie :

```
sub AUTOLOAD {
    our $AUTOLOAD;
    return "Je vois $AUTOLOAD(@_)\n";
}

print blarg(20);        # affiche: Je vois main::blarg(20)
```

Votre fonction AUTOLOAD pourrait charger une définition de la fonction non définie en utilisant eval ou require, ou utiliser l'astuce d'affectation au glob discutée précédemment, puis exécuter cette fonction en utilisant une forme spéciale du goto qui peut effacer le « stack frame » de la fonction sans laisser de trace. Ici nous définissons la fonction en affectant une fermeture au glob :

```
sub AUTOLOAD {
    my $nom = our $AUTOLOAD;
```

```
    *$AUTOLOAD = sub { print "Je vois $nom(@_)\n" };
    goto &$AUTOLOAD;    # Redémarrer la nouvelle fonction
}
blarg(30);              # affiche : je vois main::blarg(30)
glarb(40);              # affiche : je vois main::glarb(40)
blarg(50);              # affiche : je vois main::blarg(50)
```

Le module standard `AutoSplit` est utilisé par les créateurs de modules pour les aider à diviser leurs modules en plusieurs fichiers (avec des noms de fichier se terminant avec *.al*), chacun contenant une fonction. Les fichiers sont placés dans le répertoire *auto/* de la bibliothèque Perl de votre système, après quoi les fichiers peuvent être autochargés par le module standard `AutoLoader`.

Une approche similaire est utilisée par le module `SelfLoader`, mis à part qu'il autocharge des fonctions à partir de la zone `DATA` du fichier lui-même, ce qui est moins efficace à certains égards, et plus efficace à d'autres. L'autochargement de fonctions Perl par `AutoLoader` et `SelfLoader` est analoque au chargement dynamique de fonctions C compilées par `DynaLoader`, mais l'autochargement est fait avec la granularité d'un appel de fonction, alors que le chargement dynamique est fait avec la granularité d'un module entier, et liera en général un grand nombre de fonction C ou C++ à la fois. (Notez que beaucoup de programmeurs Perl se débrouillent très bien sans les modules `AutoSplit`, `AutoLoader`, `SelfLoader`, or `DynaLoader`. Il faut juste que vous sachiez qu'ils sont là, au cas où vous ne pourriez *pas* vous débrouiller sans eux.)

On peut s'amuser avec des fonctions `AUTOLOAD` qui servent d'emballage pour d'autres interfaces. Par exemple, faisons semblant qu'une fonction qui n'est pas définie doive juste appeler `system` avec ses paramètres. Tout ce que vous auriez à faire est ceci :

```
sub AUTOLOAD {
    my $programme = our $AUTOLOAD;
    $programme =~ s/.*::/;  # enlever le nom du paquetage
    system($programme, @_);
}
```

(Félicitations, vous venez d'implémenter une forme rudimentaire du module `Shell` qui vient en standard avec Perl.) Vous pouvez appeler votre autochargeur (sur Unix) ainsi :

```
date();
who('am', 'i');
ls('-l');
echo("Ga bu zo meu...");
```

En fait, si vous prédéclarez les fonctions que vous voulez appeler de cette façon, vous pouvez procéder comme si elles étaient pré-construites dans le langage et omettre les parenthèses lors de l'appel :

```
sub date (;$$);       # Permettre de zéro à deux paramètres.
sub who (;$$$$);      # Permettre de zéro à quatre paramètres.
sub ls;               # Permettre un nombre quelconque de paramètres.
sub echo ($@);        # Permettre au moins un paramètre.

date;
who "am", "i";
ls "-l";
echo "C'est tout pour aujourd'hui !";
```

11

Modules

Le module est l'unité fondamentale de réutilisation de Perl. À bien le regarder, un module n'est qu'un paquetage défini dans un fichier de même nom (avec *.pm* à la fin). Dans ce chapitre, nous allons explorer comment utiliser les modules des autres et créer les siens.

Perl inclut en standard un grand nombre de modules, que vous pouvez trouver dans le répertoire *lib* de votre distribution Perl. Beaucoup de ces modules sont décrits au chapitre 32, *Modules standards*, et au chapitre 31, *Modules de pragmas*. Tous les modules standard ont aussi une documentation détaillée en ligne, qui peut (scandale !) être plus à jour que ce livre. Essayez la commande *perldoc* si votre commande *man* ne fonctionne pas.

Le *Comprehensive Perl Archive Network* (CPAN) contient un dépôt de modules alimenté par la communauté Perl mondiale, et est présenté au chapitre 22, *CPAN*. Voir aussi *http://www.cpan.org*.

Utilisation des modules

Il y a deux sortes de modules : traditionnel et orienté objet. Les modules traditionnels définissent des fonctions et des variables destinées à être importées et utilisées par un programme qui y fait appel. Les modules orientés objet fonctionnent comme des définitions de classe auxquelles on accède par des appels de méthodes, décrites au chapitre 12, *Objets*. Certains modules font les deux.

Les modules Perl sont habituellement inclus dans votre programme en écrivant :

```
use MODULE LISTE;
```

ou simplement :

```
use MODULE;
```

MODULE doit être un identificateur qui nomme le paquetage et le fichier du module. (Les descriptions de syntaxe ne sont données ici qu'à titre indicatif, car la syntaxe complète de l'instruction `use` est détaillée au chapitre 29, *Fonctions*.)

L'instruction `use` précharge *`MODULE`* au moment de la compilation, puis importe les symboles que vous avez demandés pour qu'ils soient disponibles pour le reste de la compilation. Si vous ne fournissez pas une *`LISTE`* des symboles que vous voulez, les symboles nommés dans le tableau `@EXPORT` interne au module sont utilisés — en supposant que vous utilisez le module `Exporter`, décrit plus loin dans ce chapitre sous « Espace privé de module et Exporter ». (Si vous fournissez une *`LISTE`*, tous vos symboles doivent être mentionnés, soit dans le tableau `@EXPORT`, soit dans le tableau `@EXPORT_OK`, faute de quoi une erreur en résultera.)

Puisque les modules utilisent `Exporter` pour importer des symboles dans le paquetage courant, vous pouvez utiliser les symboles d'un module sans les qualifier avec le nom du paquetage :

```
use Fred;       # Si Fred.pm contient @EXPORT = qw(pierrafeu)
pierrafeu();    # ...ceci appelle Fred::pierrafeu().
```

Tous les fichiers de module Perl ont l'extension *.pm*. Ceci (ainsi que les doubles apostrophes) est pris en compte par `use` ainsi que par `require` afin que vous n'ayez pas à écrire "*`MODULE`*`.pm`". Cette convention permet de différencier les nouveaux modules des bibliothèques *.pl* et *.ph* utilisées dans d'anciennes versions de Perl. Elle définit aussi *`MODULE`* comme nom officiel de module, ce qui aide l'analyseur dans certaines situations ambiguës. Tout double deux-points dans le nom de module est traduit comme le séparateur de répertoires de votre système, de sorte que si votre module a pour nom `Rouge::Bleu::Vert`, Perl pourrait le chercher sous *Rouge/Bleu/Vert.pm*.

Perl ira chercher vos modules dans tous les répertoires listés dans le tableau `@INC`. Comme `use` charge les modules au moment de la compilation, toutes modifications au tableau `@INC` doivent aussi survenir au moment de la compilation. Vous pouvez faire cela avec le pragma `lib`, décrit au chapitre 31 ou avec un bloc `BEGIN`. Une fois qu'un module est inclus, un couple clef/valeur sera ajouté au hachage `%INC`. La clef sera le nom de fichier du module (`Rouge/Bleu/Vert.pm` dans notre exemple) et la valeur sera le chemin complet, qui pourrait être quelque chose comme `C:/perl/site/lib/Rouge/Bleu/Vert.pm` pour un fichier convenablement installé sur un système Windows.

Les noms de module doivent commencer avec une lettre majuscule, sauf s'ils fonctionnent comme pragmas. Les pragmas sont essentiellement des directives de compilation (des conseils pour le compilateur), donc nous réservons les noms de pragma en minuscules pour un usage ultérieur.

Lorsque vous utilisez un module avec `use`, tout le code du module est exécuté, tout comme il le serait avec un `require` ordinaire. S'il vous est égal que le module soit pris en compte à la compilation ou à l'exécution, vous pouvez juste dire :

```
require MODULE;
```

Cependant, en général on préfère `use` plutôt que `require`, car il recherche les modules à la compilation, donc vous découvrez les erreurs éventuelles plus tôt.

Ces deux instructions font presque la même chose :

```
require MODULE;
require "MODULE.pm";
```

Elles diffèrent néanmoins de deux façons. Dans la première instruction, `require` traduit

tout double deux-points en séparateur de répertoire de votre système, tout comme le fait `use`. La deuxième instruction ne fait pas de traduction, vous obligeant à spécifier explicitement le chemin de votre module, ce qui est moins portable. L'autre différence est que le premier `require` indique au compilateur que les expressions utilisant la notation objet indirecte avec « *MODULE* » (comme `$ob = purge`*`MODULE`*) sont des appels de méthode, pas des appels de fonction. (Si, ça peut réellement faire une différence, s'il y a une définition `purge` dans votre module qui entre en conflit avec une autre.)

Comme la déclaration `use` et la déclaration apparentée `no` impliquent un bloc `BEGIN`, le compilateur charge le module (et exécute son éventuel code d'initialisation) dès qu'il rencontre cette déclaration, *avant* de compiler le reste du fichier. C'est ainsi que les pragmas peuvent modifier le comportement du compilateur, et que les modules peuvent déclarer des fonctions qui sont par la suite visibles comme opérateurs de liste pour le reste de la compilation. Ceci ne marchera pas si vous utilisez un `require` à la place d'un `use`. La seule raison d'utiliser un `require` est d'avoir deux modules dont chacun nécessite une fonction de l'autre. (Et nous ne sommes pas sûrs que ce soit une bonne raison.)

Les modules Perl chargent toujours un fichier *.pm*, mais ce fichier peut à son tour charger des fichiers associés, comme des bibliothèques C ou C++ liées dynamiquement, ou des définitions de fonction Perl autochargées. Dans ce cas, les complications supplémentaires seront entièrement invisibles pour l'utilisateur du module. C'est la responsabilité du fichier *.pm* de charger (ou de s'arranger pour autocharger) toute fonctionnalité supplémentaire. Il se trouve que le module `POSIX` fait du chargement dynamique et de l'autochargement, mais l'utilisateur peut simplement dire :

```
use POSIX;
```

pour avoir accès à toutes les fonctions et variables exportées.

Création de modules

Précédemment, nous avons dit qu'un module a deux façons de mettre son interface à la disposition de votre programme : en exportant des symboles ou en permettant des appels de fonction. Nous vous montrerons un exemple de la première technique ici ; la deuxième technique est utilisée pour les modules orientés objet, et est décrite dans le prochain chapitre. (Les modules orientés objet ne devraient rien exporter, car les méthodes objet ont pour principe que Perl les trouve automatiquement à votre place en se basant sur le type de l'objet.)

Pour construire un module appelé `Bestiaire`, créez un fichier appelé *Bestiaire.pm* qui ressemble à ceci :

```
package      Bestiaire;
require      Exporter;

our @ISA       = qw(Exporter);
our @EXPORT    = qw(dromadaire);  # Symboles à exporter par défaut
our @EXPORT_OK = qw($poids);      # Symboles à exporter à la demande
our $VERSION   = 1.00;            # Numéro de version

### Mettez vos variables et vos fonctions ici
```

```
sub dromadaire { print "Chameau à une seule bosse" }

$poids = 1024;

1;
```

Maintenant un programme peut dire `use Bestiaire` pour pouvoir accéder à la fonction `dromadaire` (mais pas à la variable `$poids`), et `use Bestiaire qw(dromadaire $poids)` pour accéder à la fois à la fonction et à la variable.

Vous pouvez également créer des modules qui chargent dynamiquement du code écrit en C. Voir le chapitre 21, *Mécanismes internes et accès externes*, pour plus de détails.

Espace privé de module et Exporter

Perl ne patrouille pas automatiquement le long de frontières entre le privé et le public dans ses modules — ontrairement à des langages comme C++, Java et Ada, Perl n'est pas obsédé par le droit à la vie privée. Un module Perl préfèrerait que vous restiez hors de son salon parce que vous n'y avez pas été invité, pas parce qu'il a un fusil.

Le module et son utilisateur ont un contrat, dont une partie se base sur la loi coutumière, et dont l'autre est écrite. Une partie du contrat de loi coutumière est qu'un module doit s'abstenir de modifier tout espace de noms qu'on ne lui a pas demandé de modifier. Le contrat écrit du module (c'est-à-dire la documentation) peut prendre d'autres dispositions. Mais vous savez sans doute, après avoir lu le contrat, que quand vous dites `use RedefinirLeMonde` vous redéfinissez le monde, et vous êtes prêt à en risquer les conséquences. La façon la plus commune de redéfinir des mondes est d'utiliser le module `Exporter`. Comme nous le verrons plus loin dans ce chapitre, vous pouvez même redéfinir des fonctions Perl de base avec ce module.

Lorsque vous faites appel à un module avec `use`, le module met habituellement certaines fonctions et variables à la disposition de votre programme, ou plus précisément du paquetage courant de votre programme. Cet acte d'exportation de symboles du module (et donc d'importation dans votre programme) est parfois appelé *pollution* de votre espace de noms. La plupart des modules utilisent `Exporter` pour faire cela. C'est pourquoi la plupart des modules disent quelque chose comme ceci vers le début :

```
require Exporter;
our @ISA = ("Exporter");
```

Ces deux lignes font hériter au module de la classe `Exporter`. L'héritage est décrit au prochain chapitre, mais tout ce que vous avez besoin de savoir c'est que notre module `Bestiaire` peut maintenant exporter des symboles dans d'autres paquetages avec des lignes comme celle-ci :

```
our @EXPORT    = qw($chameau %loup belier);    # Exporter par défaut
our @EXPORT_OK = qw(leopard @lama $emeu);      # Exporter à la demande
our %EXPORT_TAGS = (                           # Exporter en groupe
                    camelides => [qw($chameau @lama)],
                    bestiaux  => [qw(belier $chameau %loup)],
                   );
```

Du point de vue du module qui exporte, le tableau `@EXPORT` contient les noms de varia-

bles et de fonctions à exporter par défaut : ce qu'obtient votre programme lorsqu'il dit `use Bestiaire`. Les variables et les fonctions dans `@EXPORT_OK` ne sont exportées que lorsque le programme en fait explicitement la demande dans l'instruction `use`. Enfin, les couples clef/valeur dans `%EXPORT_TAGS` permettent au programme d'inclure des groupes de symboles spécifiques, listés dans `@EXPORT` et `@EXPORT_OK`.

Du point de vue du module qui importe, l'instruction `use` spécifie la liste des symboles à importer, un groupe nommé `%EXPORT_TAGS`, un motif de symboles, ou rien du tout, auquel cas les symboles dans `@EXPORT` seront importés du module dans votre programme.

Vous pouvez inclure l'une quelconque de ces instructions pour importer des symboles du module `Bestiaire` :

```
use Bestiaire;                  # Importer les symboles de @EXPORT
use Bestiaire ();               # Ne rien importer
use Bestiaire qw(belier @lama); # Importer la fonction belier et le
                                # tableau @lama
use Bestiaire qw(:camelides);   # Importer $chameau et @lama
use Bestiaire qw(:DEFAULT);     # Importer les symboles de @EXPORT
use Bestiaire qw(/am/);         # Importer $chameau et @lama
use Bestiaire qw(/^\$/);        # Importer tous les scalaires
use Bestiaire qw(:bestiaux !belier);
                                # Importer bestiaux, mais exclure belier
use Bestiaire qw(:bestiaux !:camelides);
                                # Importer bestiaux, mais pas camelides
```

Omettre un symbole des listes d'exportation (ou le supprimer explicitement de la liste d'import avec le point d'exclamation) ne le rend pas inaccessible pour le programme qui utilise le module. Le programme pourra toujours accéder au contenu du paquetage du module en le qualifiant avec le nom du paquetage, comme par exemple `%Bestiaire::gecko`. (Comme les variables lexicales n'appartiennent pas au paquetage, l'espace privé est possible : voir *Méthodes privées* au chapitre suivant.)

Vous pouvez dire `BEGIN { $Exporter::Verbose=1 }` pour voir comment sont gérées les spécifications et pour voir ce qui est réellement importé dans votre paquetage.

`Exporter` est lui-même un module Perl, et si cela vous intéresse vous pouvez voir les astuces typeglob qui sont utilisées pour exporter des symboles d'un paquetage à un autre. À l'intérieur du module `Exporter`, la fonction essentielle, appelée `import`, crée les alias nécessaires pour qu'un symbole d'un paquetage soit visible dans un autre paquetage. En fait, l'instruction `use Bestiaire`*`LISTE`* est précisément équivalente à :

```
BEGIN {
    require Bestiaire;
    import Bestiaire LISTE;
}
```

Cela veut dire que vos modules ne sont pas obligés d'utiliser `Exporter`. Un module peut faire ce qui lui plaît, puisque `use` ne fait qu'appeler la méthode `import` du module, et vous pouvez définir cette méthode pour qu'elle fasse ce que vous voulez.

Exporter sans utiliser la méthode import de Exporter

Le module `Exporter` définit une méthode appelée `export_to_level` pour les situations où vous ne pouvez pas directement appeler la méthode `import` de `Exporter`. La méthode de `export_to_level` est invoquée comme ceci :

```
MODULE->export_to_level($ou_exporter, @quoi_exporter);
```

L'entier `$ou_exporter` indique à quelle hauteur dans la pile d'appel exporter vos symboles, et le tableau `$quoi_exporter` liste les symboles à exporter (habituellement `@_`).

Par exemple, supposons que notre `Bestiaire` contienne sa propre fonction `import` :

```
package Bestiaire;
@ISA = qw(Exporter);
@EXPORT_OK = qw ($zoo);

sub import {
    $Bestiaire::zoo = "ménagerie";
}
```

La présence de cette fonction `import` empêche d'hériter de la fonction `import` de `Exporter`. Si vous vouliez qu'après avoir affecté à `$Bestiaire::zoo`, la fonction `import` de `Bestiaire` se comporte exactement comme la fonction `import` de `Exporter`, vous la définiriez de la façon suivante :

```
sub import {
    $Bestiaire::zoo = "ménagerie";
    Bestiaire->export_to_level(1, @_);
}
```

Ceci exporte des symboles vers le paquetage situé un niveau « au-dessus » du paquetage courant, c'est-à-dire vers tout programme ou module qui utilise `Bestiaire`.

Vérification de version

Si votre module définit une variable `$VERSION`, un programme qui utilise votre module peut s'assurer que le module est suffisamment récent. Par exemple :

```
use Bestiaire 3.14;   # Le Bestiaire doit être de version 3.14 ou plus
use Bestiaire v1.0.4; # Le Bestiaire doit être de version 1.0.4 ou plus
```

Ces instructions sont converties en appels à `Bestiaire->require_version`, dont hérite ensuite votre module.

Gestion de symboles inconnus

Dans certaines situations, vous voudrez peut-être *empêcher* l'exportation de certains symboles. Typiquement, c'est le cas de certains modules qui ont des fonctions ou des constantes qui n'ont pas de sens sur certains systèmes. Vous pouvez empêcher `Exporter` d'exporter ces symboles en les plaçant dans le tableau `@EXPORT_FAIL`.

Si un programme tente d'importer l'un de ces symboles, `Exporter` permet au module de gèrer la situation avant de produire une erreur. Il le fait en appelant une méthode `export_fail` avec une liste des symboles qui ont échoué, que vous pouvez définir ainsi (en supposant que votre module utilise le module `Carp`) :

```
sub export_fail {
    my $class = shift;
    carp "Désolé, ces symboles ne sont pas disponibles : @_";
    return @_;
}
```

`Exporter` fournit une méthode `export_fail` par défaut, qui retourne simplement la liste inchangée et fait échouer le `use` en levant une exception pour chaque symbole. Si `export_fail` retourne une liste vide, aucune erreur n'est enregistrée et tous les symboles demandés sont exportés.

Fonctions de gestion des étiquettes

Comme les symboles listés dans `%EXPORT_TAGS` doivent également apparaître, soit dans `@EXPORT`, soit dans `@EXPORT_OK`, `Exporter` fournit deux fonctions qui vous permettent d'ajouter ces ensembles de symboles étiquetés.

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');     # ajouter aa, bb et cc à @EXPORT
Exporter::export_ok_tags('bar');  # ajouter aa, cc et dd à @EXPORT_OK
```

C'est une erreur de spécifier des noms qui ne sont pas des étiquettes.

Supplanter des fonctions internes

De nombreuses fonctions internes peuvent être *supplantées*, bien que (comme pour faire des trous dans vos murs) vous ne devriez faire cela que rarement et pour de bonnes raisons. Typiquement, ce serait fait par un paquetage qui tente d'émuler des fonctions internes manquantes sur un système autre qu'Unix. (Ne confondez pas supplanter avec *surcharger*, qui ajoute des significations orientées objet supplémentaires aux opérateurs internes, mais ne supplante pas grand chose. Le module `overload` dans le chapitre 13, *Surcharge*, donne plus de détails sur ce sujet.)

On ne peut supplanter qu'en important le nom d'un module ; la prédéclaration ordinaire ne suffit pas. Pour être parfaitement francs, c'est l'affectation d'une référence de code à un typeglob qui permet à Perl de supplanter, comme dans `*open = \&myopen`. De plus, l'affectation doit survenir dans un autre paquetage ; ceci rend intentionnellement difficile de supplanter accidentellement par alias de typeglob. Malgré tout, si vous voulez vraiment supplanter vous-même, ne désespérez pas, car le pragma `subs` vout permet de prédéclarer des fonctions avec la syntaxe d'importation, et ces noms supplantent alors ceux qui existent en interne :

```
use subs qw(chdir chroot chmod chown);
chdir $quelque_part
sub chdir { ... }
```

En général, les modules ne doivent pas exporter des noms internes comme `open` ou `chdir` comme partie de leur liste `@EXPORT` par défaut, puisque ces noms pourraient se glisser dans l'espace de noms de quelqu'un d'autre et en changer le contenu de façon inattendue. Si au lieu de cela le module inclut le nom dans la liste `@EXPORT_OK`, les importateurs seront obligés de demander explicitement que le nom interne soit supplanté, évitant ainsi les surprises.

La version d'origine des fonctions internes est toujours accessible *via* le pseudo-paquetage `CORE`. En conséquence, `CORE::chdir` sera toujours la version compilée d'origine dans Perl, même si le mot-clef `chdir` a été supplanté.

Enfin, presque toujours. Le mécanisme décrit ci-dessus pour supplanter les fonctions internes est restreint, très délibérément, au paquetage qui fait la demande d'importation. Mais il existe un mécanisme plus radical que vous pouvez utiliser lorsque vous souhaitez supplanter une fonction interne partout, sans prendre en compte les frontières d'espaces de noms. Ceci est accompli en définissant la fonction dans le pseudo-paquetage `CORE::GLOBAL`. Voyez ci-dessous un exemple qui supplante l'opérateur `glob` avec quelque chose qui comprend les expressions rationnelles. (Notez que cet exemple n'implémente pas tout ce qui est nécessaire pour supplanter proprement le `glob` interne de Perl, qui se comporte différemment selon qu'il apparaît dans un contexte scalaire ou un contexte de liste. En vérité, de nombreuses fonctions internes de Perl ont de tels comportements sensibles au contexte, et tout fonction visant à les correctement supplanter une fonction interne doit gérer adéquatement ces comportements. Pour voir un exemple complètement fonctionnel où glob est supplanté, étudiez le module `File::Glob` inclus avec Perl.) En tout cas, voici la version anti-sociale :

```
*CORE::GLOBAL::glob = sub {
    my $motif = shift;
    my @trouve;
    local *D;
    if (opendir D, '.') {
        @trouve = grep /$motif/, readdir D;
        closedir D;
    }
    return @trouve;
}

package Quelconque;

print <^[a-z_]+\.pm\$>;  # montrer tous les pragmas dans le répertoire
courant
```

En supplantant `glob` globalement, ceci impose un nouveau (et subversif) comportement pour l'opérateur `glob` dans *tous* les espaces de noms, sans la connaissance ou la coopération des modules popriétaires de ces espaces de noms. Bien sûr, s'il faut faire cela, il faut le faire avec la plus extrême prudence. Et probablement il ne faut pas le faire.

Nous supplantons avec le principe suivant : c'est sympa d'être important, mais c'est plus important d'être sympa.

12

Objets

Pour commencer, vous devez comprendre les paquetages et les modules ; voir le chapitre 10, *Paquetages*, et le chapitre 11, *Modules*. Vous devez aussi connaître les références et les structures de données ; voir le chapitre 8, *Références*, et le chapitre 9, *Structures de données*. Comme il est utile de s'y connaître un peu en programmation orientée objet, nous vous donnerons un petit cours de vocabulaire orienté objet dans la section suivante.

Bref rappel de vocabulaire orienté objet

Un *objet* est une structure de données munie d'une collection de comportements. Nous dirons parfois qu'un objet agit directement sur la base de comportements, allant parfois jusqu'à l'anthropomorphisme. Par exemple, nous dirons qu'un rectangle « sait » comment s'afficher à l'écran ou qu'il « sait » comment calculer sa propre surface.

Un objet exhibe des comportements en étant une *instance* d'une *classe*. La classe définit des *méthodes* : des comportements qui s'appliquent à la classe et à ses instances. Lorsqu'il faut faire la distinction, nous appelons *méthode d'instance* une méthode qui ne s'applique qu'à un objet donné et *méthode de classe* une méthode qui s'applique à la classe dans son entier. Mais ce n'est qu'une convention : pour Perl, une méthode n'est qu'une méthode qui ne se distingue que par le type de son premier argument.

Vous pouvez considérer qu'une méthode d'instance est une action exécutée par un objet donné, comme l'action de s'afficher, de se copier ou de modifier une ou plusieurs de ses propriétés (« affecter Anduril au nom de cette épée »). Les méthodes de classe peuvent exécuter des opérations sur un ensemble d'objets (« afficher toutes les épées ») ou fournir d'autres opérations qui ne dépendent pas d'un objet particulier (« à partir de maintenant, à chaque fois qu'une nouvelle épée est forgée, enregistrer dans cette base de données le nom de son propriétaire »). Les méthodes qui produisent des instances (ou objets) d'une classe sont appelées *constructeurs* (« créer une épée incrustée de pierres précieuses et avec une inscription secrète »). Celles-ci sont en général des méthodes de classe (« fais-moi une nouvelle épée »), mais elles peuvent aussi être des méthodes d'instance (« fais-moi une copie exacte de cette épée-ci »).

Une classe peut *hériter* des méthodes de *classes parentes*, appelées aussi *classes de base* ou *surclasses*. Dans ce cas, on l'appelle *classe dérivée* ou *sous-classe*. (Brouillant un peu les cartes, certains livres entendent par « classe de base » la surclasse « racine », mais ce n'est pas ce que nous voulons dire.) Lorsque vous invoquez une méthode dont la définition ne se trouve pas dans la classe, Perl consulte automatiquement les classes parentes pour chercher une définition. Par exemple, une classe « épée » pourrait hériter sa méthode `attaquer` d'une classe générique « arme d'estoc et de taille ». Les classes parentes peuvent elles-mêmes avoir des classes parentes, et Perl examinera également ces classes si besoin est. La classe « arme d'estoc et de taille » peut à son tour hériter la méthode `attaquer` d'une classe encore plus générique « arme ».

Lorsque la méthode `attaquer` est invoquée sur un objet, le comportement résultant peut varier, selon que l'objet est une épée ou une flèche. Il se peut qu'il n'y ait aucune différence, ce qui serait le cas si les épées et les flèches héritaient leur comportement d'attaque de la classe arme générique. Mais s'il y avait une différence de comportement, le mécanisme d'acheminement de méthode sélectionnerait toujours la méthode `attaquer` la plus appropriée pour le type d'objet donné. Cette propriété fort utile de sélection du comportement le plus approprié pour un type donné d'objet s'appelle le *polymorphisme*. C'est une façon importante de ne se soucier de rien.

Vous devez vous soucier des entrailles de vos objets lorsque vous implémentez une classe, mais lorsque vous utilisez une classe (avec *use*), vous devriez traiter vos objets comme des « boîtes noires ». Vous ne voyez pas ce qu'il y a à l'intérieur, vous n'avez pas besoin de savoir comment ça marche, et vous interagissez avec la boîte sur ses termes : par les méthodes fournies par la classe. C'est comme la télécommande de votre téléviseur : même si vous savez ce qui se passe dedans, vous ne devriez pas sans bonne raison trifouiller dans ses entrailles.

Perl vous permet de scruter l'intérieur d'un objet de l'extérieur de la classe lorsque vous en avez besoin. Mais de faire ainsi enfreint l'*encapsulation*, le principe qui sépare l'interface publique (comment un objet doit être utilisé) de l'implémentation (comment l'objet fonctionne en réalité). Perl ne fournit pas un dispositif d'interface explicite, mis à part le contrat implicite entre le concepteur et l'utilisateur. L'un et l'autre sont censés être raisonnables et respectueux : l'utilisateur en ne dépendant que de l'interface documentée, le concepteur en préservant cette interface.

Perl ne vous impose pas un style de programmation particulier, et ne partage pas l'obsession avec la vie privée d'autres langages de programmation orientée objet. Néanmoins Perl a l'obsession de la liberté, et l'une de vos libertés en tant que programmeur Perl est le droit de choisir si vous voulez beaucoup ou peu de vie privée. En fait, Perl permet un espace privé plus restrictif encore que C++. C'est-à-dire qu'il n'y a rien que Perl vous empêche de faire, et en particulier il ne vous empêche pas de vous empêcher vous-même, si ce genre de choses vous intéresse. Les sections *Méthodes privées* et *Utilisation de fermetures pour objets privés* de ce chapitre vous montrent comment vous pouvez augmenter votre dose de discipline.

Reconnaissons qu'il y a beaucoup plus à dire sur les objets, et beaucoup de façons d'en apprendre plus sur la conception orientée objet. Mais ce n'est pas notre propos. Continuons donc.

Le système objet de Perl

Perl ne fournit pas de syntaxe objet particulière pour définir les objets, les classes ou les méthodes. Au lieu de cela, il réutilise des constructions existantes pour implémenter ces trois notions.[1]

Voici quelques définitions simples que vous trouverez peut-être rassurantes :

Un *objet* n'est qu'une référence... enfin, un référent.

> Comme une référence permet de représenter une collection de données avec un seul scalaire, il ne devrait pas être surprenant qu'on utilise les références pour tous les objets. Pour être précis, un objet n'est pas la référence elle-même, mais plutôt le référent sur lequel pointe la référence. Toutefois cette distinction est souvent brouillée par les programmeurs Perl, et comme nous trouvons que c'est une jolie métonymie, nous la perpétuons ici lorsque cela nous arrange.[2]

Une *classe* n'est qu'un paquetage.

> Un paquetage agit comme une classe en utilisant les fonctions du paquetage pour exécuter les méthodes de la classe et en utilisant les variables du paquetage pour contenir les données globales de la classe. Souvent, on utilise un module pour contenir une ou plusieurs classes.

Une *méthode* n'est qu'une fonction.

> Vous déclarez simplement des fonctions dans le paquetage que vous utilisez comme classe ; celles-ci seront ensuite utilisées comme méthodes de la classe. L'invocation de méthode, une nouvelle façon d'appeler une fonction, ajoute un paramètre supplémentaire : l'objet ou le paquetage utilisé pour invoquer la méthode.

Invocation de méthode

Si vous aviez à réduire toute la programmation orientée objet à une notion essentielle, ce serait l'*abstraction*. C'est le fil conducteur de tous ces mots ronflants qu'aiment employer les enthousiastes OO, comme le polymorphisme, l'héritage et l'encapsulation. Nous croyons en ces mots savants, mais nous les traiterons du point de vue pratique de ce que signifie invoquer des méthodes. Les méthodes sont au cœur des systèmes objet parce qu'elles fournissent la couche d'abstraction nécessaire à l'implémentation de ces termes savants. Au lieu d'accéder directement à une donnée résidant dans un objet, vous invoquez une méthode d'instance. Au lieu d'appeler directement une fonction dans un paquetage, vous invoquez une méthode de classe. En interposant un niveau d'indirection entre l'utilisation d'une classe et son implémentation, le concepteur de programme reste libre de bricoler le fonctionnement interne de la classe, sans grand risque d'invalider des programmes qui l'utilisent.

Perl permet deux formes syntaxiques différentes pour invoquer les méthodes. L'une utilise le style familier que vous avez déjà vu ailleurs en Perl, et la seconde est une forme

1. En *voilà* un exemple de réutilisation de logiciel !
2. Nous préférons vigueur linguistique à rigueur mathématique. Vous serez d'accord ou non.

que vous aurez peut-être déjà rencontrée dans d'autres langages de programmation. Quelle que soit la forme d'invocation de méthode utilisée, un paramètre initial supplémentaire est passé à la fonction servant de méthode. Si une classe est utilisée pour invoquer la méthode, ce paramètre sera le nom de la classe. Si un objet est utilisé pour invoquer la méthode, ce paramètre sera la référence à l'objet. Quel que soit ce paramètre, nous l'appellerons *invoquant* de la méthode. Pour une méthode de classe, l'invoquant est le nom d'un paquetage. Pour une méthode d'instance, l'invoquant est la référence qui spécifie un objet.

En d'autres termes, l'invoquant est ce *avec* quoi la méthode a été invoquée. Certains livres OO l'appellent l'*agent* ou l'*acteur* de la méthode. D'un point de vue grammatical, l'invoquant n'est ni le sujet de l'action, ni son destinataire. Il est plutôt comme un objet indirect, le bénéficiaire au nom duquel l'action est exécutée — tout comme le mot « moi » dans la commande « Forge-moi une épée ! ». Sémantiquement vous pouvez voir l'invoquant comme ce qui invoque ou ce qui est invoqué, selon ce qui correspond mieux à votre appareil mental. Nous n'allons pas vous dire comment penser. (Pas sur ce sujet-là, en tout cas.)

La plupart des méthodes sont invoquées explicitement, mais des méthodes peuvent aussi être invoquées implicitement lorsqu'elles sont déclenchées par un destructeur d'objet, un opérateur surchargé ou une variable liée. Ce ne sont pas à proprement parler des appels de fonction ordinaires, mais plutôt des invocations de méthode déclenchées automatiquement par Perl au nom de l'objet. Les destructeurs sont décrits plus loin dans ce chapitre, la surcharge est décrite dans le chapitre 13, *Surcharge*, et les variables liées sont décrites dans le chapitre 14, *Variables liées*.

Les méthodes et les fonctions normales diffèrent par le moment auquel leur paquetage est résolu — c'est-à-dire, le moment où Perl décide quel code doit être exécuté pour la méthode ou la fonction. Le paquetage d'une fonction est résolu à la compilation, avant que le programme ne commence à s'exécuter.[3] Au contraire, le paquetage d'une méthode n'est résolu que lorsqu'elle est effectivement invoquée. (Les prototypes sont vérifiés à la compilation, c'est pourquoi les fonctions normales peuvent les utiliser, mais les méthodes ne le peuvent pas.)

La raison pour laquelle le paquetage d'une méthode ne peut pas être résolu plus tôt est relativement évidente : le paquetage est déterminé par la classe de l'invoquant, et l'invoquant n'est pas connu tant que la méthode n'a réellement été invoquée. Au cœur de l'OO est cette simple chaîne logique : si on connaît l'invoquant, on connaît la classe de l'invoquant ; si on connaît la classe, on connaît l'héritage de la classe ; si on connaît l'héritage de la classe, on connaît la fonction à appeler.

La logique de l'abstraction a un coût. À cause de la résolution tardive des méthodes, une solution orientée objet en Perl risque de s'exécuter plus lentement que la solution non-OO correspondante. Pour certaines des techniques sophistiquées décrites plus loin, elle pourrait être *beaucoup* plus lente. Toutefois, la solution de beaucoup de problèmes n'est

3. Plus précisément, l'appel de fonction est résolu jusqu'à un typeglob spécifique, et une référence à ce typeglob est rentrée dans l'arbre d'opcodes compilés. Le sens de ce typeglob est négociable même au moment de l'exécution — c'est ainsi que `AUTOLOAD` peut vous autocharger une fonction. Normalement, cependant, le sens du typeglob est aussi résolu à la compilation par la définition d'une fonction nommée de façon correspondante.

pas de travailler plus vite, mais de travailler plus intelligemment. C'est là que brille l'OO.

Invocation de méthode avec l'opérateur flèche

Nous avons indiqué qu'il existe deux styles d'invocation de méthode. Le premier style se présente comme ceci :

```
INVOQUANT->METHODE(LISTE)
INVOQUANT->METHODE
```

Pour des raisons évidentes, ce style est habituellement appelé la forme flèche d'invocation. (Ne confondez pas `->` avec `=>`, la flèche « à deux coups » utilisée comme virgule de luxe.) Les parenthèses sont exigées s'il y a des paramètres. Lorsqu'elle est exécutée, l'invocation commence par trouver la fonction déterminée conjointement par la classe de l'*INVOQUANT* et par le nom de la *METHODE*, lui passant *INVOQUANT* comme premier paramètre.

Lorsque *INVOQUANT* est une référence, nous disons que *METHODE* est invoquée comme méthode d'instance, et lorsque *INVOQUANT* est un nom de paquetage, nous disons que *METHODE* est invoquée comme méthode de classe. Il n'y a en réalité aucune différence entre les deux, mis à part que le nom de paquetage est plus clairement associé à la classe elle-même qu'avec les objets de la classe. Vous allez devoir nous faire confiance lorsque nous disons que les objets connaissent leur classe. Nous vous dirons bientôt comment associer un objet à un nom de classe, mais vous pouvez utiliser les objets sans savoir cela.

Par exemple, pour construire un objet avec la méthode de classe `invoquer` puis invoquer la méthode d'instance `dire` sur l'objet résultant, vous pourriez dire ceci :

```
$mage = Magicien->invoquer("Gandalf");  # méthode de classe
$mage->dire("ami");                     # méthode d'instance
```

Les méthodes `invoquer` et `dire` sont définies par la classe `Magicien` — ou par une des classes dont elle hérite. Mais ne vous inquiétez pas de cela. Ne vous mêlez pas des affaires de Magiciens.

Comme l'opérateur flèche est associatif vers la gauche (voir le chapitre 3, *Opérateurs unaires et binaires*), vous pouvez même combiner les deux instructions :

```
Magicien->invoquer("Gandalf")->dire("ami");
```

Parfois vous souhaiterez invoquer une méthode sans connaître son nom par avance. Vous pouvez utiliser la forme flèche d'invocation et remplacer le nom de méthode avec une simple variable scalaire :

```
$methode = "invoquer";
$mage = Magicien->$methode("Gandalf"); # Invoquer Magicien->invoquer

$voyager = $compagnon eq "Grispoil" ? "monter" : "marcher";
$mage->$voyager("sept lieues");       # Invoquer $mage->monter ou
                                      # $mage->marcher
```

Bien que vous utilisiez le nom d'une méthode pour l'invoquer indirectement, cet usage n'est pas interdit par `use strict 'refs'`, car *tous* les appels de méthode sont en fait recherchés par symbole au moment où ils sont résolus.

Dans notre exemple, nous stockons le nom d'une fonction dans `$voyager`, mais vous pourriez aussi stocker une référence de fonction. Ceci contourne l'algorithme de recherche de méthode, mais parfois c'est exactement ce que vous voulez faire. Voir la section *Méthodes privées* et la discussion de la méthode `can` dans la section *UNIVERSAL : la classe ancêtre ultime*. Pour créer une référence à la méthode spécifique d'appel pour une instance spécifique, voir la section *Fermetures* dans le chapitre 8.

Invocation de méthode avec des objets indirects

Le second style d'invocation de méthode se présente comme ceci :

```
METHODE INVOQUANT (LISTE)
METHODE INVOQUANT LISTE
METHODE INVOQUANT
```

Les parenthèses autour de *LISTE* sont optionnelles ; si elles sont omises, la méthode se comporte comme un opérateur de liste. Vous pouvez donc avoir des instructions comme les suivantes, toutes utilisant ce style d'appel de méthode :

```
$mage = invoquer Magicien "Gandalf";
$nemesis = invoquer Balrog demeure => "Moria", arme => "fouet";
deplacer $nemesis "pont";
dire $mage "Tu ne peux pas passer";
briser $baton;           # il est plus sûr d'utiliser : briser $baton ();
```

La syntaxe d'opérateur de liste devrait vous être familière, car c'est le même style qu'on utilise pour passer des handles de fichiers à `print` ou à `printf` :

```
print STDERR "au secours !!!\n";
```

Elle est également similaire à des phrases en français comme « Donner (à) Gollum le trésor », donc nous l'appelons la forme avec *objet indirect*. L'invoquant est attendu dans la *case d'objet indirect*. Lorsqu'il est fait mention de passer quelque chose à une fonction interne comme `system` ou `exec` dans sa « case d'objet indirect », cela veut dire que vous fournissez ce paramètre supplémentaire et dépourvu de virgule au même endroit que vous le feriez si vous invoquiez une méthode avec la syntaxe d'objet indirect.

La forme avec objet indirect vous permet même de spécifier que l'*INVOQUANT* soit un *BLOC* évaluable comme objet (référence) ou comme classe (paquetage). Ce qui vous permet de combiner deux invocations en une instruction ainsi :

```
dire { invoquer Magicien "Gandalf" } "ami";
```

Pièges syntaxiques avec les objets indirects

Une syntaxe sera souvent plus lisible que l'autre. La syntaxe d'objet indirect est moins lourde, mais souffre de plusieurs sortes d'ambiguïté syntaxique. La première est que la partie *LISTE* d'une invocation avec objet indirect est analysée de la même façon que tout autre opérateur de liste. Ainsi, les parenthèses de :

```
enchanter $epee ($points + 2) * $cout;
```

sont censées entourer tous les paramètres, quel que soit ce qui suit. Cela équivaut à :

```
($epee->enchanter($points + 2)) * $cout;
```

Cela ne fait probablement pas ce que vous pensez : `enchanter` n'est appelée qu'avec `$points + 2`, et la valeur de retour de la méthode est ensuite multipliée par `$cout`. Comme avec d'autres opérateurs de liste, vous devez faire attention à la précédence de `&&` et de `||` par rapport à `and` et à `or`.

Par exemple ceci :

```
appeler $epee $ancien_nom || "Glamdring";  # ne pas utiliser "or" ici !
```

devient comme prévu :

```
$epee->appeler($ancien_nom || "Glamdring");
```

mais ceci :

```
dire $mage "ami" && entrer();              # il aurait fallu "and" ici !
```

devient le douteux :

```
$mage->dire("ami" && entrer());
```

qui pourrait être corrigé en récrivant avec une des formes équivalentes :

```
entrer() if $mage->dire("ami");
$mage->dire("ami") && entrer();
dire $mage "ami" and entrer();
```

Le deuxième ennui syntaxique de la forme avec objet indirect est que l'*INVOQUANT* ne peut être qu'un nom, une variable scalaire non-indicée ou un bloc.[4] Dès que l'analyseur voit une de ces choses, il a son *INVOQUANT*, donc il commence à chercher sa *LISTE*. Ainsi ces invocations :

```
deplacer $groupe->{CHEF};              # probablement faux !
deplacer $cavaliers[$i];               # probablement faux !
```

sont en fait analysées comme celles-ci :

```
$groupe->deplacer->{CHEF};
$cavaliers->deplacer([$i]);
```

au lieu de ce que vous vouliez sans doute :

```
$groupe->{CHEF}->deplacer;
$cavaliers[$i]->deplacer;
```

L'analyseur ne regarde qu'un petit peu vers l'avant pour trouver l'invoquant d'un objet indirect, pas même aussi loin qu'il ne regarderait dans le cas d'un opérateur unaire. Cette bizarrerie n'a pas lieu avec la première forme syntaxique, donc vous souhaiterez peut-être faire de la flèche votre arme de choix.

Même en français il peut se poser un problème similaire. Pensez à l'instruction : « Envoyez-moi par la poste la lettre pour que je la lise ». Si vous analysez cette phrase trop vite, vous finirez par envoyer quelqu'un, et non pas une lettre, par la poste. Comme Perl, le français a parfois deux syntaxes différentes pour décrire un agent : « Envoyez-moi la lettre » et « Envoyez la lettre à moi ». Parfois la forme plus longue est plus claire et plus naturelle, et parfois c'est le contraire. Au moins en Perl, vous êtes obligé d'utiliser des accolades autour d'un objet indirect complexe.

4. Les lecteurs attentifs se souviendront que c'est précisément la même liste d'items syntaxiques qui sont permis après les drôles de caractères (`$`, `@`, `%` ou `*`) pour indiquer un déréférencement de variable — par exemple, `@tableau`, `@$ref_tableau` ou `@{$ref_tableau}`.

Classes à paquetage explicite

La dernière ambiguïté syntaxique du style d'invocation de méthode avec objet indirect est qu'elle peut ne pas du tout être analysée comme appel de méthode, parce que le paquetage courant peut contenir une fonction ayant le même nom que la méthode. Lorsque vous utilisez une méthode de classe avec un nom de paquetage littéral comme invoquant, il y a moyen de résoudre cette ambiguïté tout en gardant la syntaxe d'objet indirect : explicitez le nom de paquetage en y ajoutant un double deux-points.

```
$obj = methode CLASSE::;  # imposer "CLASSE"->methode
```

Ceci est important car la notation courante :

```
$obj = new CLASSE;        # pourrait ne pas être analysé comme méthode
```

ne se comportera pas toujours correctement si le paquetage courant comporte une fonction nommée `new` ou *`CLASSE`*. Même si prudemment vous utilisez la forme avec flèche au lieu de la forme avec objet indirect pour invoquer des méthodes, ceci peut, rarement, rester un problème. Au prix d'un peu plus de ponctuation, la notation *`CLASSE`*`::` garantit comment Perl analysera votre invocation de méthode. Dans les exemples suivants, les deux premiers ne sont pas toujours analysés de la même façon, mais les deux derniers le sont :

```
$obj = new AnneauElfique;           # pourrait être new("AnneauElfique")
                                    # ou même new(AnneauElfique())
$obj = AnneauElfique->new;          # pourrait être AnneauElfique()->new()

$obj = new AnneauElfique::;         # toujours "AnneauElfique"->new()
$obj = AnneauElfique::->new;        # toujours "AnneauElfique"->new()
```

Cette notation de paquetage explicite peut être enjolivée avec un peu d'alignement :

```
$obj = new AnneauElfique::
            nom          => "Narya",
            proprietaire => "Gandalf",
            domaine      => "feu",
            pierre       => "rubis";
```

Néanmoins, vous vous exclamerez peut-être « Qu'il est laid ! » en voyant ce double deux-points, donc nous vous dirons que vous pouvez presque toujours vous en sortir avec un simple nom de classe, à deux conditions. Premièrement, qu'il n'y ait pas de fonction avec le même nom que la classe. (Si vous observez la convention que les noms de fonction comme `new` commencent avec une lettre minuscule, et que les noms de classe comme `AnneauElfique` commencent avec une majuscule, ceci ne posera jamais de problème.) Deuxièmement, que la classe soit chargée avec l'une des instructions suivantes :

```
use AnneauElfique;
require AnneauElfique;
```

Chacune de ces déclarations assure que Perl sait que `AnneauElfique` est un nom de module, ce qui oblige tout nom simple comme `new` avant le nom de classe `AnneauElfique` à être interprété comme un appel de méthode, même s'il se trouve que vous avez vous-même déclaré une fonction `new` dans le paquetage courant. On ne rencontre habituellement de problème avec les objets indirects que si l'on entasse plusieurs classes dans un même fichier, auquel cas Perl ne sait pas forcément qu'un paquetage spécifique était

censé être un nom de classe. Ceux qui nomment des fonctions avec des noms qui ressemblent à `NomsDeModule` finissent également par le regretter un jour ou l'autre.

Construction d'objet

Tout objet est une référence, mais toute référence n'est pas un objet. Une référence ne fonctionnera comme objet que si son référent est marqué spécialement pour dire à Perl à quel paquetage il appartient. On appelle *consacrer* (du sens anglais de la fonction `bless`) l'acte de marquer un référent avec le nom d'un paquetage — et donc de sa classe, puisqu'une classe n'est qu'un paquetage. Vous pouvez considérer que consacrer une référence la transforme en objet, bien qu'il soit plus précis de dire que cela transforme la référence en référence à un objet.

La fonction `bless` prend un ou deux paramètres. Le premier paramètre est une référence et le deuxième est le paquetage avec lequel est consacré le référent. Si le deuxième paramètre est omis, c'est le paquetage courant qui est utilisé.

```
$obj = { };              # Obtenir une référence à un hachage anonyme.
bless($obj);             # Consacrer le hachage avec le paquetage courant.
bless($obj, "Bestiau");  # Consacrer le hachage avec la classe Bestiau.
```

Ici nous avons utilisé une référence à un hachage anonyme, ce que les gens utilisent habituellement comme structure de données pour leurs objets. Les hachages sont après tout extrêmement flexibles. Mais permettez-nous de souligner que vous pouvez consacrer une référence à toute chose à laquelle vous pouvez faire une référence en Perl, comme les scalaires, les tableaux, les fonctions et les typeglobs. Vous pouvez même consacrer une référence au hachage de la table de symboles d'un paquetage si vous trouvez une bonne raison de le faire. (Ou même si vous n'en trouvez pas.) L'orientation objet de Perl est entièrement distincte de la structure des données.

Une fois que le référent est consacré, l'appel de la fonction interne `ref` sur sa référence retourne le nom de la classe consacrée au lieu du type de base, comme `HASH`. Si vous voulez le type de base, utilisez la fonction `reftype` du module `attributes`. Voir `use attributes` dans le chapitre 31, *Modules de pragmas*.

Voilà comment créer un objet. Prenez simplement une référence à quelque chose, donnez-lui une classe en la consacrant avec un paquetage, et vous avez terminé. C'est tout ce que vous avez à faire si vous créez une classe minimale. Si vous utilisez une classe, vous avez encore moins à faire, car l'auteur de la classe aura caché le `bless` à l'intérieur d'une fonction nommée *constructeur*, qui crée et retourne des instances de la classe. Comme `bless` retourne son premier paramètre, un constructeur typique peut être aussi simple que ceci :

```
package Bestiau;
sub engendrer { bless {}; }
```

Ou, pour être un peu plus explicite :

```
package Bestiau;
sub engendrer {
    my     $self = {};        # Référence à un hachage anonyme vide
    bless  $self, "Bestiau";  # Faire de ce hachage un objet Bestiau
    return $self;             # Retourner le Bestiau nouvellement créée
}
```

Muni de cette définition, voici comment on pourrait créer un objet `Bestiau` :

```
$animal = Bestiau->engendrer;
```

Constructeurs héritables

Comme toutes les méthodes, un constructeur n'est qu'une fonction, mais nous ne l'appelons pas fonction. Nous l'invoquons toujours comme méthode — ici une méthode de classe, car l'invoquant est un nom de paquetage. Les invocations de méthode diffèrent des appels de fonctions normales de deux façons. Premièrement, elles reçoivent un paramètre supplémentaire, comme indiqué précédemment. Deuxièmement, elles obéissent à l'héritage, permettant ainsi à une classe d'utiliser les méthodes d'une autre.

Nous décrirons plus rigoureusement la mécanique sous-jacente de l'héritage dans la section suivante, mais pour l'instant, quelques exemples simples de ses effets devraient vous permettre de créer vos constructeurs. Par exemple, supposons que nous avons une classe `Araignee` qui hérite des méthodes de la classe `Bestiau`. En particulier, supposons que la classe `Araignee` ne possède pas sa propre méthode `engendrer`. On obtient les correspondances suivantes :

Appel de méthode	Appel de fonction résultant
`Bestiau->engendrer()`	`Bestiau::engendrer("Bestiau").`
`Araignee->engendrer()`	`Bestiau::engendrer("Araignee").`

La fonction appelée est la même dans les deux cas, mais le paramètre est différent. Notez que notre constructeur `engendrer` ne prêtait aucune attention à son paramètre, ce qui signifie que notre objet `Araignee` était incorrectement consacré avec la classe `Bestiau`. Un meilleur constructeur fournirait le nom du paquetage (passé comme premier paramètre) à `bless` :

```
sub engendrer {
    my $classe  = shift;      # Enregistrer le nom de paquetage
    my $self    = { };
    bless($self, $classe);    # Consacrer la référence avec ce paquetage
    return $self;
}
```

Maintenant vous pourriez utiliser la même fonction pour les deux cas :

```
$vermine = Bestiau->engendrer;
$shelob  = Araignee->engendrer;
```

Et chaque objet serait de la classe appropriée. Ceci fonctionne même indirectement, comme par exemple :

```
$type  = "Araignee";
$shelob = $type->engendrer;   # pareil que "Araignee"->engendrer
```

Cela reste une méthode de classe, pas une méthode d'instance, parce que son invoquant contient une chaîne et non une référence.

Si `$type` était un objet au lieu d'un nom de classe, la définition de constructeur ci-dessus ne fonctionnerait pas, car `bless` nécessite un nom de classe. Mais pour beaucoup de clas-

ses, il est intéressant d'utiliser un objet existant comme modèle pour la création d'un nouvel objet. Dans ces cas, vous pouvez définir vos constructeurs de façon à ce qu'ils fonctionnent à la fois avec des objets ou avec des noms de classe :

```
sub engendrer {
    my $invoquant = shift;
    my $classe    = ref($invoquant) || $invoquant; # Objet ou nom de classe
    my $self      = { };
    bless($self,   $classe);
    return $self;
}
```

Initialiseurs

La majorité des objets gèrent des informations internes qui sont indirectement manipulées par les méthodes de l'objet. Tous nos constructeurs jusqu'à maintenant ont créé des hachages vides, mais il n'y a pas de raison de les laisser vides. Par exemple, un constructeur pourrait accepter des paramètres supplémentaires à ranger dans le hachage comme couples clef/valeur. Les livres OO appellent souvent de telles données *propriétés*, *attributs*, *accesseurs*, *données membres*, *données d'instance* ou *variables d'instance*. La section *Gestion des données d'instance* plus loin dans ce chapitre donne plus de détails sur les attributs.

Imaginez une classe `Cheval` avec des attributs d'instance comme « nom » et « couleur » :

```
$monture = Cheval->new(nom => "Grispoil", couleur => "blanc");
```

Si l'objet est implémenté comme référence à un hachage, les couples clef/valeur peuvent être interpolés directement dans le hachage une fois que l'invoquant est supprimé de la liste des paramètres :

```
sub new {
    my $invoquant = shift;
    my $classe = ref($invoquant) || $invoquant;
    my $self = { @_ };     # Les paramètres restants deviennent des attributs
    bless($self, $classe);# Transformer en objet
    return $self;
}
```

Cette fois-ci nous avons utilisé une méthode appelée `new` comme constructeur de la classe, ce qui pourrait bien donner aux programmeurs C++ l'impression de savoir ce qui se passe. Mais Perl ne considère pas « `new` » comme étant quelque chose de spécial ; vous pouvez nommer vos constructeurs comme vous voulez. Toute méthode qui crée et retourne un objet est de fait un constructeur. En général, nous vous recommandons de donner à vos constructeurs des noms qui correspondent au contexte du problème que vous résolvez. Par exemple, les constructeurs du module `Tk` ont le même nom que le widget qu'ils créent. Dans le module `DBI`, un constructeur nommé `connect` retourne un objet handle de base de données, et un autre constructeur nommé `prepare` est invoqué comme méthode d'instance et retourne un objet handle de statement. Mais s'il n'y a pas de nom de constructeur qui convienne au contexte, `new` n'est peut-être pas un mauvais choix. Cela dit, ce n'est peut-être pas une si mauvaise chose de choisir au hasard un nom

pour obliger les gens à lire le contrat d'interface (c'est-à-dire la documentation de la classe) avant d'utiliser ses constructeurs.

Vous pouvez aussi configurer votre constructeur avec des couples clef/valeur par défaut, que l'utilisateur peut supplanter en les passant comme paramètres :

```
sub new {
    my $invoquant = shift;
    my $classe    = ref($invoquant) || $invoquant;
    my $self = {
        couleur       => "bai",
        pattes        => 4,
        proprietaire  => undef,
        @_,                     # Supplanter les attributs précédents
    };
    return bless $self, $classe;
}

$ed     = Cheval->new;                      # Un cheval bai à 4 pattes
$etalon = Cheval->new(couleur => "noir");  # Un cheval noir à 4 pattes
```

Ce constructeur de `Cheval` ne prête pas attention aux attributs existants de son invoquant lorsqu'il est utilisé comme méthode d'instance. Vous pourriez créer un deuxième constructeur conçu pour être appelé comme méthode d'instance, et s'il était conçu correctement, vous pourriez utiliser les valeurs de l'objet invoquant comme valeurs par défaut du nouvel objet :

```
$monture  = Cheval->new(couleur => "gris");
$poulain  = $monture->clone(proprietaire => "EquuGen Guild, Ltd.");

sub clone {
    my $modele = shift;
    my $self   = $modele->new(%$modele, @_);
    return $self;     # Déjà consacré par ->new
}
```

(Vous auriez également pu intégrer cette fonction directement dans `new`, mais alors le nom ne correspondrait plus tout à fait à son rôle.)

Remarquez comment, même dans le constructeur `clone`, nous ne codons pas en dur le nom de la classe `Cheval`. Ainsi l'objet d'origine invoque sa méthode `new`, quelle qu'elle soit. Si nous avions écrit cela `Cheval->new` au lieu de `$modele->new`, la classe n'aurait pas facilité l'héritage par une classe `Zebre` ou une classe `Licorne`. Il serait dommage de cloner Pégase et de se retrouver avec un vulgaire bidet.

Cependant, parfois vous avez le problème contraire : au lieu de partager un même constructeur entre plusieurs classes, vous essayez de faire partager l'objet d'une même classe entre plusieurs constructeurs. Cela se passe à chaque fois qu'un constructeur veut appeler le constructeur d'une classe de base pour faire une partie du travail de construction. Perl ne fait pas de construction hiérarchique à votre place. C'est-à-dire que Perl n'appelle pas automatiquement les constructeurs (ou les destructeurs) des classes de base de la classe en question, donc votre constructeur devra faire cela lui-même et ensuite rajouter tout autre attribut dont a besoin la classe dérivée. Donc la situation est similaire à la mé-

thode `clone`, sauf qu'au lieu de copier un objet existant vers un nouvel objet, vous voulez appeler le constructeur de votre classe de base et ensuite transformer le nouvel objet de base en votre nouvel objet dérivé.

Héritage de classe

Comme pour le reste du système objet de Perl, l'héritage d'une classe par une autre ne nécessite l'ajout au langage d'aucune syntaxe particulière. Lorsque vous invoquez une méthode pour laquelle Perl ne trouve pas de fonction dans le paquetage de l'invoquant, le tableau `@ISA`[5] de ce paquetage est examiné. C'est ainsi que Perl implémente l'héritage : chaque élément du tableau `@ISA` d'un paquetage donné contient le nom d'un autre paquetage, dans lequel est faite une recherche lorsque manquent des méthodes. Par exemple, ce qui suit fait de la classe `Cheval` une sous-classe de la classe `Bestiau`. (Nous déclarons `@ISA` avec `our` parce qu'il doit être une variable de paquetage, pas une variable lexicale déclarée avec `my`.)

```
package Cheval;
our @ISA = "Bestiau";
```

Vous devriez maintenant pouvoir utiliser un objet ou une classe `Cheval` partout où `Bestiau` était utilisé. Si votre classe réussit ce *test de sous-classe vide*, vous savez que `Bestiau` est une classe de base correcte, apte à l'héritage.

Supposons que vous avez un objet `Cheval` dans `$monture` et que vous invoquez une méthode `deplacer` dessus :

```
$monture->deplacer(10);
```

Comme `$monture` est un `Cheval`, le premier choix de Perl pour cette méthode est la fonction `Cheval::deplacer`. S'il n'y en a pas, au lieu de lever une exception à l'exécution, Perl consulte d'abord le premier élément du tableau `@Cheval::ISA`, qui lui indique de chercher `Bestiau::deplacer` dans le paquetage `Bestiau`. Si cette fonction n'est pas trouvée non plus, et que `Bestiau` a son propre tableau `@Bestiau::ISA`, alors lui aussi sera consulté pour trouver le nom d'un paquetage ancestral qui puisse fournir une méthode `deplacer`, et ainsi de suite en remontant la hiérarchie d'héritage jusqu'à ce que l'on tombe sur un paquetage sans un `@ISA`.

La situation que nous venons de décrire est l'*héritage simple*, dans lequel chaque classe n'a qu'un seul parent. Perl permet aussi l'*héritage multiple* : il suffit d'ajouter d'autres paquetages à l'`@ISA` de la classe. Ce type d'héritage fonctionne davantage comme une structure de données en arbre, car chaque paquetage peut avoir plus d'un parent immédiat. Certaines personnes trouvent cela plus sexy.

Lorsque vous invoquez une méthode *`nom_meth`* sur un invoquant de type *`nom_classe`*, Perl essaie six façons différentes de trouver une fonction à utiliser :

1. D'abord, Perl cherche une fonction nommée *`nom_classe`*`::`*`nom_meth`* dans le paquetage de l'invoquant. Si cela échoue, l'héritage entre en jeu, et nous passons à la deuxième étape.

5. Se prononce « is a » (« est un ») en anglais, comme « A horse is a critter. » (« Un cheval est un bestiau. »)

2. Ensuite, Perl cherche une méthode héritée de classes de base en cherchant dans tous les paquetages *parent* listés dans `@`*nom_classe*`::ISA` une fonction *parent*`::`*nom_meth*. La recherche se fait de gauche à droite, récursivement, et en cherchant en profondeur d'abord (depth-first). La récursivité garantit que l'on recherche dans les classes grand-parent, arrière-grand-parent, arrière-arrière-grand-parent et ainsi de suite.
3. Si cela échoue, Perl cherche une fonction nommée `UNIVERSAL::`*nom_meth*.
4. À ce stade, Perl abandonne *nom_meth* et commence à chercher un `AUTOLOAD`. D'abord, il cherche une fonction nommée *nom_classe*`::AUTOLOAD`.
5. S'il n'en trouve pas, Perl cherche dans toutes les classes *parent* listées dans `@`*nom_classe*`::ISA` une fonction *parent*`::AUTOLOAD` . La recherche est à nouveau de gauche à droite, récursive, et en profondeur d'abord.
6. Pour finir, Perl cherche une fonction nommée `UNIVERSAL::AUTOLOAD`.

Perl s'arrête à la première tentative réussie et invoque cette fonction. Si aucune fonction n'est trouvée, une exception est levée, que vous verrez fréquemment :

```
Can't locate object method "nom_meth" via package "nom_classe"
```

Si vous avez compilé une version Perl de débogage avec l'option **`-DDEBUGGING`** de votre compilateur C, en utilisant l'option **`-Do`** de Perl, vous pouvez le regarder passer par chacune de ces étapes lorsqu'il résout une invocation de méthode.

Nous détaillerons le mécanisme d'héritage au fur et à mesure dans ce qui suit.

Héritage avec @ISA

Si `@ISA` contient plusieurs noms de paquetage, les paquetages sont tous parcourus de gauche à droite. La recherche se fait en profondeur d'abord, donc si vous avez une classe `Mule` configurée pour l'héritage de la façon suivante :

```
package Mule;
our @ISA = ("Cheval", "Ane");
```

Perl cherche les méthodes manquantes de `Mule` d'abord dans `Cheval` (et dans tous ses ancêtres, comme `Bestiau`) avant de continuer à chercher dans `Ane` et ses ancêtres.

Si une méthode manquante est trouvée dans la classe de base, pour plus d'efficacité, Perl place cet emplacement en interne dans un cache dans la classe courante, pour qu'il n'ait pas à chercher aussi loin la prochaine fois qu'il doit trouver cette méthode. Modifier `@ISA` ou définir de nouvelles méthodes invalide le cache et oblige Perl à faire la recherche de nouveau.

Lorsque Perl cherche une méthode, il s'assure que vous n'avez pas créé de hiérarchie d'héritage circulaire. Ceci pourrait se produire si deux classes héritaient l'une de l'autre, même indirectement par d'autres classes. Essayer d'être votre propre grand-père est trop paradoxal, même pour Perl, donc la tentative lève une exception. Toutefois, Perl ne considère pas comme une erreur d'hériter de plusieurs classes partageant de mêmes ancêtres, ce qui fait penser à un mariage entre cousins. Simplement, votre hiérarchie d'héritage ne ressemble plus à un arbre et commence à ressembler à un graphe acyclique orienté. Ceci ne dérange pas Perl — tant que le graphe est réellement acyclique.

Lorsque vous initialisez `@ISA`, l'affectation se produit normalement à l'exécution, donc sauf si vous prenez des précautions, le code dans un bloc `BEGIN`, `CHECK` ou `INIT` ne pourra pas utiliser la hiérarchie d'héritage. Le pragma `use base` est une précaution (ou commodité) qui vous permet d'utiliser des classes avec `require` et les ajoute à `@ISA` à la compilation. Voici comment vous pourriez l'utiliser :

```
package Mule;
use base ("Cheval", "Ane");   # déclarer les surclasses
```

Ceci est un raccourci de :

```
package Mule;
BEGIN {
    our @ISA = ("Cheval", "Ane");
    require Cheval;
    require Ane;
}
```

sauf que `use base` prend aussi en compte toute déclaration `use fields`.

Il arrive que les gens s'étonnent que l'inclusion d'une classe dans `@ISA` n'effectue pas le `require` correspondant à votre place. C'est parce que le système de classes de Perl est largement séparé du système de modules. Un fichier peut contenir plusieurs classes (puisqu'elles ne sont que des paquetages), et un paquetage peut être mentionné dans plusieurs fichiers. Mais dans la situation la plus courante, dans laquelle un paquetage et une classe et un module et un fichier deviennent assez interchangeables si vous plissez suffisamment les yeux, le pragma `use base` fournit une syntaxe déclarative qui établit l'héritage, charge les fichiers de module, et gère les déclarations de champs de classe de base. C'est une de ces diagonales commodes que nous mentionnons souvent.

Voir pour plus de détails les descriptions de `use base` et de `use fields` au chapitre 31.

Accès aux méthodes supplantées

Lorsqu'une classe définit une méthode, cette fonction supplante les méthodes de même nom dans toutes les classes de base. Imaginons que vous avez un objet `Mule` (dérivé de la classe `Cheval` et de la classe `Ane`), et que vous décidez d'invoquer la méthode `procreer` de votre objet. Bien que les classes parentes aient leur propres méthodes `procreer`, le concepteur de la classe `Mule` les a supplantées en fournissant à la classe `Mule` sa propre méthode `procreer`. Cela veut dire que le croisement suivant risque de ne pas être productif :

```
$etalon = Cheval->new(sexe => "male");
$molly = Mule->new(sexe => "femelle");
$poulain = $molly->procreer($etalon);
```

Supposons maintenant que par le miracle de l'ingénierie génétique, vous trouvez un moyen de contourner le problème bien connu de stérilité de la mule, donc vous souhaitez passer outre la méthode `Mule::procreer` non viable. Vous *pourriez* appeler votre méthode comme une fonction ordinaire, en s'assurant de passer explicitement l'invoquant :

```
$poulain = Cheval::procreer($molly, $etalon);
```

Toutefois ceci contourne l'héritage, ce qui est presque toujours la mauvaise chose à faire.

Il est parfaitement imaginable que la fonction `Cheval::procreer` n'existe pas, car `Cheval` et `Ane` dérivent ce comportement d'un parent commun nommé `Equide`. Si, d'un autre côté, vous voulez que Perl *commence* à chercher une méthode dans une classe donnée, utilisez simplement l'invocation de méthode ordinaire, mais qualifiez le nom de méthode avec la classe :

```
$poulain = $molly->Cheval::procreer($etalon);
```

Parfois vous voudrez qu'une méthode dans une classe dérivée serve d'emballage pour une méthode dans une classe de base. La méthode de la classe dérivée peut elle-même appeler la méthode dans la classe de base, ajoutant ses propres actions avant ou après cette invocation. Vous *pourriez* utiliser la notation ci-dessus pour spécifier dans quelle classe commencer la recherche. Mais dans la plupart des cas de méthodes supplantées, vous voudrez probablement éviter de savoir ou de spécifier quelle classe parente appartient à la méthode supplantée que vous voulez exécuter.

C'est là que la pseudo-classe `SUPER` est très utile. Elle vous permet d'invoquer une méthode supplantée de classe de base sans avoir à spécifier quelle classe a défini cette méthode.[6] La fonction suivante consulte le tableau `@ISA` du paquetage courant sans vous faire spécifier de nom de classe :

```
package Mule;
our @ISA = qw(Cheval Ane);
sub coup_de_pied {
    my $self = shift;
    print "La mule donne un coup de pied !\n";
    $self->SUPER::coup_de_pied(@_);
}
```

Le pseudo-paquetage n'a un sens que lorsqu'il est utilisé *à l'intérieur* d'une méthode. Bien que celui qui implémente une classe peut faire usage de `SUPER` dans son propre code, quelqu'un qui ne fait qu'utiliser les objets de la classe ne le peut pas.

`SUPER` ne fonctionne pas toujours comme vous le voudriez dans le cas d'un héritage multiple. Comme vous pourriez vous y attendre, il parcourt `@ISA` exactement comme le fait le mécanisme d'héritage normal : de gauche à droite, récursivement, et en profondeur d'abord. Si `Cheval` et `Ane` avaient chacun une méthode `dire`, et que vous préfériez la méthode d'`Ane`, vous auriez à nommer explicitement la classe parente :

```
sub dire {
    my $self = shift;
    print "La mule parle !\n";
    $self->Ane::dire(@_);
}
```

Des manières plus élaborées de traiter les situations avec héritage multiple peuvent être créées avec la méthode `UNIVERSAL::can` décrite dans la section suivante. Vous pouvez aussi récupérer le module `Class::Multimethods` de CPAN, qui fournit de nombreuses

6. Ceci n'est pas à confondre avec le mécanisme mentionné au chapitre 11 pour supplanter les fonctions internes de Perl, qui ne sont pas des méthodes d'objet et ne sont donc pas supplantées par héritage. Vous appelez les fonctions internes supplantées avec le pseudo-paquetage `CORE`, pas le pseudo-paquetage `SUPER`.

solutions sophistiquées, dont une qui trouve la méthode la plus proche au lieu de celle située le plus à gauche.

Chaque bout de code Perl sait dans quel paquetage il se trouve, d'après la dernière instruction `package`. Une méthode `SUPER` ne consulte le `@ISA` que du paquetage dans lequel l'appel à `SUPER` a été compilé. Il ne se soucie pas de la classe de l'invoquant, ni du paquetage de la fonction appelée. Ceci peut causer des problèmes si vous tentez de définir des méthodes dans une autre classe simplement en jouant sur le nom de méthode :

```
package Oiseau;
use Libellule;
sub Libellule::bombarder { shift->SUPER::bombarder(@_) }
```

Malheureusement, ceci invoque la surclasse d'`Oiseau`, pas celle de `Libellule`. Pour faire ce que vous essayez de faire, vous devez aussi explicitement passer dans le paquetage approprié pour la compilation de `SUPER` :

```
package Oiseau;
use Libellule;
{
    package Libellule;
    sub bombarder { shift->SUPER::bombarder(@_) }
}
```

Comme l'indique cet exemple, il n'est jamais nécessaire de modifier un fichier de module pour simplement ajouter des méthodes à une classe existante. Puisqu'une classe n'est qu'un paquetage, et qu'une méthode n'est qu'une fonction, tout ce que vous devez faire c'est définir une fonction dans ce paquetage comme nous l'avons fait ici, et la classe possède soudainement une nouvelle méthode. L'héritage n'est pas nécessaire. Seul compte le paquetage, et comme les paquetages sont globaux, on peut accéder à tout paquetage de n'importe où dans le programme. (Vous a-t-on informé que nous allions installer une baignoire dans votre salon la semaine prochaine ?)

UNIVERSAL : la classe ancêtre ultime

Si aucune définition de méthode de nom correspondant n'est trouvée après avoir recherché dans la classe de l'invoquant et récursivement dans toutes ses classes ancêtres, une dernière recherche de méthode de ce nom est faite dans la classe spéciale prédéfinie nommée `UNIVERSAL`. Ce paquetage n'apparaît jamais dans un `@ISA`, mais il est toujours consulté lorsque la recherche dans `@ISA` échoue. Vous pouvez voir `UNIVERSAL` comme l'ancêtre ultime dont dérivent implicitement toutes les classes.

Les méthodes prédéfinies suivantes sont disponibles dans la classe `UNIVERSAL`, et donc dans toutes les classes. Elles fonctionnent toutes qu'elles soient invoquées comme méthodes de classe ou comme méthodes d'objet.

INVOQUANT`->isa(`*CLASSE*`)`

> La méthode `isa` retourne vrai si la classe d'*INVOQUANT* est *CLASSE* ou une classe héritant de *CLASSE*. À la place d'un nom de paquetage, *CLASSE* peut aussi être un des types internes, comme « `HASH` » ou « `ARRAY` ». (Mais la détermination du type exact n'est pas bon signe pour l'encapsulation et le polymorphisme. Vous devriez dépendre de l'acheminement de méthode pour vous donner la bonne méthode.)

```
use FileHandle;
if (FileHandle->isa("Exporter")) {
    print "FileHandle est un Exporter.\n";
}

$fh = FileHandle->new();
if ($fh->isa("IO::Handle")) {
    print "\$fh est un objet en rapport avec IO.\n";
}
if ($fh->isa("GLOB")) {
    print "\$fh est en fait une référence à un GLOB.\n";
}
```

INVOQUANT->can(*METHODE*)

La méthode can retourne une référence à la fonction qui serait appelée si *METHODE* était appliquée à *INVOQUANT*. Si une telle fonction n'est pas trouvée, can retourne undef.

```
if ($invoquant->can("copier")) {
    print "Notre invoquant peut copier.\n";
}
```

Ceci pourrait être utilisé pour n'invoquer une méthode que si elle existe :

```
$obj->grogner if $obj->can("grogner");
```

Avec l'héritage multiple, ceci permet à une méthode d'invoquer toutes les méthodes de classe de base supplantées, et non pas seulement celle la plus à gauche :

```
sub grogner {
    my $self = shift;
    print "Grognement: @_\n";
    my %vues;
    for my $parent (@ISA) {
        if (my $code = $parent->can("grogner")) {
            $self->$code(@_) unless $vues{$code}++;
        }
    }
}
```

Nous utilisons le hachage %vues pour nous souvenir des fonctions déjà appelées. Ceci pourrait se produire si plusieurs classes parentes avaient un ancêtre commun.

Les méthodes qui déclencheraient un AUTOLOAD (décrit dans la section suivante) ne seront pas correctement signalées sauf si le paquetage a déclaré (mais pas défini) les fonctions qu'il souhaite voir autocharger.

INVOQUANT->VERSION(*REQUIS*)

La méthode VERSION retourne le numéro de version de la classe d'*INVOQUANT*, tel qu'il a été enregistré dans la variable $VERSION du paquetage. Si le paramètre *REQUIS* est fourni, elle vérifie que la version courante n'est pas plus petite que *REQUIS*, et lève une exception si elle l'est. C'est la méthode qu'invoque use pour déterminer si un module est suffisamment récent.

```
use Thread 1.0;   # appelle Thread->VERSION(1.0)
print "Exécute la version ", Thread->VERSION, " de Thread.\n";
```

Vous pouvez fournir votre propre méthode VERSION pour supplanter la méthode dans UNIVERSAL. Néanmoins, ceci obligera aussi toute classe dérivée de votre classe à utiliser la méthode supplantée. Si vous ne souhaitez pas que cela se produise, vous devriez concevoir votre méthode de telle sorte qu'elle délègue à UNIVERSAL les requêtes de version des autres classes.

Les méthodes dans UNIVERSAL sont des fonctions internes de Perl, que vous pouvez appeler si vous les qualifiez et leur passez deux paramètres, comme dans UNIVERSAL::isa($obj_form, "HASH"). (Mais ceci n'est pas vraiment conseillé puisque can a normalement la réponse que vous cherchez vraiment.)

Vous êtes libre d'ajouter vos propres méthodes à la classe UNIVERSAL. (Vous devez être très prudent, évidemment ; vous pourriez vraiment gâcher les choses pour quelqu'un qui s'attend à ne *pas* trouver le nom de méthode que vous définissez, peut-être pour qu'il puisse l'autocharger d'un autre endroit.) Ici nous créons une méthode copier que les objets d'une classe quelconque peuvent utiliser s'ils ne l'ont pas définie eux-mêmes. (Elle échoue de façon spectaculaire si elle est invoquée pour une classe au lieu d'un objet.)

```
use Data::Dumper;
use Carp;
sub UNIVERSAL::copier {
    my $self = shift;
    if (ref $self) {
        return eval Dumper($self);  # pas de ref de CODE
    } else {
        confess "UNIVERSAL::copier ne peut pas copier la classe $self";
    }
}
```

Cette utilisation de Data::Dumper ne fonctionne pas si l'objet contient des références aux fonctions, car elle ne peuvent pas être reproduites correctement. Même si le source était disponible, les liens lexicaux seraient perdus.

Autochargement de méthode

Normalement, lorsque vous appelez une fonction non définie dans un paquetage qui définit une fonction AUTOLOAD, la fonction AUTOLOAD est appelée au lieu de lever une exception (voir la section *Autochargement* au chapitre 10). Avec les méthodes, ceci fonctionne un peu différemment. Si la recherche de méthode normale (dans la classe, dans ses ancêtres, et finalement dans UNIVERSAL) échoue, la même séquence est de nouveau exécutée, en cherchant cette fois une fonction AUTOLOAD. Si elle est trouvée, cette fonction est appelée comme méthode, en affectant à la variable $AUTOLOAD du paquetage le nom complet de la fonction au nom de laquelle a été appelé AUTOLOAD.

Vous devez être un peu prudent lorsque vous autochargez des méthodes. Premièrement, la fonction AUTOLOAD doit retourner immédiatement si elle est appelée au nom d'une méthode nommée DESTROY, sauf si votre but était de simuler DESTROY, qui en Perl a une signification spéciale décrite dans la section *Destructeurs d'instance* plus loin dans ce chapitre.

```
sub AUTOLOAD {
    return if our $AUTOLOAD =~ /::DESTROY$/;
    ...
}
```

Deuxièmement, si une classe fournit un filet de sûreté AUTOLOAD, vous ne pourrez pas utiliser UNIVERSAL::can sur un nom de méthode pour vérifier si l'on peut l'invoquer. Vous devez vérifier AUTOLOAD séparément :

```
if ($obj->can("nom_meth") || $obj->can("AUTOLOAD")) {
    $obj->nom_meth();
}
```

Finalement, avec l'héritage multiple, si une classe hérite de deux classes ou plus, chacune ayant un AUTOLOAD, seul celui le plus à gauche sera jamais déclenché, puisque Perl s'arrête dès qu'il trouve le premier AUTOLOAD.

Ces deux dernières tracasseries sont facilement contournées en déclarant les fonctions dans le paquetage dont le AUTOLOAD est censé gérer ces méthodes. Vous pouvez faire ceci, soit avec des déclarations individuelles :

```
package Goblin;
sub taper;
sub mordre;
sub griffer;
```

soit avec le pragma use subs, qui est plus commode si vous avez de nombreuses méthodes à déclarer :

```
package Goblin;
use subs qw(taper mordre griffer);
```

Même si vous n'avez que déclaré ces fonctions sans les définir, c'est assez pour que le système pense qu'elles sont réelles. Elles apparaissent dans la vérification avec UNIVERSAL::can, et plus important encore, elles apparaissent dans l'étape 2 de recherche de méthode, qui n'ira jamais jusqu'à l'étape 3 et encore moins la 4.

« Mais, mais, » vous vous exclamez, « elles invoquent AUTOLOAD, non ? » Eh bien oui, elles finissent par le faire, mais le mécanisme est différent. Ayant trouvé le bout de méthode grâce à l'étape 2, Perl essaie de l'appeler. Lorsqu'il découvre que la méthode n'est pas tout ce qu'elle aurait pu être, la recherche de méthode AUTOLOAD démarre de nouveau, mais cette fois-ci, elle démarre dans la classe contenant le bout de méthode, ce qui restreint la recherche de méthode à la classe et à ses ancêtres (et à UNIVERSAL). Voilà comment Perl trouve le bon AUTOLOAD à exécuter et ignore les AUTOLOAD de la mauvaise partie de l'arbre généalogique.

Méthodes privées

Il existe un moyen d'invoquer une méthode de telle façon que Perl ignore complètement l'héritage. Si au lieu d'utiliser un nom de méthode littéral, vous spécifiez une simple variable scalaire contenant une référence à la fonction, alors la fonction est appelée immédiatement. Dans la précédente description de UNIVERSAL->can, le dernier exemple invoque toutes les méthodes supplantées en utilisant la référence à la fonction, pas son nom.

Un aspect intrigant de ce comportement est qu'il peut être utilisé pour implémenter des appels de méthode privée. Si vous mettez la classe dans un module, vous pouvez utiliser la portée lexicale du fichier pour faire un espace privé. Premièrement, stockez une fonction anonyme dans une variable lexicale dont la portée est le fichier :

```
# déclarer une méthode privée
my $porte_secrete = sub {
    my $self = shift;
    ...
};
```

Plus loin dans ce fichier, vous pouvez utiliser la variable comme si elle contenait un nom de variable. La fermeture sera appelée directement, sans prendre l'héritage en considération. Comme pour toute autre méthode, l'invoquant est passé en paramètre supplémentaire.

```
sub frapper {
    my $self = shift;
    if ($self->{frappe}++ > 5) {
        $self->$porte_secrete();
    }
}
```

Ceci permet aux fonctions appartenant à un fichier (les méthodes de classe) d'invoquer une méthode à laquelle tout code hors de leur portée lexicale ne peut accéder.

Destructeurs d'instance

Comme pour tout autre référent en Perl, lorsque la dernière référence à un objet disparaît, sa mémoire est implicitement recyclée. Avec un objet, vous avez la possibilité de prendre la main juste au moment où ceci est près de se passer en définissant une fonction `DESTROY` dans le paquetage de la classe. Cette méthode est déclenchée automatiquement au moment approprié, avec l'objet prêt à être recyclé comme unique paramètre.

Les destructeurs sont rarement nécessaires en Perl, parce que la mémoire est gérée automatiquement pour vous. Cependant, la mémoire mise à part, certains objets peuvent avoir des informations d'état dont vous pouvez souhaiter vous occuper, comme des handles de fichier ou des connexions à une base de données.

```
package NotifierCourrier;
sub DESTROY {
    my $self = shift;
    my $fh   = $self->{handle_courrier};
    my $id   = $self->{nom};
    print $fh "\n$id s'en va à " . localtime() . "\n";
    close $fh;  # fermer la connexion au système de courrier
}
```

De même que Perl n'utilise qu'une unique méthode pour construire un objet même lorsque la classe du constructeur hérite d'une ou plusieurs autres classes, Perl n'utilise aussi qu'une unique méthode `DESTROY` par objet détruit quel que soit l'héritage. En d'autres termes, Perl ne fait pas de la destruction hiérarchique à votre place. Si votre classe supplante le destructeur d'une surclasse, alors votre méthode `DESTROY` devra peut-être

invoquer la méthode DESTROY d'une classe de base :

```
sub DESTROY {
    my $self = shift;
    # voir s'il y a un destructeur supplanté...
    $self->SUPER::DESTROY if $self->can("SUPER::DESTROY");
    # faites vos choses maintenant, avant ou après
}
```

Ceci ne s'applique qu'aux classes héritées ; un objet qui est simplement *contenu* dans l'objet courant — comme, par exemple, une valeur dans un hachage — sera libéré et détruit automatiquement. C'est une raison pour laquelle l'appartenance par simple agrégation (parfois appelée relation « has-a » ou « possède-un ») est souvent plus propre et plus claire que l'héritage (une relation « est-un »). En d'autres termes, il suffit parfois de stocker un objet directement à l'intérieur d'un autre au lieu d'employer l'héritage, qui peut inutilement ajouter de la complexité. Parfois lorsque des utilisateurs se tournent vers l'héritage multiple, l'héritage simple peut suffire.

Il est possible, mais rarement nécessaire, d'appeler DESTROY explicitement. Cela peut même être dangereux car l'exécution du destructeur plus d'une fois sur un même objet pourrait s'avérer désagréable.

Ramasse-miettes avec les méthodes DESTROY

Comme nous l'avons décrit dans la section *Ramasse-miettes, références circulaires et références faibles* du chapitre 8, une variable qui fait référence à elle-même n'est libérée que quand le programme (ou l'interpréteur intégré) est sur le point de se terminer (il en est de même pour plusieurs variables qui se font référence indirectement entre elles). Si vous voulez récupérer la mémoire plus tôt, vous devez normalement annuler explicitement la référence ou l'affaiblir avec le module WeakRef de CPAN.

Avec les objets, une autre solution est de créer une classe conteneur qui contient un pointeur à la structure de données autoréférentielle. Définissez une méthode DESTROY pour la classe de l'objet conteneur qui annule manuellement les boucles dans la structure autoréférentielle. Vous trouverez un exemple de ceci dans le chapitre 13 du *Perl en action* dans la recette 13.13, *Gérer les structures de données circulaires*.

Lorsqu'un interpréteur se termine, tous ses objets sont détruits, ce qui est important pour les applications Perl multi-threadées ou intégrés dans un autre logiciel. Les objets sont presque toujours détruits lors d'une étape distincte, avant les références ordinaires. Ceci afin d'empêcher les méthodes DESTROY d'utiliser des références qui ont elles-mêmes été détruites. (Et aussi parce que les références ordinaires ne sont soumises au ramasse-miettes que dans les interpréteurs intégrés, puisque terminer un processus est une façon très *rapide* de récupérer les références. Mais terminer n'exécutera pas les destructeurs d'objet, donc Perl le fait d'abord.)

Gestion des données d'instance

La plupart des classes créent des objets qui ne sont essentiellement que des structures de données avec plusieurs champs de données internes (ou variables d'instance) avec des méthodes pour les manipuler.

Les classes Perl héritent de méthodes, pas de données, mais tant que tout accès à l'objet se fait par des appels de méthode, tout se passe très bien. Si vous voulez l'héritage de données, vous devez le réaliser par l'héritage de méthodes. Généralement, ceci n'est pas nécessaire en Perl, car la plupart des classes stockent les attributs de leur objet dans un hachage anonyme. Les données d'instance de l'objet sont contenues dans ce hachage, qui constitue son propre petit espace de noms dans lequel peut tailler toute classe faisant quelque chose avec l'objet. Par exemple, si vous voulez qu'un objet appelé `$ville` possède un champ nommé `altitude`, vous pouvez y accéder simplement avec `$ville->{altitude}`. Aucune déclaration n'est nécessaire. Mais les méthodes d'emballage ont leur utilité.

Supposons que vous voulez implémenter un objet `Personne`. Vous décidez d'avoir un champ appelé « nom », que par curieuse coïncidence vous stockez sous la clef `nom` dans le hachage anonyme qui servira d'objet. Pour profiter de l'encapsulation, les utilisateurs nécessitent des méthodes qui permettent d'accéder à cette variable d'instance sans lever le voile de l'abstraction.

Par exemple, vous pourriez créer une paire d'accesseurs :

```
sub recuperer_nom {
    my $self = shift;
    return $self->{nom};
}

sub affecter_nom {
    my $self     = shift;
    $self->{nom} = shift;
}
```

ce qui permet le code suivant :

```
$lui = Person->new();
$lui->affecter_nom("Frodon");
$lui->affecter_nom( ucfirst($lui->recuperer_nom) );
```

Vous pourriez même combiner les deux méthodes :

```
sub nom {
    my $self = shift;
    if (@_) { $self->{nom} = shift }
    return $self->{nom};
}
```

Ce qui permettrait le code suivant :

```
$lui = Person->new();
$lui->nom("Frodon");
$lui->nom( ucfirst($lui->nom) );
```

L'avantage d'écrire une fonction différente pour chaque variable d'instance (qui pour notre classe `Personne` pourrait être nom, âge, taille et ainsi de suite) est d'être direct, évident et flexible. Le désavantage est que chaque fois que vous voulez une nouvelle classe, vous finissez par définir une ou deux méthodes identiques par variable d'instance. Au début, ceci n'est pas gênant, et vous êtes libre de le faire si vous le souhaitez. Mais si vous préférez la commodité à la flexibilité, vous préférerez peut-être une des techniques décrites dans les sections suivantes.

Notez que nous ferons varier l'implémentation et non l'interface. Si les utilisateurs de votre classe respectent l'encapsulation, vous pourrez passer de manière transparente d'une implémentation à l'autre sans que les utilisateurs le remarquent. (Les membres de la famille de votre arbre d'héritage ne seront peut-être pas aussi tolérants, puisqu'ils vous connaissent beaucoup mieux que ne vous connaissent des étrangers.) Si vos utilisateurs ont pris l'habitude de mettre leur nez ou leurs mains dans les affaires privées de votre classe, le désastre inévitable est de leur faute et ne vous concerne pas. Tout ce que vous pouvez faire c'est respecter votre partie du contrat en gardant la même interface. Essayer d'obliger tout le monde à ne jamais rien faire de mal prendra tout votre temps et toute votre énergie — et en fin de compte échouera de toute façon.

Il est plus difficile de s'occuper des membres de la famille. Si une sous-classe supplante l'accesseur d'attribut d'une surclasse, doit-elle ou non accéder au même champ dans le hachage ? Les deux approches sont défendables, selon la nature de l'attribut. Pour être plus sûr dans le cas général, chaque accesseur peut préfixer le nom de champ de hachage avec son propre nom de classe, de telle façon que la sous-classe et la surclasse puissent chacune avoir leur propre version. Plusieurs des exemples ci-dessous, dont le module `Class::Struct`, utilisent cette stratégie « résistante aux sous-classes ». Vous verrez des accesseurs qui ressemblent à ceci :

```
sub nom {
    my $self = shift;
    my $champ = __PACKAGE__ . "::nom";
    if (@_) { $self->{$champ} = shift }
    return $self->{$champ};
}
```

Dans chacun des exemples suivants, nous créons une classe `Personne` simple avec les champs `nom`, `race`, et `surnoms`, chacun avec une interface identique mais une implémentation complètement différente. Nous n'allons pas vous dire laquelle nous préférons, car nous les préférons toutes, selon l'occasion. Et tous les goûts sont dans la nature. Certains préfèrent le ragoût de lapin, d'autres le poisson.

Déclarations de champ avec use fields

Un objet n'est pas obligatoirement implémenté avec un hachage anonyme. N'importe quelle référence suffira. Par exemple, si vous utilisiez un tableau anonyme, vous pourriez définir un constructeur comme ceci :

```
sub new {
    my $invoquant = shift;
    my $classe = ref($invoquant) || $invoquant;
    return bless [], $classe;
}
```

et des accesseurs comme ceci :

```
sub nom {
    my $self = shift;
    if (@_) { $self->[0] = shift }
    return $self->[0];
}
```

```
sub race {
    my $self = shift;
    if (@_) { $self->[1] = shift }
    return $self->[1];
}

sub surnoms {
    my $self = shift;
    if (@_) { $self->[2] = shift }
    return $self->[2];
}
```

Les tableaux sont un peu plus rapides d'accès que les hachages et occupent un peu moins de mémoire, mais ils ne sont pas terriblement commodes à utiliser. Vous devez tenir compte des numéros d'index (pas juste dans votre classe, mais aussi dans votre surclasse), qui doivent d'une façon ou d'un autre indiquer quelles parties du tableau utilise votre classe. Sinon, vous pourriez réutiliser une case.

Le pragma `use fields` traite tous ces points :

```
package Personne;
use fields qw(nom race surnoms);
```

Ce pragma ne vous crée pas les méthodes accesseurs, mais il dépend de magie interne (appelée *pseudo-hachages*) pour faire quelque chose de similaire. (Vous souhaiterez peut-être tout de même emballer les champs dans des accesseurs, comme nous le faisons dans l'exemple suivant.) Les pseudo-hachages sont des références de tableau que vous pouvez utiliser comme des hachages parce qu'ils sont associés avec un tableau qui met les clefs en correspondance avec les index. Le pragma `use fields` vous met en place cette table de correspondance, déclarant effectivement quels champs sont valides pour l'objet `Personne` ; ceci les fait prendre en compte par le compilateur Perl. Si vous déclarez le type de votre variable objet (comme avec `my Personne $self` dans l'exemple suivant), le compilateur est suffisamment intelligent pour optimiser l'accès aux champs avec des simples accès par tableau. De façon peut-être plus importante, il fait à la compilation de la validation de type pour les noms de champ (c'est en quelque sorte une validation de typographie contre les fautes de frappe). (Voir la section *Pseudo-hachages* au chapitre 8.)

Un constructeur et des exemples d'accesseurs ressembleraient à ceci :

```
package Personne;
use fields qw(nom race surnoms);
sub new {
    my $type = shift;
    my Personne $self = fields::new(ref $type || $type);
    $self->{nom} = "pas de nom";
    $self->{race}  = "inconnue";
    $self->{surnoms} = [];
    return $self;
}
sub nom {
    my Personne $self = shift;
    $self->{nom} = shift if @_;
    return $self->{nom};
}
```

```
sub race {
    my Personne $self = shift;
    $self->{race} = shift if @_;
    return $self->{race};
}
sub surnoms {
    my Personne $self = shift;
    $self->{surnoms} = shift if @_;
    return $self->{surnoms};
}
1;
```

Si vous écrivez incorrectement l'une des clefs littérales utilisées pour accéder au pseudo-hachage, vous n'aurez pas à attendre l'exécution pour vous en rendre compte. Le compilateur sait à quel type `$self` est censé faire référence (parce que vous le lui avez dit), donc il peut vérifier que le code n'accède qu'à des champs réellement existants dans les objets `Personne`. S'il arrive que vos doigts s'emballent et que vous tentez d'accéder à un champ inexistant (comme `$self->{mon}`), le compilateur peut tout de suite signaler cette erreur et ne laissera jamais exécuter par l'interpréteur le programme erroné.

Il reste un peu de redite dans la déclaration de méthodes pour accéder aux variables d'instance, donc vous souhaiterez peut-être encore automatiser la création de méthodes accesseur simples avec l'une des techniques ci-dessous. Toutefois, comme ces techniques utilisent toutes une forme d'indirection ou une autre, vous perdrez l'avantage de la vérification à la compilation de fautes de frappe pour les accès typés lexicalement aux hachages. Il vous restera néanmoins les (légères) économies de temps et d'espace.

Si vous décidez d'utiliser un pseudo-hachage pour implémenter votre classe, toute classe qui en hérite doit prendre en compte cette implémentation sous-jacente. Si un objet est implémenté avec un pseudo-hachage, tous les membres de la hiérarchie d'héritage doivent utiliser les déclarations `use base` et `use fields`. Par exemple :

```
package Magicien;
use base "Personne";
use fields qw(baton couleur sphere);
```

Ceci fait du module `Magicien` une sous-classe de la classe `Personne`, et charge le fichier *Personne.pm*. Ceci déclare également trois nouveaux champs dans cette classe, qui s'ajoutent à ceux de `Personne`. Ainsi lorsque vous écrirez :

```
my Magicien $mage = fields::new("Magicien");
```

vous aurez un objet pseudo-hachage avec accès aux champs des deux classes :

```
$mage->nom("Gandalf");
$mage->couleur("Gris");
```

Puisque toutes les sous-classes doivent savoir qu'elles utilisent une implémentation de pseudo-hachage, elles doivent utiliser la notation directe de pseudo-hachage pour des raisons d'efficacité et de validation des types :

```
$mage->{nom} = "Gandalf";
$mage->{couleur} = "Gris";
```

Toutefois, si vous voulez garder des implémentations interchangeables, les utilisateurs extérieurs de votre classe doivent utiliser les méthodes accesseurs.

Bien que `use base` ne permette que l'héritage simple, ceci est rarement une contrainte sévère. Voir les descriptions de `use base` et de `use fields` au chapitre 31.

Génération de classes avec Class::Struct

Le module standard `Class::Struct` exporte une fonction appelée `struct`. Celle-ci crée tout ce qu'il vous faut pour commencer l'écriture d'une classe entière. Elle génère un constructeur appelé `new`, ainsi que des méthodes accesseurs pour chacun des champs (variables d'instance) nommés dans cette structure.

Par exemple, si vous mettez la classe dans un fichier *Personne.pm* :

```
package Personne;
use Class::Struct;
struct Personne => {  # créer la définition d'une "Personne"
    nom     => '$',   #    le champ nom est un scalaire
    race    => '$',   #    le champ race est aussi un scalaire
    surnoms => '@',   #    mais le champ surnoms est une réf de tableau
};
1;
```

Alors vous pourriez utiliser le module de cette façon :

```
use Personne;
my $mage = Personne->new();
$mage->nom("Gandalf");
$mage->race("Istar");
$mage->surnoms( ["Mithrandir", "Olorin", "Incanus"] );
```

Le module `Class::Struct` a créé ces quatre méthodes. Comme il respecte le principe « résistant aux sous-classes » de toujours préfixer le nom du champ avec le nom de la classe, il permet aussi sans conflit à une sous-classe d'avoir son propre champ séparé de même nom qu'un champ de la classe de base. Cela veut dire qu'il utilise ici « `Personne::nom` » au lieu du simple « `nom` » comme clef de hachage pour cette variable d'instance.

Les champs d'une déclaration `struct` ne sont pas obligatoirement des types Perl de base. Ils peuvent aussi spécifier d'autres classes, mais les classes créées avec `struct` fonctionnent le mieux car la fonction fait des présuppositions sur le comportement des classes qui ne sont pas généralement vraies de toutes les classes. Par exemple, la méthode `new` de la classe appropriée est invoquée pour initialiser le champ, mais beaucoup de classes ont des constructeurs nommés différemment.

Voir la description de `Class::Struct` au chapitre 32, *Modules standards*, et sa documentation en ligne pour plus d'information. Beaucoup de modules standard utilisent `Class::Struct` pour implémenter leurs classes, dont notamment `User::pwent` et `Net::hostent`. La lecture de leur code peut s'avérer instructive.

Génération d'accesseurs par autochargement

Comme nous l'avons mentionné précédemment, lorsque vous invoquez une méthode inexistante, Perl cherche une méthode `AUTOLOAD` de deux façons différentes, selon que vous ayez déclaré la méthode ou non. Vous pouvez utiliser cette propriété pour donner

accès aux données d'instance de l'objet sans définir une fonction différente pour chaque variable d'instance. Dans la fonction AUTOLOAD, le nom de la méthode réellement invoquée peut être déterminé à partir de la variable $AUTOLOAD. Examinons le code suivant :

```
use Personne;
$lui = Personne->new;
$lui->nom("Aragorn");
$lui->race("Homme");
$lui->surnoms( ["Grands-Pas", "Estel", "Elessar"] );
printf "La race de %s est : %s.\n", $lui->nom, $lui->race;
print "Ses surnoms sont : ", join(", ", @{$lui->surnoms}), ".\n";
```

Comme précédemment, cette classe Personne implémente une structure de données avec trois champs : nom, race, et surnoms :

```
package Personne;
use Carp;

my %Fields = (
    "Personne::nom"  => "pas de nom",
    "Personne::race"   => "inconnue",
    "Personne::surnoms"  => [],
);

# Cette déclaration nous assure d'obtenir notre propre autochargeur
use subs qw(nom race surnoms);

sub new {
    my $invoquant = shift;
    my $classe = ref($invoquant) || $invoquant;
    my $self  = { %Fields, @_ };    # cloner comme Class::Struct
    bless $self, $classe;
    return $self;
}

sub AUTOLOAD {
    my $self = shift;
    # ne gérer que les méthode d'instance, pas les méthodes de classe
    croak "$self n'est pas un objet" unless ref($self);
    my $nom = our $AUTOLOAD;
    return if $nom =~ /::DESTROY$/;
    unless (exists $self->{$nom}) {
        croak "impossible d'accéder au champ ‘$nom’ dans $self";
    }
    if (@_) { return $self->{$nom} = shift }
    else    { return $self->{$nom} }
}
```

Comme vous le voyez, il n'y a aucune mention de méthodes nommées nom, race ou surnoms. La fonction AUTOLOAD prend tout cela en charge. Lorsque quelqu'un utilise $lui->nom("Aragorn"), la fonction AUTOLOAD est appelée avec la valeur « Personne::nom » affectée à $AUTOLOAD. Commodément, le nom qualifié est exacte-

ment sous la forme nécessaire pour accéder au champ du hachage de l'objet. Ainsi si vous utilisez cette classe à l'intérieur d'une hiérarchie de classes, vous n'entrerez pas en conflit avec l'utilisation du même nom dans d'autres classes.

Génération d'accesseurs par fermetures

La plupart des méthodes accesseurs font essentiellement la même chose : elles ne font que récupérer ou stocker une valeur dans une variable d'instance. En Perl, la façon la plus naturelle de créer une famille de fonctions quasiment identiques est de boucler autour d'une fermeture. Mais une fermeture est une fonction anonyme, et une méthode, pour qu'elle puisse être appelée par nom, doit être une fonction nommée dans la table de symboles du paquetage de la classe. Ceci n'est pas un problème — il suffit d'affecter la référence à la fermeture à un typeglob de nom approprié.

```
package Personne;

sub new {
    my $invoquant = shift;
    my $self = bless({}, ref $invoquant || $invoquant);
    $self->init();
    return $self;
}

sub init {
    my $self = shift;
    $self->nom("pas de nom");
    $self->race("inconnue");
    $self->surnoms([]);
}

for my $champ (qw(nom race surnoms)) {
    my $slot = __PACKAGE__ . "::$champ";
    no strict "refs";   # Pour que fonctionne la réf symbolique au typeglob
    *$slot = sub {
        my $self = shift;
        $self->{$champ} = shift if @_;
        return $self->{$champ};
    };
}
```

Les fermetures sont la façon « fait main » la plus propre de créer une multitude de méthodes accesseurs pour vos données d'instance. C'est une méthode efficace à la fois pour l'ordinateur et pour vous. Non seulement tous les accesseurs partagent le même bout de code (ils ne nécessitent que leur propre emplacement lexical), mais plus tard si vous décidez d'ajouter un autre attribut, les changements requis sont minimaux : il suffit d'ajouter un mot de plus à la liste de la boucle `for`, et peut-être quelque chose à la méthode `init`.

Utilisation de fermetures pour objets privés

Jusqu'ici, ces techniques de gestion de données d'instance n'ont proposé aucun mécanisme pour les « protéger » d'accès externes. Toute personne en dehors de la classe peut ouvrir la boîte noire de l'objet et fouiner dedans — si elle n'a pas peur d'annuler la garantie. Imposer un espace privé obligatoire tend à mettre des bâtons dans les roues de personnes qui essaient simplement de faire leur travail. Perl part du principe qu'il est mieux d'encapsuler ses données avec un écriteau sur lequel il est écrit :

```
EN CAS D'INCENDIE
 BRISER LA GLACE
```

Vous devez si possible respecter l'encapsulation, tout en gardant un accès aisé au contenu en cas d'urgence, comme pour le débogage.

Mais si vous souhaitez imposer un espace privé, Perl ne va pas vous en empêcher. Perl fournit des briques de base de bas niveau dont vous pouvez faire usage pour entourer votre classe et ses objets d'un bouclier impénétrable d'espace privé — un bouclier en fait plus fort que celui dont on dispose dans de nombreux langages orientés objet. Les portées lexicales et les variables lexicales qu'ils contiennent en sont les éléments principaux, et les fermetures y jouent un rôle essentiel.

Dans la section *Méthodes privées*, nous avons vu comment une classe peut utiliser les fermetures pour implémenter des méthodes qui sont invisibles à l'extérieur du fichier de module. Plus loin nous verrons des méthodes accesseurs qui régulent des données de classe si privées que même le reste de la classe n'y a pas un accès illimité. Ces techniques restent des utilisations assez traditionnelles des fermetures. L'approche véritablement intéressante est d'utiliser une fermeture pour l'objet lui-même. Les variables d'instance de l'objet sont verrouillées à l'intérieur d'une portée à laquelle seul l'objet — c'est-à-dire, la fermeture — a librement accès. Ceci est une forme très forte d'encapsulation ; elle empêche non seulement toute altération de l'extérieur, mais aussi d'autres méthodes de la même classe doivent utiliser les méthodes d'accès appropriées pour accéder aux données d'instance de l'objet.

Voici un exemple de comment cela pourrait fonctionner. Nous utiliserons des fermetures à la fois pour les objets eux-mêmes et pour les accesseurs générés :

```
package Personne;
sub new {
    my $invoquant  = shift;
    my $classe = ref($invoquant) || $invoquant;
    my $donnees = {
       NOM      => "pas de nom",
       RACE     => "inconnue",
       SURNOMS  => [],
    };
    my $self = sub {
       my $champ = shift;
       ##########################################
       ### METTRE LES VERIFICATIONS D'ACCES ICI ###
       ##########################################
       if (@_) { $donnees->{$champ} = shift }
       return    $donnees->{$champ};
```

```
    };
    bless($self, $classe);
    return $self;
}
# générer les noms de méthodes
for my $champ (qw(nom race surnoms)) {
    no strict "refs";  # pour l'accès à la table de symboles
    *$champ = sub {
        my $self = shift;
        return $self->(uc $champ, @_);
    };
}
```

L'objet créé et retourné par la méthode `new` n'est plus un hachage, comme c'était le cas des autres constructeurs que nous avons vus. C'est une fermeture avec accès unique aux données des attributs stockés dans le hachage référencé par `$donnees`. Une fois l'appel au constructeur terminé, l'accès à `$donnees` (et donc aux attributs) ne se fait plus que par la fermeture.

Dans un appel comme `$lui->nom("Bombadil")`, l'objet invoquant stocké dans `$self` est la fermeture consacrée et retournée par le constructeur. On ne peut pas faire grand-chose avec une fermeture sinon l'appeler, donc c'est ce que nous faisons avec `$self->(uc $champ, @_)`. Ne vous laissez pas tromper par la flèche : ceci n'est qu'un appel de fonction indirect, pas une invocation de méthode. Le paramètre initial est la chaîne « `nom` », et d'autres paramètres peuvent être passés à sa suite.[7] Une fois qu'on s'exécute à l'intérieur de la fermeture, la référence de hachage dans `$donnees` est à nouveau accessible. La fermeture est alors libre de permettre ou d'interdire l'accès à ce qui lui plaît.

Personne en dehors de l'objet fermeture n'a d'accès direct à ces données d'instance extrêmement privées, pas même les autres méthodes de la classe. Elles pourraient essayer d'appeler la fermeture de la même façon que le font les méthodes générées par la boucle `for`, définissant éventuellement une variable d'instance dont la classe n'aurait jamais entendu parler. Mais cette approche est aisément bloquée en insérant divers bouts de code dans le constructeur à l'endroit du commentaire sur les vérifications d'accès. D'abord, nous avons besoin d'un préambule commun :

```
use Carp;
local $Carp::CarpLevel = 1;  # Afficher des messages croak courts
my ($pack_appelant, $fichier_appelant) = caller();
```

Ensuite nous faisons chacune des vérifications. La première s'assure que le nom d'attribut existe :

```
croak "Pas de champ valide '$champ' dans l'object"
    unless exists $donnees->{$champ};
```

Celle-ci ne permet l'accès qu'aux appelants du même fichier :

```
carp "Accès direct interdit aux fichiers étrangers"
    unless $fichier_appelant eq __FILE__;
```

7. Bien sûr, le double appel de fonction est lent, mais si vous vouliez que ce soit rapide, auriez vous vraiment choisi d'utiliser des objets ?

Celle-ci ne permet l'accès qu'aux appelants du même paquetage :

```
carp "Accès direct interdit au paquetage étranger ${pack_appelant}::"
    unless $pack_appelant eq __PACKAGE__;
```

Et celle-ci ne permet l'accès qu'aux appelants dont la classe hérite de la nôtre :

```
carp "Accès direct interdit à la classe inconnue ${pack_appelant}::"
    unless $pack_appelant->isa(__PACKAGE__);
```

Toutes ces vérifications ne font qu'empêcher l'accès direct. Les utilisateurs d'une classe qui, poliment, n'emploient que les méthodes de classes prévues ne souffrent pas d'une telle restriction. Perl vous donne les outils pour être aussi maniaque que vous le souhaitez. Heureusement, peu de gens souhaitent être vraiment maniaques.

Mais certaines personnes devraient l'être. C'est bien d'être maniaque lorsqu'on écrit un logiciel de calculateur de vol. Si vous voulez être ou devriez être une telle personne, et que vous préférez utiliser du code fonctionnel au lieu de tout réinventer vous-même, jetez un coup d'œil sur CPAN au module `Tie::SecureHash` de Damian Conway. Il implémente des hachages restreints qui permettent des maniaqueries publiques, protégées ou privées. Damian a aussi écrit un module encore plus ambitieux, `Class::Contract`, qui impose un régime d'ingénierie logicielle formelle au système objet flexible de Perl. La liste des fonctionnalités de ce module ressemble à la liste de contrôle d'un livre écrit par un professeur d'ingénierie logicielle[8], avec notamment l'encapsulation obligatoire, l'héritage statique, et la vérification de conditions de conception par contrat pour le Perl orienté objet, avec une syntaxe déclarative pour les définitions d'attribut, de méthode, de constructeur, et de destructeur, à la fois au niveau de l'objet et de la classe, et des préconditions, des post-conditions, et des invariants de classe. Ouf !

Nouvelles astuces

Depuis la version 5.6 de Perl, vous pouvez aussi déclarer une méthode pour indiquer qu'elle retourne une lvalue. Ceci se fait avec l'attribut de fonction lvalue (à ne pas confondre avec les attributs d'objet). Cette fonctionnalité expérimentale vous permet de traiter une méthode comme quelque chose qui apparaîtrait du côté gauche d'un signe égal :

```
package Bestiau;

sub new {
    my $classe = shift;
    my $self = { petits => 0, @_ }; # Supplanter la valeur par défaut.
    bless $self, $classe;
}

sub petits : lvalue {               # Nous affecterons à petits() plus loin.
    my $self = shift;
    $self->{petits};
}
```

8. Pouvez-vous deviner la profession de Damian ? À propos, nous vous recommandons fortement son livre, *Object Oriented Perl* (Manning Publications, 1999).

```
package main;
$vermine = Bestiau->new(petits => 4);
$vermine->petits *= 2;              # Affecter à $vermine->petits !
$vermine->petits =~ s/(.)/$1$1/;    # Modifier $vermine->petits sur place !
print $vermine->petits;             # Nous avons maintenant 88 petits.
```

Ceci vous laisse faire semblant que `$vermine->petits` est une variable tout en respectant l'encapsulation. Voir la section *L'attribut lvalue* du chapitre 6, *Sous-programmes*.

Si vous exécutez une version de Perl avec threads et que vous voulez vous assurer qu'un seul thread n'appelle une méthode donnée pour un objet, vous pouvez faire cela avec les attributs `locked` et `method` :

```
sub petits : locked method {
    ...
}
```

Lorsqu'un thread invoque la méthode `petits` pour un objet, Perl verrouille l'objet avant exécution, empêchant d'autres threads de faire la même chose. Voir la section *Les attibuts locked et method* du chapitre 6.

Gestion des données de classe

Nous avons examiné plusieurs approches pour accéder aux données associées à un objet. Mais parfois vous voulez un état commun partagé par tous les objets d'une classe. Au lieu d'être seulement un attribut d'une instance de la classe, ces variables sont globales pour la classe entière, quelle que soit l'instance de classe (l'objet) que vous utilisez pour y accéder. (Les programmeurs C++ les appelleraient des membres statiques de la classe.) Voici des situations où ces variables de classe seraient commodes :

- Pour compter au fur et à mesure tous les objets créés, ou tous les objets en cours d'utilisation.
- Pour garder la liste de tous les objets, liste que vous pouvez parcourir.
- Pour stocker le nom ou le descripteur d'un fichier log utilisé par une méthode de débogage de toute la classe.
- Pour recenser des données accumulées, comme la somme totale de liquide distribuée en une journée par tous les distributeurs de billets d'un réseau.
- Pour suivre le dernier objet créé par une classe ou l'objet le plus utilisé.
- Pour gérer un cache des objets en mémoire qui ont déjà été reconstitués à partir d'une mémoire persistante.
- Pour fournir une table associative inversée permettant de trouver un objet à partir de la valeur de l'un de ses attributs.

Il reste à décider où stocker l'état de ces attributs partagés. Perl n'a pas de mécanisme syntaxique particulier pour déclarer les attributs de classe, pas plus qu'il n'en a pour les attributs d'instance. Perl fournit au développeur un large ensemble de fonctionnalités puissantes mais flexibles qui peuvent être adaptées habilement aux exigences particulières d'une situation. Vous pouvez donc sélectionner le mécanisme qui correspond le mieux à la situation donnée au lieu de devoir vivre avec les choix de conception de quel-

qu'un d'autre. Ou alors, vous pouvez vivre avec les choix de conception que quelqu'un d'autre a empaqueté et mis sur CPAN. Comme toujours, il y a plus d'une façon de le faire, TMTOWTDI.

Comme pour tout ce qui a trait à une classe, on ne doit pas accéder directement aux données de classe, particulièrement de l'extérieur de l'implémentation de la classe elle-même. Ce n'est guère respecter l'encapsulation que de soigneusement créer des méthodes accesseurs pour les variables d'instance mais ensuite d'inviter le public à tripoter directement vos variables de classe, en affectant par exemple `$UneClasse::Debug = 1`. Pour établir une barrière bien claire entre interface et implémentation, vous pouvez, pour manipuler les données de classe, créer des méthodes accesseurs similaires à celles que vous employez pour les données d'instance.

Imaginons que nous voulions connaître à tout moment le compte de la population mondiale des objets `Bestiau`. Nous rangeons ce nombre dans une variable de paquetage, mais nous fournissons une méthode appelée `population` pour que les utilisateurs de la classe n'aient pas à connaître son implémentation.

```
Bestiau->population()       # Accès par le nom de classe
$gollum->population()       # Accès par l'instance
```

Comme en Perl une classe n'est qu'un paquetage, une variable de paquetage est l'endroit le plus naturel pour ranger une donnée de classe. Voici une implémentation simple d'une telle classe. La méthode `population` ne tient pas compte de son invoquant et ne fait que retourner la valeur courante de la variable de paquetage `$Population`. (Certains programmeurs aiment commencer leurs variables globales par une majuscule.)

```
package Bestiau;
our $Population = 0;
sub population { return $Population; }
sub DESTROY { $Population-- }
sub engendrer {
    my $invoquant = shift;
    my $classe = ref($invoquant) || $invoquant;
    $Population++;
    return bless { nom => shift || "anon" }, $classe;
}
sub nom {
    my $self = shift;
    $self->{nom} = shift if @_;
    return $self->{nom};
}
```

Si vous voulez créer des méthodes de données de classe qui fonctionnent comme les accesseurs de données d'instance, faites ceci :

```
our $Deboguage = 0;     # donnée de classe
sub debug {
    shift;              # on ignore intentionnellement l'invoquant
    $Deboguage = shift if @_;
    return $Deboguage;
}
```

Maintenant vous pouvez spécifier le niveau de débogage global avec la classe ou avec l'une quelconque de ses instances.

Comme `$Deboguage` est une variable de paquetage, elle est globalement accessible. Mais si vous changez la variable `our` en variable `my`, seul peut la voir le code situé à sa suite dans le fichier. Vous pouvez en faire encore plus — vous pouvez même restreindre l'accès aux attributs de classe au reste de la classe elle-même. Emballez la déclaration de variable dans une portée de bloc :

```
{
    my $Deboguage = 0;          # donnée de classe à portée lexicale
    sub debug {
        shift;                  # on ignore intentionnellement l'invoquant
        $Deboguage = shift if @_;
        return $Deboguage;
    }
}
```

Maintenant personne n'a le droit de lire ou d'écrire les attributs de classe sans utiliser la méthode accesseur, puisque seule cette fonction est dans la même portée que la variable et y a accès.

Si une classe dérivée hérite de ces accesseurs de classe, ils accèdent tout de même aux données d'origine, que ces variables soient déclarées avec `our` ou avec `my`. Les données ne sont pas relatives au paquetage. Vous pouvez considérer que les accesseurs s'exécutent dans la classe où ils ont été définis à l'origine, pas dans la classe qui les invoque.

Pour certaines données de classe, cette approche fonctionne correctement, mais pour d'autres non. Supposons que nous créions une sous-classe `Ouargue` de `Bestiau`. Si nous voulons garder séparées nos populations, `Ouargue` ne peut hériter de la méthode population de `Bestiau`, car cette méthode telle qu'elle est écrite retourne la valeur de `$Bestiau::Population`.

Il vous faudra décider selon le cas de l'utilité de rendre les attributs de classe relatifs au paquetage. Si vous voulez des attributs relatifs au paquetage, utilisez la classe de l'invoquant pour situer le paquetage contenant les données de classe :

```
sub debug {
    my $invoquant = shift;
    my $classe   = ref($invoquant) || $invoquant;
    my $nom_var = $classe . "::Deboguage";
    no strict "refs";             # pour accéder symboliquement
                                  #aux données de p aquetage
    $$nom_var = shift if @_;
    return $$nom_var;
}
```

Nous annulons temporairement le pragma strict references car sans cela nous ne pourrions pas utiliser le nom symbolique qualifié de la variable globale de paquetage. Ceci est parfaitement raisonnable : puisque toutes les variables de paquetage se trouvent par définition dans la table de symboles du paquetage, il n'y a pas de mal à y accéder par la table de symboles de ce paquetage.

Une autre approche consiste à rendre disponible tout ce dont a besoin un objet — y compris ses données globales de classe — par l'objet (ou passé en tant que paramètre). Pour ce faire, vous aurez souvent à créer un constructeur spécifique pour chaque classe ou au moins une fonction d'initialisation spécifique à appeler par le constructeur. Dans

le constructeur ou l'initialiseur, vous stockez des références aux données de classe directement dans l'objet lui-même, pour ne pas devoir les chercher plus tard. Les méthodes accesseurs utilisent l'objet pour trouver une référence aux données.

Au lieu de mettre la complexité de la recherche des données de classe dans chaque méthode, laissez simplement dire par l'objet à la méthode où se situent les données. Cette approche ne fonctionne bien que lorsque les méthodes d'accès aux données de classe sont invoquées comme méthodes d'instance, car les données de classe pourraient être dans des variables lexicales inaccessibles à partir du nom d'un paquetage.

De quelque façon que l'on s'y prenne, les données de classes relatives à un paquetage sont toujours un peu incommodes. C'est vraiment plus propre si, lorsque vous héritez d'une méthode d'accès à une donnée de classe, vous héritez aussi de la donnée d'état à laquelle elle accède. Voir la page man *perltootc* pour de nombreuses approches plus élaborées de la gestion des données de classe.

Résumé

C'est à peu près tout ce qu'il y a à savoir, sauf tout le reste. Maintenant vous n'avez plus qu'à aller acheter un livre sur la méthodologie de la conception orientée objet et vous cogner le front avec durant six mois environ.

13

Surcharge

Les objets sont chics, mais parfois ils sont juste un peu *trop* chics. Parfois l'on préférerait qu'ils se comportent un peu moins comme des objets et un peu plus comme des types de données ordinaires. Mais voilà le hic : les objets sont des référents représentés par des références, et celles-ci ne sont guère utiles sauf comme références. Vous ne pouvez pas ajouter des références, ou les afficher, ou leur appliquer (utilement) la plupart des opérateurs prédéfinis de Perl. La seule chose que vous puissiez faire c'est les déréférencer. Donc vous finissez par écrire beaucoup d'invocations de méthode explicites, comme ceci :

```
print $objet->comme_chaine;
$nouvel_objet = $sujet->ajouter($objet);
```

De telles références explicites sont en général une bonne chose ; vous ne devez jamais confondre vos références avec vos référents, sauf quand vous voulez les confondre. C'est le cas qui se présente ici. Si vous concevez votre classe avec la *surcharge*, vous pouvez faire comme si les références n'étaient pas là et dire simplement :

```
print $objet;
$nouvel_objet = $sujet + $objet;
```

Lorsque vous surchargez l'un des opérateurs prédéfinis de Perl, vous définissez comment il se comporte lorsqu'il est appliqué aux objets d'une classe donnée. Un certain nombre de modules Perl standard utilisent la surcharge, comme `Math::BigInt`, qui vous permet de créer des objets `Math::BigInt` se comportant comme des entiers ordinaires mais sans limite de taille. Vous pouvez les ajouter avec `+`, les diviser avec `/`, les comparer avec `<=>`, et les afficher avec `print`.

Notez que la surcharge diffère de l'autochargement, qui charge à la demande une fonction manquante. Il faut également distinguer entre surcharger et supplanter, qui masque une fonction avec une autre. La surcharge ne cache rien ; elle ajoute un sens à une opération qui n'aurait pas de sens sur une simple référence.

Le pragma overload

Le pragma `use overload` implémente la surcharge d'opérateur. Vous lui fournissez une liste de couples clef/valeur d'opérateurs et de leur comportement associé :

```
package MaClasse;

use overload   '+' => \&mon_addition,     # réf de code
               '<' => "inferieur_a",      # méthode nommée
             'abs' => sub { return @_ };  # fonction anonyme
```

Maintenant lorsque vous ajoutez deux objets `MaClasse`, la fonction `mon_addition` est appelée pour créer le résultat.

Lorsque vous comparez deux objets `MaClasse` avec l'opérateur `<`, Perl remarque que le comportement est spécifié par une chaîne et interprète la chaîne comme le nom d'une méthode et non comme un simple nom de fonction. Dans l'exemple ci-dessus, la méthode `inferieur_a` pourrait être fournie par le paquetage `MaClasse` lui-même ou héritée d'une classe de base de `MaClasse`, mais la fonction `mon_addition` doit être fournie dans le paquetage courant. La fonction anonyme pour `abs` est fournie encore plus directement. De quelque manière que ces fonctions soient fournies, nous les appellerons des *handlers*.

Dans le cas des opérateurs unaires (ceux qui ne prennent qu'un seul opérande, comme `abs`), le handler spécifié pour une classe est invoqué à chaque fois que l'opérateur est appliqué à un objet de cette classe.

Dans le cas des opérateurs binaires comme `+` ou `<`, le handler est invoqué lorsque le premier opérande est un objet de la classe *ou* lorsque le deuxième opérande est un objet de la classe et le premier opérande n'a pas de comportement de surcharge. C'est pour que vous puissiez dire soit :

```
$objet + 6
```

soit :

```
6 + $objet
```

sans avoir à vous soucier de l'ordre des opérandes. (Dans le deuxième cas, les opérandes seront *permutés* lorsqu'ils seront passés au handler.) Si notre expression était :

```
$animal + $legume
```

et `$animal` et `$legume` étaient des objets de classes différentes, chacune utilisant la surcharge, c'est le comportement de surcharge de `$animal` qui serait déclenché. (Nous espérons que l'animal aime les légumes.)

Il n'y a qu'un seul opérateur trinaire (ternaire) en Perl, `?:`, et vous ne pouvez pas le surcharger. Heureusement.

Handlers de surcharge

Lorsqu'un opérateur surchargé s'exécute, le handler correspondant est invoqué avec trois paramètres. Les deux premiers paramètres sont les deux opérandes. Si l'opérateur n'utilise qu'un opérande, le deuxième paramètre est `undef`.

Le troisième paramètre indique si les deux premiers paramètres sont permutés. Même selon les règles de l'arithmétique ordinaire, certaines opérations ne se soucient pas de l'ordre de leurs paramètres, comme l'addition et la multiplication, alors que d'autres si, comme la soustraction ou la division.[1] Considérez la différence entre :

```
$objet - 6
```

et :

```
6 - $objet
```

Si les deux premiers paramètres d'un handler ont été permutés, le troisième paramètre sera vrai. Sinon, le troisième paramètre sera faux, auquel cas on fera aussi une distinction plus fine : si le handler a été déclenché par un autre handler en rapport avec l'affectation (si par exemple += appelle + pour déterminer comment faire une addition), alors le troisième paramètre n'est pas simplement faux, mais `undef`. Cette distinction permet certaines optimisations.

Par exemple, voici une classe qui vous permet de manipuler un intervalle borné d'entiers. Elle surcharge à la fois + et - de manière à contraindre le résultat de l'addition ou de la soustraction d'objets à être une valeur dans l'intervalle de 0 à 255 :

```
package OctetBorne;

use overload '+' => \&ajouter_borne,
             '-' => \&soustraire_borne;

sub new {
    my $classe = shift;
    my $valeur = shift;
    return bless \$valeur => $classe;
}

sub ajouter_borne {
    my ($x, $y) = @_;
    my ($valeur) = ref($x) ? $$x : $x;
    $valeur     += ref($y) ? $$y : $y;
    $valeur = 255 if $valeur > 255;
    $valeur =   0 if $valeur < 0;
    return bless \$valeur => ref($x);
}

sub soustraire_borne {
    my ($x, $y, $swap) = @_;
    my ($valeur) = (ref $x) ? $$x : $x;
    $valeur     -= (ref $y) ? $$y : $y;
```

1. Vos objets surchargés n'ont pas l'obligation de respecter les règles de l'arithmétique ordinaire, bien entendu, mais il est habituellement conseillé de ne pas surprendre son monde. Curieusement, de nombreux langages font l'erreur de surcharger + avec la concaténation de chaînes, qui n'est pas commutative et qui n'est que vaguement additive. Pour une approche différente, voir Perl.

```
    if ($swap) { $valeur = -$valeur }
    $valeur = 255 if $valeur > 255;
    $valeur =   0 if $valeur < 0;
    return bless \$valeur => ref($x);
}

package main;

$octet1 = OctetBorne->new(200);
$octet2 = OctetBorne->new(100);

$octet3 = $octet1 + $octet2;    # 255
$octet4 = $octet1 - $octet2;    # 100
$octet5 = 150 - $octet2;        # 50
```

Vous noterez que chacune des fonctions est ici par nécessité un constructeur, donc chacune s'assure de consacrer (avec `bless`) son nouvel objet dans la classe courante, quelle qu'elle soit ; nous supposons que la classe peut être héritée. Nous supposons aussi que si `$y` est une référence, c'est une référence à un objet de notre propre type. Au lieu de tester `ref($y)`, nous pourrions appeler `$y->isa("OctetBorne")`, si nous voulions être plus complet (et exécuter plus lentement).

Opérateurs surchargeables

Vous ne pouvez surcharger que certains opérateurs, qui sont listés dans le *tableau 13-1*. Les opérateurs sont également listés dans le hachage `%overload::ops` mis à disposition lorsque vous employez `use overload`, bien que la catégorisation y soit légèrement différente.

Tableau 13-1. Opérateurs surchargeables

Catégorie	Opérateurs
Conversion	`"" 0+ bool`
Arithmétique	`+ - * / % ** x . neg`
Logique	`!`
Sur les bits	`& | ~ ^ ! << >>`
Affectation	`+= -= *= /= %= **= x= .= <<= >>= ++ --`
Comparaison	`== < <= > >= != <=> lt le gt ge eq ne cmp`
Mathématique	`atan2 cos sin exp abs log sqrt`
Itératif	`<>`
Déréférence	`${} @{} %{} &{} *{}`
Pseudo	`nomethod fallback =>`

Notez que `neg`, `bool`, `nomethod` et `fallback` ne sont pas réellement des opérateurs Perl. Les cinq opérateurs de déréférence, `""` et `0+` ne *ressemblent* sans doute pas non plus à des opérateurs. Néanmoins tous sont des clefs valides de la liste de paramètres que vous

fournissez à `use overload`. Ceci n'est pas vraiment un problème. Nous allons vous révéler un petit secret : c'est un petit mensonge de dire que le pragma `overload` surcharge les opérateurs. Il surcharge les opérations sous-jacentes, qu'elles soient invoquées explicitement par leurs opérateurs « officiels » ou implicitement par un opérateur apparenté. (Les pseudo-opérateurs mentionnés ci-dessus ne peuvent être invoqués qu'implicitement.) En d'autres termes, la surcharge ne se produit pas au niveau syntaxique, mais au niveau sémantique. Le but n'est pas d'avoir l'air bien, mais de faire ce qui est bien. Vous pouvez généraliser.

Notez aussi que `=` ne surcharge *pas* l'opérateur d'affectation de Perl, comme vous pourriez le croire. Cela ne serait pas la bonne chose à faire. Nous verrons cela plus tard.

Commençons par traiter les opérateurs de conversion, non parce qu'ils sont les plus évidents (ils ne le sont pas), mais parce qu'ils sont les plus utiles. De nombreuses classes ne surchargent que la conversion en chaîne, spécifiée par la clef `""`. (Oui, ce sont vraiment deux doubles apostrophes à la suite.)

Opérateurs de conversion : `""`*,* `0+`*,* `bool`

Ces trois clefs vous permettent de fournir à Perl des comportements pour les conversions automatiques en chaîne, en nombre et en valeur booléenne, respectivement.

La conversion en chaîne ou *stringification* a lieu lorsqu'une variable non-chaîne est utilisée en tant que chaîne. C'est ce qui se passe lorsque vous convertissez une variable en chaîne par affichage, par interpolation, par concaténation ou même en l'utilisant comme clef de hachage. C'est la conversion en chaîne qui fait que vous voyez quelque chose comme `SCALAR(0xba5fe0)` lorsque vous essayez d'afficher un objet avec `print`.

La conversion en nombre ou *numification* a lieu lorsqu'une variable non-numérique est convertie en nombre dans un contexte numérique comme une expression mathématique, un indice de tableau, ou même un opérande de `..`, l'opérateur d'intervalle.

Enfin, bien que personne ici n'ose tout à fait appeler cela *boolification*, vous pouvez définir comment un objet doit être interprété en contexte booléen (comme `if`, `unless`, `while`, `for`, `and`, `or`, `&&`, `||`, `?:`, ou dans le bloc d'une expression `grep`) en créant un handler de `bool`.

Si vous avez défini l'un des trois opérateurs de conversion, les deux autres peuvent être *autogénérés* (nous expliquerons l'autogénération plus loin). Vos handlers peuvent retourner la valeur qu'ils veulent. Notez que si l'opération qui a déclenché la conversion est également surchargée, *cette* dernière surcharge aura lieu immédiatement après.

Voici une démonstration de `""` qui invoque le handler `comme_chaine` d'un objet au moment de la conversion en chaîne. N'oubliez pas de délimiter les doubles apostrophes :

```
package Personne;

use overload q("") => \&comme_chaine;

sub new {
    my $classe = shift;
```

```
    return bless { @_ } => $classe;
}

sub comme_chaine {
    my $self = shift;
    my ($clef, $valeur, $resultat);
    while (($clef, $valeur) = each %$self) {
        $resultat .= "$clef => $valeur\n";
    }
    return $resultat;
}

$obj = Personne->new(taille => 72, poids => 165, yeux => "marron");

print $obj;
```

Au lieu de quelque chose comme `Personne=HASH(0xba1350)`, ceci affiche (dans l'ordre du hachage) :

```
poids => 165
taille => 72
yeux => marron
```

(Nous espérons sincèrement que cette personne n'a pas été mesurée en kilos et en centimètres.)

Opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%`, `**`, `x`, `.`, `neg`

Ceux-ci devraient tous vous êtres familiers, sauf `neg`, qui est une clef spéciale de surcharge du moins unaire : le `-` de `-123`. La distinction entre les clefs `neg` et `-` vous permet de spécifier des comportements différents pour le moins unaire et pour le moins binaire, plus connu sous le nom de soustraction.

Si vous surchargez `-` mais pas `neg`, et qu'ensuite vous utilisez le moins unaire, Perl vous émulera un handler de `neg`. Ceci est appelé *autogénération*, et se produit lorsque certains opérateurs peuvent raisonnablement être déduits d'autres opérateurs (en supposant que les opérateurs surchargés auront le même rapport que les opérateurs ordinaires). Comme le moins unaire peut être exprimé en fonction du moins binaire (ainsi `-123` équivaut à `0 - 123`), Perl ne vous oblige pas à surcharger `neg` lorsque `-` suffit. (Bien sûr, si vous avez arbitrairement spécifié que le moins binaire divise le second paramètre par le premier, le moins unaire sera une excellente manière de lever une exception de division par zéro.)

La concaténation avec l'opérateur `.` peut être autogénérée à partir du handler de conversion de chaîne (voir `""` ci-dessus).

Opérateur logique : `!`

Si un handler pour `!` n'est pas spécifié, il peut être autogénéré à partir du handler de `bool`, `""`, ou `0+`. Si vous surchargez l'opérateur `!`, l'opérateur `not` déclenchera aussi le comportement que vous avez spécifié. (Vous souvenez-vous de notre petit secret ?)

Vous serez peut-être surpris par l'absence des autres opérateurs logiques, mais ils ne peuvent pas pour la plupart être surchargés car ils font un court-circuit. Ce sont en réalité des opérateurs de flot de contrôle qui doivent être capables de retarder l'éva-

luation de certains de leurs paramètres. C'est aussi la raison pour laquelle l'opérateur `?:` n'est pas surchargeable.

Opérateurs sur les bits : `&`, `|`, `~`, `^`, `<<`, `>>`

L'opérateur `~` est un opérateur unaire ; tous les autres sont binaires. Voici comment nous pourrions surcharger `>>` pour faire quelque chose comme `chop` :

```
package DecaleChaine;

use overload
    '>>' => \&decale_droite,
    '""' => sub { ${ $_[0] } };

sub new {
    my $classe = shift;
    my $valeur = shift;
    return bless \$valeur => $classe;
}

sub decale_droite {
    my ($x, $y) = @_;
    my $valeur = $$x;
    substr($valeur, -$y) = "";
    return bless \$valeur => ref($x);
}

$chameau = DecaleChaine->new("Chameau");
$belier = $chameau >> 2;
print $belier;            # Chame
```

Opérateurs d'affectation : `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `x=`, `.=`, `<<=`, `>>=`, `++`, `--`

Ces opérateurs d'affectation peuvent changer la valeur de leurs paramètres ou les laisser inchangés. Le résultat n'est affecté à l'opérateur de gauche que si la nouvelle valeur diffère de l'ancienne. Ceci permet d'utiliser le même handler à la fois pour `+=` et pour `+`. Bien que ce soit permis, ceci est rarement recommandé, puisque d'après la sémantique décrite plus loin dans la section *Lorsqu'un handler de surcharge fait défaut (nomethod et fallback)*, Perl invoquera de toute façon le handler de `+`, en supposant que `+=` n'ait pas été surchargé directement.

La concaténation (`.=`) peut être autogénérée avec la conversion de chaîne, suivie de la concaténation de chaînes ordinaire. Les opérateurs `++` et `--` peuvent être autogénérés à partir de `+` et de `-` (ou de `+=` et de `-=`).

Il est attendu des handlers implémentant `++` et `--` qu'ils *mutent* (altèrent) leur paramètre. Si vous vouliez que l'autodécrémentation fonctionne pour les lettres comme pour les nombres, vous pourriez le faire avec un handler comme ceci :

```
package DecMagique;

use overload
    q(--) => \&decrementer,
    q("") => sub { ${ $_[0] } };
```

```
sub new {
    my $classe = shift;
    my $valeur = shift;
    bless \$valeur => $classe;
}

sub decrementer {
    my @chaine = reverse split(//, ${ $_[0] } );
    my $i;
    for ($i = 0; $i < @chaine; $i++ ) {
        last unless $chaine[$i] =~ /a/i;
        $chaine[$i] = chr( ord($chaine[$i]) + 25 );
    }
    $chaine[$i] = chr( ord($chaine[$i]) - 1 );
    my $resultat = join('', reverse @chaine);
    $_[0] = bless \$resultat => ref($_[0]);
}

package main;

for $normal (qw/perl NZ Pa/) {
    $magique = DecMagique->new($normal);
    $magique--;
    print "$normal devient $magique\n";
}
```

Ce qui affiche :

```
perl devient perk
NZ devient NY
Pa devient Oz
```

inversant précisément l'opérateur magique d'auto-incrémentation de Perl.

L'opération `++$a` peut être autogénérée avec `$a += 1` ou avec `$a = $a + 1` , et `$a--` avec `$a -= 1` ou `$a = $a - 1`. Cependant, cela ne déclenche pas le comportement de copie d'un vrai opérateur `++`. Voir *Le constructeur de copie (=)* plus loin dans ce chapitre.

Opérateurs de comparaison : `==`, `<`, `<=`, `>`, `>=`, `!=`, `<=>`, `lt`, `le`, `gt`, `ge`, `eq`, `ne`, `cmp`

Si `<=>` est surchargé, il peut servir pour autogénérer les comportements des opérateurs suivants : `<`, `<=`, `>`, `>=`, `==`, et `!=`. De manière similaire, si `cmp` est surchargé, il peut servir pour autogénérer les comportements des opérateurs suivants : `lt`, `le`, `gt`, `ge`, `eq`, and `ne`.

Notez que la surcharge de `cmp` ne vous permettra pas de trier les objets aussi facilement que vous le souhaiteriez, car ce sont les versions converties en chaînes des objets qui seront comparées, et non les objets eux-mêmes. Si votre but était de comparer les objets eux-mêmes, vous auriez aussi à surcharger `""`.

Fonctions mathématiques : `atan2`, `cos`, `sin`, `exp`, `abs`, `log`, `sqrt`

Si `abs` n'est pas disponible, il peut être autogénéré à partir de `<` ou de `<=>` en combinaison avec le moins unaire ou avec la soustraction.

Un opérateur `-` surchargé peut servir à autogénérer les handlers manquants du

moins unaire ou de la fonction `abs`, qui peuvent aussi être surchargés séparément. (Oui, nous savons que `abs` ressemble à une fonction, alors que le moins unaire ressemble à un opérateur, mais ils ne sont pas si différents que cela en ce qui concerne Perl.)

Opérateur itératif : `<>`

Le handler de `<>` peut être déclenché en utilisant la fonction `readline` (lorsqu'elle lit dans un fichier, comme dans `while (<FH>)`) ou la fonction `glob` (lorsqu'elle sert à faire un glob de fichiers, comme dans `@files = <*.*>`).

```
package TirageHeureux;

use overload
    '<>' => sub {
        my $self = shift;
        return splice @$self, rand @$self, 1;
    };


sub new {
    my $classe = shift;
    return bless [@_] => $classe;
}

package main;

$loto = new TirageHeureux 1 .. 49;

for (qw(1er 2eme 3eme 4eme 5eme 6eme)) {
    $numero_chanceux = <$loto>;
    print "Le $_ numéro chanceux est : $numero_chanceux\n";
}

$numero_chanceux = <$loto>;
print "\nEt le numéro complémentaire est : $numero_chanceux\n";
```

En France, ceci affiche :

```
The 1er numéro chanceux est : 14
The 2eme numéro chanceux est : 29
The 3eme numéro chanceux est : 45
The 4eme numéro chanceux est : 23
The 5eme numéro chanceux est : 9
The 6eme numéro chanceux est : 25

Et le numéro complémentaire est : 8
```

et vous rapporte 98 450 870 F.

Opérateurs de déréférencement : `${}`, `@{}`, `%{}`, `&{}`, `*{}`

Les tentatives de déréférencement d'une référence de scalaire, de tableau, de hachage, de fonction, ou de glob peuvent être interceptées en surchargeant ces cinq symboles.

La documentation Perl en ligne de `overload` montre comment vous pouvez utiliser ces opérateurs pour simuler vos propres pseudo-hachages. Voici un exemple plus simple qui implémente un objet avec un tableau anonyme mais permet de le référencer comme un hachage. N'essayez pas de le traiter comme un vrai hachage ; vous ne pourrez pas utiliser `delete` pour supprimer des couples clef/valeur de l'objet. Si vous voulez combiner les notations de tableau et de hachage, utilisez un vrai pseudo-hachage (si l'on peut dire).

```
package PsychoHachage;

use overload '%{}' => \&comme_hachage;

sub comme_hachage {
    my ($x) = shift;
    return { @$x };
}

sub new {
    my $classe = shift;
    return bless [ @_ ] => $classe;
}

$bestiau = new PsychoHachage( taille => 72, poids => 365, type =>
"chameau" );

print $bestiau->{poids};   # affiche 365
```

Voyez aussi le chapitre 14, *Variables liées*, pour un mécanisme qui vous permet de redéfinir les opérations de base sur les hachages, les tableaux, et les scalaires.

Lorsque vous surchargez un opérateur, essayez de ne pas créer des objets avec des références sur eux-mêmes. Par exemple,

```
use overload '+' => sub { bless [ \$_[0], \$_[1] ] };
```

Ceci revient à chercher des ennuis, car si vous dites `$animal += $legume`, le résultat fera de `$animal` une référence à un tableau consacré dont le premier élément est `$animal`. Ceci est une *référence circulaire*, ce qui veut dire que si vous détruisez `$animal`, sa mémoire ne sera libérée que lorsque se termine votre processus (ou votre interpréteur). Voir *Ramasse-miettes, références circulaires et références faibles* au chapitre 8, *Références*.

Le constructeur de copie (=)

Bien qu'il ressemble à un opérateur ordinaire, `=` a une signification spéciale et légèrement non-intuitive en tant que clef de surcharge. Il ne surcharge *pas* l'opérateur d'affectation de Perl. Il ne le peut pas, car cet opérateur doit être réservé pour l'affectation des références, sinon plus rien ne fonctionne.

Le handler de `=` est utilisé dans des situations où un opérateur de mutation (comme `++`, `--` ou l'un des opérateurs d'affectation) est appliqué à une référence qui partage son objet avec une autre référence. Le handler de `=` vous permet d'intercepter l'opérateur de mutation et de copier l'objet vous-même de telle façon que la copie seule soit mutée (al-

térée). Sans cela, vous écraseriez l'objet d'origine.

```
$copie = $origine;   # ne copie que la référence
++$copie;            # modifie l'objet partagé sous-jacent
```

C'est ici qu'il faut faire bien attention. Supposons que `$origine` soit une référence à un objet. Pour faire en sorte que `++$copie` ne modifie que `$copie` et non `$origine`, on commence par faire une copie de `$copie` et l'on affecte à `$copie` une référence à ce nouvel objet. Cette opération n'a lieu que lorsque `++$copie` est exécuté, donc `$copie` est identique à `$origine` avant l'incrémentation — mais pas après. En d'autres termes, c'est le `++` qui reconnaît la nécessité de faire une copie et qui appelle votre constructeur de copie.

La nécessité de faire une copie n'est reconnue que par les opérateurs de mutations comme `++` ou `+=`, ou par `nomethod`, décrit plus loin. Si l'opération est autogénérée avec `+`, comme dans :

```
$copie = $origine;
$copie = $copie + 1;
```

alors il n'y a pas de copie, car `+` ne sait pas qu'il est utilisé comme opérateur de mutation.

Si l'exécution d'un opérateur de mutation nécessite le constructeur de copie, mais qu'un handler pour `=` n'est pas spécifié, il peut être autogénéré comme copie de chaîne tant que l'objet reste un simple scalaire et non quelque chose de plus compliqué.

Par exemple, le code réellement exécuté lors de la séquence :

```
$copie = $origine;
...
++$copie;
```

pourrait ressembler à quelque chose comme ceci :

```
$copie = $origine;
...
$copie = $copie->clone(undef, "");
$copie->incr(undef, "");
```

Dans cet exemple `$origine` fait référence à un objet surchargé, `++` est surchargé avec `\&incr`, et `=` est surchargé avec `\&clone`.

Un comportement similaire est déclenché par `$copie = $origine++`, ce qui est interprété comme `$copie = $origine; ++$origine`.

Lorsqu'un handler de surcharge fait défaut (nomethod et fallback)

Si vous appliquez un opérateur non surchargé à un objet, Perl tente d'abord d'autogénérer un comportement à partir d'autres opérateurs surchargés en utilisant les règles décrites ci-dessus. Si cela échoue, Perl cherche un comportement de surcharge de `nomethod` et l'utilise s'il est disponible. Ce handler joue le même rôle pour les opérateurs que joue la fonction `AUTOLOAD` pour les fonctions : c'est ce que vous faites lorsque vous ne savez pas quoi faire d'autre.

Si elle est employée, la clef `nomethod` doit être suivie d'une référence à un handler prenant quatre paramètres (et non trois comme pour tous les autres handlers). Les trois premiers paramètres ne diffèrent pas de ceux des autres handlers ; le quatrième est une chaîne correspondante à l'opérateur dont le handler fait défaut. Ceci joue le même rôle que la variable `$AUTOLOAD` pour les fonctions `AUTOLOAD`.

Si Perl doit chercher un handler de `nomethod` mais n'en trouve pas, une exception est levée.

Si vous voulez empêcher l'autogénération, ou si vous voulez qu'une tentative d'autogénération infructueuse ne produise aucune surcharge, vous pouvez définir la clef spéciale de surcharge `fallback`. Celle-ci comporte trois états :

`undef`

: Si `fallback` n'est pas définie, ou si `undef` lui est explicitement affecté, la séquence d'événements de surcharge n'est pas modifiée : Perl cherche les handlers, tente l'autogénération, et finit par invoquer le handler de `nomethod`. Si cela échoue une exception est levée.

faux

: Si l'on affecte à `fallback` une valeur définie mais fausse (comme `0`), Perl ne tente jamais d'autogénération. Il appellera le handler de `nomethod` s'il existe, mais lèvera une exception dans le cas contraire.

vrai

: Ceci est presque le même comportement que pour `undef`, mais aucune exception n'est levée si un handler approprié ne peut être synthétisé par autogénération. Au lieu de cela, Perl retourne au comportement non surchargé de cet opérateur, comme si la classe ne contenait aucun pragma `use overload`.

Surcharge de constantes

Vous pouvez modifier la façon dont Perl interprète les constantes avec `overload::constant`, qui est le plus utilement placé dans la méthode `import` d'un paquetage. (Si vous faites cela, vous devriez normalement invoquer `overload::remove_constant` dans la méthode `unimport` du paquetage pour que le paquetage puisse tout nettoyer lui-même lorsque vous le lui demandez.)

Les fonctions `overload::constant` et `overload::remove_constant` prennent chacune comme paramètre une liste de couples clef/valeur. Les clefs doivent être parmi les suivantes : `integer`, `float`, `binary`, `q`, et `qr`. Chaque valeur doit être le nom d'une fonction, une fonction anonyme ou une référence de code qui traitera les constantes.

```
sub import { overload::constant ( integer => \&entier_handler,
                                  float   => \&decimal_handler,
                                  binary  => \&base_handler,
                                  q       => \&chaine_handler,
                                  qr      => \&regex_handler ) }
```

Les handlers que vous spécifiez pour `integer` et `float` sont invoqués à chaque fois que l'analyseur lexical de Perl traite un nombre constant. Ceci est indépendant du pragma `use constant` ; de simples instructions comme :

```
$annee = cube(12) + 1;        # entier
$pi    = 3.14159265358979;    # nombre décimal
```

déclencheront le handler spécifié.

La clef `binary` vous permet d'intercepter des constantes binaires, octales et hexadécimales. `q` traite les chaînes entre apostrophes (dont les chaînes introduites avec `q`) et les sous-chaînes constantes contenues dans les chaînes citées avec `qq` et `qx` et les documents ici-même. Pour finir, `qr` traite les parties constantes contenues dans les expressions régulières, comme il est mentionné à la fin du chapitre 5, *Recherche de motif*.

Le handler prend trois paramètres. Le premier paramètre est la constante d'origine, sous la forme fournie à Perl. Le deuxième paramètre contient la constante une fois qu'elle est interprétée par Perl ; par exemple, `123_456` apparaîtra comme `123456`.

Le troisième paramètre n'est défini que pour les chaînes traitées par les handlers de `q` et de `qr` et aura pour valeur `qq`, `q`, `s` ou `tr` selon l'utilisation de la chaîne. `qq` signifie que la chaîne provient d'un contexte interpolé, comme les doubles apostrophes, les apostrophes inverses, une correspondance `m//` ou le motif d'une substitution `s///`. `q` signifie que la chaîne provient d'un contexte non interpolé, `s` signifie que la constante est la chaîne de remplacement d'une substitution `s///`, et `tr` signifie qu'elle est un composant d'une expression `tr///` ou `y///`.

Le handler doit retourner un scalaire, qui sera utilisé à la place de la constante. Souvent, ce scalaire sera une référence à un objet surchargé, mais rien ne vous empêche de faire quelque chose de plus pervers :

```
package Doubleur;    # Un module à placer dans Doubleur.pm
use overload;

sub import { overload::constant ( integer => \&handler,
                                  float   => \&handler ) }

sub handler {
    my ($orig, $interp, $contexte) = @_;
    return $interp * 2;          # doubler toutes les constantes
}

1;
```

Notez que `handler` est partagé par les deux clefs, ce qui fonctionne correctement dans ce cas. Maintenant quand vous dites :

```
use Doubleur;

$ennuis = 123;      # ennuis est maintenant 246
$danger = 3.21;     # danger est maintenant 6.42
```

vous redéfinissez le monde.

Si vous interceptez les chaînes constantes, il est conseillé de fournir également un opérateur de concaténation (« `.` »), puisqu'une expression interpolée comme `"ab$cd!!"` n'est qu'un raccourci pour l'expression plus longue `'ab' . $cd . '!!'`. De façon similaire, on considère les nombres négatifs comme les négations de constantes positives, donc vous devez fournir un handler pour `neg` lorsque vous interceptez des entiers ou

des nombres décimaux. (Il n'y avait pas besoin de faire cela ci-dessus car nous retournons des nombres, et non des références d'objets surchargés.)

Notez que `overload::constant` ne se propage pas à l'intérieur d'un `eval` compilé au moment de l'exécution, ce qui est soit un bogue soit une fonctionnalité selon votre point de vue.

Fonctions publiques de surcharge

Depuis la version 5.6 de Perl, le pragma `use overload` fournit les fonctions suivantes à usage public :

`overload::StrVal(`*`OBJ`*`)`

Cette fonction retourne la valeur de chaîne que retournerait *`OBJ`* en l'absence de la surcharge de la conversion en chaîne (`""`).

`overload::Overloaded(`*`OBJ`*`)`

Cette fonction retourne vrai si *`OBJ`* est sujet à la surcharge d'un opérateur quel qu'il soit, et faux s'il ne l'est pas.

`overload::Method(`*`OBJ, OPERATEUR`*`)`

Cette fonction retourne une référence au code qui implémente la surcharge de *`OPERATEUR`* lorsqu'il opère sur *`OBJ`*, ou `undef` si une telle surcharge n'existe pas.

Héritage et surcharge

L'héritage interagit avec la surcharge de deux façons. La première a lieu lorsqu'un handler est nommé en tant que chaîne au lieu d'être spécifié comme référence de code ou comme fonction anonyme. Lorsqu'il est nommé en tant que chaîne le handler est interprété comme une méthode, et peut donc être hérité de surclasse.

La deuxième interaction entre l'héritage et la surcharge provient du fait que toute classe dérivée d'une classe surchargée est elle-même sujette à cette surcharge. En d'autres termes, la surcharge est elle-même héritée. L'ensemble des handlers d'une classe est l'union des handlers de tous les ancêtres de cette classe, récursivement. Si un handler peut être trouvé dans plusieurs ancêtres différents, le choix du handler à utiliser est régi par les règles habituelles d'héritage de méthode. Par exemple, si la classe `Alpha` hérite des classes `Betaet Gamma` dans cet ordre, et que la classe `Beta` surcharge `+` avec `\&Beta::fonction_plus`, mais que la classe `Gamma` surcharge `+` avec la chaîne `"methode_plus"`, alors c'est `Beta::fonction_plus` qui sera appelée lorsque vous appliquez `+` à un objet `Alpha`.

Comme la valeur de la clef `fallback` n'est pas un handler, son héritage n'est pas régi par les règles ci-dessus. Dans l'implémentation actuelle, on utilise la valeur de `fallback` du premier ancêtre surchargé, mais ceci est accidentel et sujet à modification sans préavis (enfin, sans trop de préavis).

Surcharge à l'exécution

Comme les instructions `use` sont exécutées à la compilation, la seule façon de modifier la surcharge à l'exécution est :

```
eval " use overload '+' => \&mon_addition ";
```

Vous pouvez aussi dire :

```
eval " no overload '+', '--', '<=' ";
```

bien que l'emploi de ces formules à l'exécution soit douteux.

Surcharge de diagnostics

Si votre Perl est compilé avec `-DDEBUGGING`, vous pouvez voir les messages de diagnostics de surcharge lorsque vous exécutez un programme avec l'option **`-Do`** ou son équivalent. Vous pouvez aussi déduire quelles opérations sont surchargées en utilisant la commande `m` du débogueur interne de Perl.

Si maintenant vous vous sentez un peu surchargé, le chapitre suivant vous permettra peut-être de lier un peu la sauce.

14

Variables liées

Certaines activités humaines nécessitent un déguisement. Parfois l'intention est de tromper, mais le plus souvent il s'agit de communiquer quelque chose de vrai à un niveau plus profond. Souvent, par exemple, quelqu'un qui vous fait passer un entretien d'embauche s'attend à ce que vous portiez une cravate pour signifier que vous souhaitez sérieusement vous intégrer, même si vous savez tous les deux que vous ne porterez jamais de cravate au travail. C'est curieux lorsqu'on y pense : nouer un morceau de tissu autour de son cou peut magiquement vous obtenir un emploi. Dans la culture Perl, l'opérateur de liaison *tie*[1] joue un rôle similaire : il vous permet de créer une variable apparemment normale qui, derrière son déguisement, est en fait un objet Perl à part entière dont on peut s'attendre qu'elle ait sa propre personnalité intéressante. C'est juste un drôle de bout de magie, comme tirer Bugs Bunny d'un chapeau.

Formulé autrement, les drôles de caractères `$`, `@`, `%` ou `*` devant un nom de variable en disent long à Perl et à ses programmeurs — ils sous-entendent chacun un ensemble spécifique de comportements caractéristiques. Vous pouvez déformer ces comportements de diverses manières utiles avec `tie` en associant la variable à une classe qui implémente un nouvel ensemble de comportements. Par exemple, vous pouvez créer un hachage Perl ordinaire et le lier avec `tie` à une classe qui transforme le hachage en base de données, de sorte que quand vous lisez des valeurs du hachage, Perl va par magie chercher les données dans un fichier de données externe, et que quand vous affectez des valeurs au hachage, Perl stocke par magie les données dans le même fichier. Ici, « par magie » signifie « en effectuant de manière transparente quelque chose de très compliqué ». Vous connaissez le vieux dicton : toute technologie suffisamment avancée est indiscernable d'un script Perl. (Sérieusement, les gens qui jouent avec les tripes de Perl utilisent *magique* comme terme technique désignant toute sémantique supplémentaire attachée à des variables telles que `%ENV` ou `%SIG`. Les variables liées n'en sont qu'une extension.)

Perl possède déjà les fonctions internes `dbmopen` et `dbmclose`, qui lient magiquement des variables de hachage à des bases de données, mais ces fonctions datent du temps où Perl

1. Note du traducteur : en anglais « tie » signifie lier ou attacher, et... cravate.

n'avait pas l'opérateur `tie`. Maintenant, `tie` fournit un mécanisme plus général. En fait, Perl lui-même implémente `dbmopen` et `dbmclose` à l'aide de `tie`.

Vous pouvez lier avec `tie` un scalaire, un tableau, un hachage ou un handle de fichier (via son typeglob) à toute classe qui fournit les méthodes nommées de manière appropriée pour intercepter et simuler un accès normal à ces variables. La première de ces méthodes est invoquée au moment de la liaison elle-même : le fait de lier une variable invoque toujours un constructeur, qui retourne en cas de succès un objet que Perl stocke à l'abri des regards, caché à l'intérieur de la variable « normale ». Vous pouvez toujours récupérer cet objet plus tard en appliquant la fonction `tied` à la variable normale :

```
tie VARIABLE, NOM_CLASSE, LISTE;  # relie VARIABLE à NOM_CLASSE
$objet = tied VARIABLE;
```

Ces deux lignes sont équivalentes à :

```
$objet = tie VARIABLE, NOM_CLASSE, LISTE;
```

Une fois qu'elle est liée, vous pouvez traiter la variable normale normalement, mais chaque accès invoque automatiquement une méthode de l'objet sous-jacent ; toute la complexité de la classe est cachée derrière ces appels de méthodes. Si plus tard vous voulez rompre l'association entre la variable et la classe, vous pouvez délier la variable avec `untie` :

```
untie VARIABLE;
```

Vous pouvez presque considérer `tie` comme une espèce de `bless` bizarre, à ceci près qu'il consacre une simple variable au lieu d'une référence à un objet. Il peut aussi prendre des paramètres supplémentaires, à l'instar d'un constructeur — ce qui n'est pas terriblement surprenant, dans la mesure où il invoque un constructeur en interne, dont le nom dépend du type de la variable que vous liez : `TIESCALAR`, `TIEARRAY`, `TIEHASH` ou `TIEHANDLE`.[2] Ces constructeurs sont invoqués en tant que méthodes de classe, avec *`NOM_CLASSE`* comme invoquant, plus tout paramètre additionnel que vous donnez dans *`LISTE`*. (La *`VARIABLE`* n'est pas passée au constructeur.)

Ces quatre constructeurs retournent chacun un objet de la façon habituelle. Ni les constructeurs, ni les autres méthodes de la classe ne se soucient vraiment de savoir s'ils ont été invoqués par `tie`, puisque vous pouvez toujours les invoquer directement si vous le souhaitez. En un sens, toute la magie se situe dans `tie` et non dans la classe implémentant le `tie`. La classe n'est en ce qui la concerne qu'une classe ordinaire avec des drôles de noms de méthode. (En effet, certains modules liés fournissent des méthodes supplémentaires qui ne sont pas visibles par la variable liée ; ces méthodes doivent être appelées explicitement comme vous le feriez pour toute autre méthode d'objet. De telles méthodes supplémentaires pourraient fournir des services comme le verrouillage de fichiers, la protection de transactions, ou tout autre chose que pourrait faire une méthode d'instance.)

Ainsi ces constructeurs consacrent (avec `bless`) et retournent une référence d'objet comme le ferait toute autre constructeur. Cette référence ne doit pas forcément référen-

2. Comme les constructeurs ont des noms distincts, vous pourriez même fournir une classe unique les implémentant tous. Cela vous permettrait de lier des scalaires, des tableaux, des hachages et des handles de fichiers à la même classe, bien que ce ne soit généralement pas fait, car cela rendrait délicat l'écriture des autres méthodes magiques.

cer le même type de variable que celle qui est liée ; elle doit simplement être consacrée afin que la variable liée puisse retrouver le chemin de votre classe en cas de besoin. Par exemple, notre long exemple de `TIEARRAY` utilise un objet basé sur un hachage afin qu'il puisse commodément contenir des informations supplémentaires sur le tableau qu'il émule.

La fonction `tie` n'exécutera pas à votre place de `use` ou de `require` — vous devez le faire vous-même explicitement, avant d'appeler le `tie`. (Cela dit, pour des raisons de compatibilité descendante, la fonction `dbmopen` tentera de faire un `use` de l'une ou l'autre implémentation DBM. Mais vous pouvez l'empêcher de choisir en faisant un `use` explicite, du moment que le module que vous spécifiez figure dans la liste des modules que peut choisir `dbmopen`. Voir la doc en ligne du module `AnyDBM_File` pour une explication plus complète.)

Les méthodes appelées par une variable liée ont un nom prédéterminé comme `FETCH` (récupérer) et `STORE` (stocker), puisqu'elles sont invoquées implicitement (c'est-à-dire déclenchées par des événements spécifiques) de l'intérieur de la machinerie de Perl. Ces noms sont en `MAJUSCULES`, suivant la convention courante pour les fonctions appelées implicitement. (D'autres noms spéciaux qui suivent cette convention sont `BEGIN`, `CHECK`, `INIT`, `END`, `DESTROY` et `AUTOLOAD`, sans parler de `UNIVERSAL->VERSION`. En fait, presque toutes les variables prédéfinies et les handles de fichiers de Perl sont en majuscules : `STDIN`, `SUPER`, `CORE`, `CORE::GLOBAL`, `DATA`, `@EXPORT`, `@INC`, `@ISA`, `@ARGV` et `%ENV`. Bien entendu, les opérateurs internes et les pragmas, à l'extrême opposé, n'ont pas du tout de majuscules.)

La première chose que nous traiterons est extrêmement simple : comment lier une variable scalaire.

Liaison de scalaire

Afin d'implémenter un scalaire lié, une classe doit définir les méthodes suivantes : `TIESCALAR`, `FETCH` et `STORE` (et éventuellement `DESTROY`). Lorsque vous liez une variable scalaire avec `tie`, Perl appelle `TIESCALAR`. Lorsque vous consultez la variable liée, il appelle `FETCH`, et lorsque vous affectez une valeur à la variable, il appelle `STORE`. Si vous avez gardé en main l'objet retourné par le `tie` initial (ou si vous le récupérez plus tard avec `tied`), vous pouvez accéder vous-même à l'objet sous-jacent — ceci ne déclenche ni `FETCH` ni `STORE`. En tant qu'objet, il n'est pas du tout magique, et tout à fait objectif.

S'il existe une méthode `DESTROY`, Perl l'invoque lorsque disparaît la dernière référence à l'objet lié, comme pour tout autre objet. Cela se produit lorsque votre programme se termine ou lorsque vous appelez `untie`, ce qui élimine la référence utilisée par tie. Cependant, `untie` n'élimine pas les références existantes que vous pourriez avoir stockées ailleurs ; `DESTROY` est reporté jusqu'à ce que ces références aient également disparu.

Les paquetages `Tie::Scalar` et `Tie::StdScalar`, situés tous les deux dans le module `Tie::Scalar`, fournissent des définitions de classe de base simples si vous ne souhaitez pas définir toutes ces méthodes vous-même. `Tie::Scalar` fournit des méthodes élémentaires qui ne font quasiment rien, et `Tie::StdScalar` fournit des méthodes qui font se comporter un scalaire lié comme un scalaire Perl ordinaire. (Ce qui semble particulièrement inutile, mais parfois vous voulez juste un léger emballage autour de la sémantique de scalaire ordinaire, comme par exemple pour compter le nombre de fois qu'une variable donnée est affectée.)

Avant de vous montrer un exemple recherché et une description complète de toute la mécanique, voici juste un avant-goût pour vous mettre en appétit — et pour vous montrer à quel point c'est facile. Voici un programme complet :

```
#!/usr/bin/perl
package Centime;
sub TIESCALAR { bless \my $self, shift }
sub STORE { ${ $_[0] } = $_[1] }  # faire l'action par défaut
sub FETCH { sprintf "%.02f", ${ my $self = shift } } # valeur arrondie

package main;
tie $francs, "Centime";
$francs = 45.00;
$francs *= 1.0715; # taxe
$francs *= 1.0715; # et double taxe !
print "Ce sera $francs francs, s'il vous plaît.\n";
```

À l'exécution, le programme affiche :

```
Ce sera 51.67 francs, s'il vous plaît.
```

Pour voir la différence, mettez l'appel à `tie` en commentaire. Vous obtiendrez :

```
Ce sera 51.66505125 francs, s'il vous plaît.
```

Reconnaissons que cela représente beaucoup d'efforts juste pour arrondir des nombres.

Méthodes de liaison de scalaire

Maintenant que vous avez vu un échantillon de ce qui va venir, développons une classe de liaison de scalaire plus recherchée. Au lieu d'utiliser un paquetage prêt à l'emploi pour la classe de base (et étant donné que les scalaires sont si simples), nous examinerons tour à tour chacune des quatre méthodes en construisant une classe exemple nommée `FichierScalaire`. Les scalaires liés à cette classe contiennent des chaînes ordinaires, et chaque variable de ce type est implicitement associée au fichier dans lequel la chaîne est stockée. (Vous pourriez nommer vos variables de manière à vous rappeler à quel fichier vous faites référence.) Les variables sont liées à la classe de la manière suivante :

```
use FichierScalaire;       # charger FichierScalaire.pm
tie $chameau, "FichierScalaire", "/tmp/camel.ide";
```

Une fois la variable liée, son contenu d'origine est écrasé, et la connexion interne entre la variable et son objet supplante la sémantique normale de la variable. Lorsque vous demandez la valeur de `$chameau`, elle lit le contenu de */tmp/camel.ide*, et lorsque vous affectez une valeur à `$chameau`, elle écrit le nouveau contenu dans */tmp/camel.ide*, écrasant tout occupant précédent.

La liaison est faite sur la variable et non sur sa valeur, donc la nature liée d'une variable n'est pas préservée par l'affectation. Par exemple, supposons que vous copiiez une variable liée :

```
$dromadaire = $chameau;
```

Au lieu de lire la valeur comme à l'ordinaire dans la variable scalaire `$chameau`, Perl invoque la méthode `FETCH` de l'objet associé sous-jacent. C'est comme si vous aviez écrit

ceci :

```
$dromadaire = (tied $chameau)->FETCH():
```

Ou si vous retenez l'objet retourné par `tie`, vous pouvez utiliser cette référence directement, comme dans le code exemple suivant :

```
$cide = tie $chameau, "FichierScalaire", "/tmp/camel.ide";
$dromadaire = $chameau;        # par l'interface implicite
$dromadaire = $cide->FETCH();  # pareil, mais explicitement
```

Si la classe fournit des méthodes autres que `TIESCALAR`, `FETCH`, `STORE`, et `DESTROY`, vous pouvez utiliser `$cide` pour les invoquer manuellement. Toutefois, ordinairement on s'occupe de ses propres affaires et on laisse l'objet sous-jacent tranquille, ce qui explique qu'on utilise rarement la valeur de retour de `tie`. Vous pouvez toujours récupérer l'objet avec `tied` si vous en avez besoin plus tard (par exemple, si la classe documente d'autres méthodes dont vous avez besoin). Ne pas utiliser l'objet retourné élimine aussi certains types d'erreur, ce que nous traiterons plus loin.

Voici le préambule de notre classe, que nous mettrons dans *FichierScalaire.pm* :

```
package FichierScalaire;
use Carp;                # Transmettre gentiment les messages d'erreur.
use strict;              # S'imposer un peu de discipline.
use warnings;            # Activer les avertissements de portée lexicale.
use warnings::register;  # Permettre à l'utilisateur de dire
                         # "use warnings 'FichierScalaire'".
my $compteur = 0;        # Compteur interne de FichierScalaires liés.
```

Le module standard `Carp` exporte les fonctions `carp`, `croak`, et `confess`, que nous utiliserons dans le code plus loin. Comme d'habitude, voir le chapitre 32, *Modules standards*, ou la doc en ligne pour plus de détails sur `Carp`.

Les méthodes suivantes sont définies par la classe :

NOM_CLASSE`->TIESCALAR(`*LISTE*`)`

La méthode `TIESCALAR` de la classe est déclenchée lorsque vous liez une variable avec `tie`. La *LISTE* en option contient les paramètres nécessaires pour initialiser l'objet correctement. (Dans notre exemple, il n'y a qu'un seul paramètre : le nom du fichier.) La méthode doit retourner un objet, mais celui-ci n'est pas nécessairement une référence à un scalaire. Mais il l'est dans notre exemple.

```
sub TIESCALAR {              # dans FichierScalaire.pm
    my $classe      = shift;
    my $nom_fichier = shift;
    $compteur++;             # Une lexicale visible dans le fichier,
                             # privée pour la classe
    return bless \$nom_fichier, $classe;
}
```

Comme il n'y a pas d'équivalent scalaire des compositeurs de tableau et de hachage anonymes [ ] et {}, nous consacrons simplement le référent de la variable lexicale, qui devient de fait anonyme dès la sortie de la portée lexicale du nom. Ceci fonctionne bien (vous pourriez faire de même avec les tableaux et les hachages) tant que la variable est réellement lexicale. Si vous tentiez cette astuce avec une variable globale, vous pourriez croire vous en sortir jusqu'à ce que vous tentiez de créer un

autre *camel.ide*. Ne soyez pas tenté d'écrire quelque chose comme ceci :

```
sub TIESCALAR { bless \$_[1], $_[0] }  # FAUX, pourrait référencer
                                       # une variable globale
```

Un constructeur écrit de manière plus robuste pourrait vérifier que le fichier est accessible. Nous vérifions d'abord que le fichier est lisible, comme nous ne souhaitons pas écraser la variable existante. (En d'autres termes, il ne faut pas supposer que l'utilisateur commence par écrire dans le fichier. Il pourrait garder précieusement son ancien fichier *camel.ide* produit par une exécution antérieure du programme.) Si nous ne pouvons pas ouvrir ou créer le fichier spécifié, nous indiquons l'erreur gentiment en retournant `undef` et en affichant en option un avertissement avec `carp`. (Au lieu de cela nous pourrions moins gentiment faire un `croak` — c'est une affaire de goût.) Nous utilisons le pragma `warnings` pour déterminer si l'utilisateur s'intéresse ou non à notre avertissement :

```
sub TIESCALAR {           # dans FichierScalaire.pm
    my $classe      = shift;
    my $nom_fichier = shift;
    my $hf;
    if (open $hf, "<", $nom_fichier or
        open $hf, ">", $nom_fichier)
    {
        close $hf;
        $compteur++;
        return bless \$nom_fichier, $classe;
    }
    carp "Impossible de lier $nom_fichier: $!" if warnings::enabled();
    return;
}
```

Étant donné un tel constructeur, nous pouvons à présent associer le scalaire `$chaine` au fichier *camel.ide* :

```
tie ($chaine, "FichierScalaire", "camel.ide") or die;
```

(Nous faisons encore des suppositions que nous ne devrions pas faire. Dans une version en production, nous n'ouvririons probablement qu'une fois le fichier, en retenant le handle de fichier ainsi que le fichier pour la durée de la liaison, et en gardant tout ce temps un verrouillage exclusif sur le handle à l'aide de `flock`. Sans cela nous risquons des situations de concurrence (race conditions) — voir *Gestion des problèmes de synchronisation* dans le chapitre 23, *Sécurité*.)

SELF->FETCH

Cette méthode est invoquée à chaque fois que vous accédez à une variable liée (c'est-à-dire, que vous la lisez). Elle ne prend pas de paramètre hormis l'objet lié à la variable. Dans notre exemple, cet objet contient le nom de fichier :

```
sub FETCH {
    my $self  = shift;
    confess "Je ne suis pas une méthode de classe" unless ref $self;
    return unless open my $hf, $$self;
    read($hf, my $valeur, -s $hf);     # NB: n'utilisez pas -s avec
                                       # les tubes !
    return $valeur;
}
```

Cette fois nous avons décidé d'exploser (lever une exception) si FETCH reçoit autre chose qu'une référence. (Soit elle était invoquée comme méthode de classe, soit quelqu'un l'a appelée par erreur en tant que fonction.) Nous n'avons pas d'autre moyen de retourner une erreur, donc c'est probablement la réaction appropriée. En fait, Perl aurait de toute façon levé une exception dès que nous aurions tenté de déréférencer `$self` ; nous sommes simplement polis en utilisant `confess` pour déverser une trace arrière complète de la pile sur l'écran de l'utilisateur. (Si l'on peut considérer cela poli.)

Nous pouvons maintenant voir le contenu de *camel.ide* lorsque nous disons :

```
tie($chaine, "FichierScalaire", "camel.ide");
print $chaine;
```

SELF->STORE(*VALEUR*)

Cette méthode est exécutée lorsqu'une valeur est affectée à la variable liée. Le premier paramètre, *SELF*, est comme toujours l'objet associé à la variable ; *VALEUR* est ce qui est affecté à la variable. (Nous utilisons le terme « affecté » au sens large — toute opération qui modifie la variable peut appeler STORE.)

```
sub STORE {
    my($self,$valeur) = @_;
    ref $self             or confess "pas une méthode de classe";
    open my $hf, ">", $$self or croak "impossible d'écraser $$self: $!";
    syswrite($hf, $valeur) == length $valeur
                          or croak "impossible d'écrire dans $$self: $!";
    close $hf             or croak "impossible de fermer $$self: $!";
    return $valeur;
}
```

Après avoir « affecté » la nouvelle valeur, nous la retournons — car c'est ce que fait l'affectation. Si l'affectation n'a pas réussi, nous affichons l'erreur avec `croak`. Parmi les causes éventuelles d'échec : nous n'avions pas la permission d'écrire dans le fichier associé, ou le disque s'est rempli, ou des malins génies ont infesté le contrôleur de disque. Parfois vous contrôlez la magie, et parfois c'est elle qui vous contrôle.

Nous pouvons maintenant écrire dans *camel.ide* en disant ceci :

```
tie($chaine, "FichierScalaire", "camel.ide");
$chaine  = "Voici la première ligne de camel.ide\n";
$chaine .= "Et voici une deuxième ligne, rajoutée automatiquement.\n";
```

SELF->DESTROY

Cette méthode est déclenchée lorsque l'objet associé à la variable liée est sur le point d'être recyclé par le ramasse-miettes, au cas où il devrait faire quelque chose de spécial pour nettoyer autour de lui. Comme pour les autres classes, cette méthode est rarement nécessaire, puisque Perl vous libère automatiquement la mémoire de l'objet moribond. Ici, nous définissons une méthode DESTROY qui décrémente notre compteur de fichiers liés :

```
sub DESTROY {
    my $self = shift;
    confess "mauvais type" unless ref $self;
    $compteur--;
}
```

Nous pourrions alors aussi fournir une méthode de classe supplémentaire pour récupérer la valeur courante du compteur. À vrai dire, elle ignore si elle est appelée comme méthode de classe ou d'objet, mais vous n'avez plus d'objet après le DESTROY, non ?

```
sub count {
    # my $invoquant = shift;
    $compteur;
}
```

Vous pouvez l'appeler comme méthode de classe à tout moment comme ceci :

```
if (FichierScalaire->count) {
    warn "Il reste des FichierScalaires liés quelque part...\n";
}
```

C'est à peu près tout ce qu'il y a à faire. À vrai dire, c'est plus que tout ce qu'il y a à faire, puisque nous avons fait quelques choses gentilles pour être complets, robustes, et esthétiques (ou pas). Il est certainement possible de faire des classes TIESCALAR plus simples.

Variables compteur magiques

Voici une classe Tie::Counter simple, inspirée du module CPAN de même nom. Les variables liées à cette classe s'incrémentent de 1 à chaque fois qu'elles sont utilisées. Par exemple :

```
tie my $compte    "Tie::Counter", 100;
@tableau = qw /Rouge Vert Bleu/;
for my $couleur (@tableau) {          # Affiche :
    print "  $compteur  $couleur\n";   #   100  Rouge
}                                     #   101  Vert
                                      #   102  Bleu
```

Le constructeur prend en option un paramètre supplémentaire spécifiant la valeur initiale du compteur, par défaut 0. L'affectation au compteur lui donne une nouvelle valeur. Voici la classe :

```
package Tie::Counter;
sub FETCH     { ++ ${ $_[0] } }
sub STORE     { ${ $_[0] } = $_[1] }
sub TIESCALAR {
    my ($classe, $valeur) = @_;
    $valeur = 0 unless defined $valeur;
    bless \$valeur => $classe;
}
1;  # nécessaire dans un module
```

Vous voyez comme c'est court ? Nul besoin de beaucoup de code pour assembler une classe comme celle-ci.

Bannir $_ par magie

Cette classe de liaison curieusement exotique permet d'interdire les emplois non localisés de $_. Au lieu d'intégrer un module avec use, qui invoque la méthode import de la

classe, ce module doit être chargé avec `no` pour appeler la méthode `unimport` rarement utilisée. L'utilisateur dit :

```
no Souligne;
```

Alors toutes les utilisations de `$_` comme variable globale non localisée lèvent une exception.

Voici une suite de tests pour le module :

```
#!/usr/bin/perl
no Souligne;
@tests = (
    "Affectation"  => sub { $_ = "Mal" },
    "Lecture"      => sub { print },
    "Comparaison"  => sub { $x = /mauvais/ },
    "Chop"         => sub { chop },
    "Test fichier" => sub { -x },
    "Emboîté"      => sub { for (1..3) { print } },
);


while ( ($nom, $code) = splice(@tests, 0, 2) ) {
    print "Tester $nom : ";
    eval { &$code };
    print $@ ? "détecté" : " raté !";
    print "\n";
}
```

qui affiche ce qui suit :

```
Tester Affectation : détecté
Tester Lecture : détecté
Tester Comparaison : détecté
Tester Chop : détecté
Tester Test fichier : détecté
Tester Emboîté : 123 raté !
```

Ce dernier est « raté » parce qu'il est localisé correctement par la boucle `for` et donc sûr d'accès.

Voici le module curieusement exotique `Souligne` lui-même. (Avons-nous mentionné qu'il est curieusement exotique ?) Il fonctionne car la magie liée est de fait cachée par un `local`. Le module fait le `tie` dans son propre code d'initialisation pour le cas où il serait appelé par un `require`.

```
package Souligne;
use Carp;
sub TIESCALAR { bless \my $toto => shift }
sub FETCH { croak 'Accès en lecture à $_ interdit'  }
sub STORE { croak 'Accès en écriture à $_ interdit' }
sub unimport { tie($_, __PACKAGE__) }
sub import   { untie $_ }
tie($_, __PACKAGE__) unless tied $_;
1;
```

Il est difficile de mêler de manière utile dans votre programme des appels à use et à no pour cette classe, car ils se produisent tous à la compilation, et non à l'exécution. Vous pourriez appeler Souligne->import et Souligne->unimport directement, comme le font use et no. Mais normalement, pour vous permettre d'utiliser $_ librement, vous lui appliqueriez simplement local, ce qui est tout l'intérêt de la manipulation.

Liaison de tableau

Une classe qui implémente un tableau lié doit définir au moins les méthodes TIEARRAY, FETCH, et STORE. Il y a aussi beaucoup de méthodes optionnelles : la méthode DESTROY omniprésente, bien entendu, mais aussi les méthodes STORESIZE et FETCHSIZE, qui donnent accès à $#tableau et à scalar(@tableau). De plus, CLEAR est déclenchée lorsque Perl doit vider un tableau, et EXTEND lorsque Perl aurait par avance alloué plus de mémoire pour un vrai tableau.

Vous pouvez aussi définir les méthodes POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, et EXISTS si vous voulez que les fonctions Perl correspondantes fonctionnent avec le tableau lié. La classe Tie::Array peut servir de classe de base pour implémenter les cinq premières de ces fonctions à partir de FETCH et de STORE. (L'implémentation par défaut dans Tie::Array de DELETE et de EXISTS appelle simplement croak.) Tant que vous définissez FETCH et STORE, le type de structure de données contenu par votre objet n'a pas d'importance.

D'un autre côté, la classe de base Tie::StdArray (définie dans le module standard Tie::Array module) fournit des méthodes par défaut qui supposent que l'objet contient un tableau ordinaire. Voici une simple classe de liaison de tableau qui en fait usage. Parce qu'elle utilise Tie::StdArray comme classe de base, seules sont à définir les méthodes qui doivent être traitées d'une manière non standard.

```
#!/usr/bin/perl
package TableauHorloge;
use Tie::Array;
our @ISA = 'Tie::StdArray';
sub FETCH {
    my($self,$emplacement) = @_;
    $self->[ $emplacement % 12 ];
}
sub STORE {
    my($self,$emplacement,$valeur) = @_;
    $self->[ $emplacement % 12 ] = $valeur;
}

package main;
tie my @tableau, 'TableauHorloge';
@tableau = ( "a" ... "z" );
print "@tableau\n";
```

Lorsqu'il est exécuté, le programme affiche "y z o p q r s t u v w x". Cette classe implémente un tableau ne comportant que douze cases, comme les heures du cadran d'une horloge, numérotées de 0 à 11. Si vous demandez le 15e élément du tableau, vous obtenez en réalité le 3e. Vous pouvez voir la classe comme une aide pour les gens qui n'ont pas appris à lire l'heure sur une horloge à 24 heures.

Méthodes de liaison de tableau

C'était la méthode simple. Voyons maintenant les menus détails. Pour illustrer notre propos, nous implémenterons un tableau de taille limitée à la création. Si vous tentez d'accéder à quelque chose au-delà de cette limite, une exception est levée. Par exemple :

```
use TableauLimite;
tie @tableau, "TableauLimite", 2;

$tableau[0] = "bon";
$tableau[1] = "bien";
$tableau[2] = "super";
$tableau[3] = "oula";      # Interdit : affiche un message d'erreur
```

Le code en préambule de cette classe est le suivant :

```
package TableauLimite;
use Carp;
use strict;
```

Pour éviter de devoir définir `SPLICE` plus tard, nous l'hériterons de la classe `Tie::Array` :

```
use Tie::Array;
our @ISA = ("Tie::Array");
```

NOM_CLASSE`->TIEARRAY(`*LISTE*`)`

En tant que constructeur de la classe, `TIEARRAY` doit retourner une référence consacrée qui émulera le tableau lié.

Dans cet exemple, juste pour vous montrer que vous ne devez pas *nécessairement* retourner une référence de tableau, nous choisissons de représenter notre objet avec une référence de hachage. Un hachage fonctionne bien comme type générique d'enregistrement : la valeur de la clef « `LIMITE` » du hachage stocke la limite maximale autorisée, et sa valeur « `DONNEES` » contient les données réelles. Si quelqu'un à l'extérieur de la classe tente de déréférencer l'objet retourné (en pensant sans doute qu'il s'agit d'une référence à un tableau), une exception est levée.

```
sub TIEARRAY {
    my $classe = shift;
    my $limite = shift;
    confess "usage: tie(\@tableau, 'TableauLimite', index_max)"
        if @_ || $limite =~ /\D/;
    return bless { LIMITE => $limite, DONNEES => [] }, $classe;
}
```

Maintenant nous pouvons dire :

```
tie(@tableau, "TableauLimite", 3);  # l'index maximum autorisé est 3
```

pour nous assurer que le tableau n'aura jamais plus de quatre éléments. À chaque fois qu'un élément du tableau est consulté ou enregistré, `FETCH` ou `STORE` sera appelée tout comme elle l'était pour les scalaires, mais avec l'index en paramètre supplémentaire.

SELF`->FETCH(`*INDEX*`)`

Cette méthode est exécutée à chaque fois que l'on accède à un élément du tableau lié. Elle prend un paramètre après l'objet : l'index de la valeur que nous tentons de récupérer.

```
sub FETCH {
    my ($self, $index) = @_;
    if ($index > $self->{LIMITE}) {
        confess "Dépassement de limite du tableau :
                 $index > $self->{LIMITE}";
    }
    return $self->{DONNEES}[$index];
}
```

SELF->STORE(*INDEX*,*VALEUR*)

Cette méthode est invoquée à chaque fois que l'on affecte une valeur à un élément du tableau. Elle prend deux paramètres après l'objet : l'index de la case où nous tentons de stocker quelque chose et la valeur que nous tentons d'y mettre.

```
sub STORE {
    my($self, $index, $valeur) = @_;
    if ($index > $self->{LIMITE} ) {
        confess "Dépassement de limite du tableau :
                 $index > $self->{LIMITE}";
    }
    return $self->{DONNEES}[$index] = $valeur;
}
```

SELF->DESTROY

Perl appelle cette méthode lorsque la variable liée doit être détruite et sa mémoire récupérée. Ceci n'est presque jamais utile dans un langage avec un ramasse-miettes, donc cette fois-ci nous la laissons de côté.

SELF->FETCHSIZE

La méthode `FETCHSIZE` devrait retourner le nombre total d'éléments dans le tableau lié associé à *SELF*. C'est l'équivalent de `scalar(@tableau)`, qui est habituellement égal à `$#tableau + 1`.

```
sub FETCHSIZE {
    my $self = shift;
    return scalar @{$self->{DONNEES}};
}
```

SELF->STORESIZE(*NOMBRE*)

Cette méthode spécifie le *NOMBRE* total d'éléments dans le tableau lié associé à *SELF*. Si le tableau rétrécit, vous devez supprimer les éléments au-delà de *NOMBRE*. Si le tableau grandit, vous devez vous assurer que les nouvelles positions sont indéfinies. Dans notre classe `TableauLimite`, nous empêchons aussi que le tableau ne grandisse au-delà de la limite spécifiée initialement.

```
sub STORESIZE {
    my ($self, $nombre) = @_;
    if ($nombre > $self->{LIMITE}) {
        confess "Dépassement de limite du tableau :
                 $nombre > $self->{LIMITE}";
    }
    $#{$self->{DONNEES}} = $nombre;
}
```

SELF->EXTEND(*NOMBRE*)

Perl se sert de la méthode EXTEND pour indiquer que le tableau risque d'augmenter pour accommoder *NOMBRE* éléments. Ainsi vous pouvez tout de suite allouer la mémoire en un gros morceau au lieu de l'allouer plus tard et petit à petit avec de nombreux appels successifs. Comme notre TableauLimite possède une taille limite fixe, nous ne définissons pas cette méthode.

SELF->EXISTS(*INDEX*)

Cette méthode vérifie l'existence dans le tableau lié de l'élément situé à la position *INDEX*. Pour notre TableauLimite, nous ne faisons qu'employer la fonction interne exists de Perl après avoir vérifié qu'il ne s'agit pas d'une tentative de regarder au-delà de la limite supérieure fixée.

```
sub EXISTS  {
    my ($self, $index) = @_;
    if ($index > $self->{LIMITE}) {
        confess "Dépassement de limite du tableau :
                 $index > $self->{LIMITE}";
    }
    exists $self->{DONNEES}[$index];
}
```

SELF->DELETE(*INDEX*)

La méthode DELETE supprime l'élément à la position *INDEX* du tableau lié *SELF*. Dans notre classe TableauLimite, la méthode est presque identique à EXISTS, mais ceci n'est pas toujours le cas.

```
sub DELETE {
    my ($self, $index) = @_;
    print STDERR "suppression !\n";
    if ($index > $self->{LIMITE}) {
        confess "Dépassement de limite du tableau :
                 $index > $self->{LIMITE}";
    }
    delete $self->{DONNEES}[$index];
}
```

SELF->CLEAR

Cette méthode est appelée lorsque le tableau doit être vidé. Cela se produit lorsqu'on affecte une liste de nouvelles valeurs (ou une liste vide) au tableau, mais pas lorsqu'il est soumis à la fonction undef. Comme un TableauLimite vidé satisfait toujours la limite supérieure, nous n'avons rien à vérifier ici.

```
sub CLEAR {
    my $self = shift;
    $self->{DONNEES} = [];
}
```

Si vous affectez une liste au tableau, CLEAR sera déclenchée mais ne verra pas les valeurs de la liste. Donc si vous passez outre la limite supérieure ainsi :

```
tie(@tableau, "TableauLimite", 2);
@tableau = (1, 2, 3, 4);
```

la méthode CLEAR retournera quand même avec succès. L'exception ne sera levée que lors du STORE qui suit. L'affectation déclenche un CLEAR et quatre STORE.

SELF->PUSH(*LISTE*)

Cette méthode ajoute les éléments de *LISTE* à la fin du tableau. Voici à quoi elle pourrait ressembler pour notre classe TableauLimite :

```
sub PUSH    {
    my $self = shift;
    if (@_ + $#{$self->{DONNEES}} > $self->{LIMITE}) {
        confess "Tentative d'ajouter trop d'éléments";
    }
    push @{$self->{DONNEES}}, @_;
}
```

SELF->UNSHIFT(*LISTE*)

Cette méthode insère les éléments de *LISTE* au début du tableau. Dans notre classe TableauLimite, la fonction ressemblerait à PUSH.

SELF->POP

La méthode POP enlève le dernier élément du tableau et le retourne. Pour TableauLimite, c'est une ligne :

```
sub POP { my $self = shift; pop @{$self->{DONNEES}} }
```

SELF->SHIFT

La méthode SHIFT enlève le premier élément de la liste et le retourne. Pour TableauLimite, c'est similaire à POP.

SELF->SPLICE(*POSITION*,*LONGUEUR*,*LISTE*)

Cette méthode vous permet d'épisser (splice) le tableau *SELF*. Pour imiter la fonction interne splice de Perl, *POSITION* devrait être en option et zéro par défaut, et compter en arrière à partir de la fin du tableau s'il est négatif. *LONGUEUR* devrait également être en option, et par défaut la longueur du reste du tableau. *LISTE* peut être vide. Si elle imite correctement la fonction interne, la méthode retournera une liste des *LONGUEUR* éléments à partir de *POSITION* (c'est-à-dire la liste des éléments devant être remplacés par LISTE).

Comme l'épissage est une opération assez complexe, nous ne la définirons pas du tout ; nous ne ferons qu'utiliser la fonction SPLICE du module Tie::Array dont nous disposons sans rien faire en héritant de Tie::Array. De cette manière nous définissons SPLICE en fonction d'autres méthodes de TableauLimite, et donc la vérification de limites aura tout de même lieu.

Ceci complète notre classe TableauLimite. Elle ne fait que légèrement déformer la sémantique des tableaux. Mais nous pouvons faire mieux, et plus court.

Commodité de notation

Une des choses bien avec les variables c'est qu'elles interpolent. Une des choses moins bien avec les fonctions c'est qu'elles ne le font pas. Vous pouvez utiliser un tableau lié pour faire une fonction qui peut être interpolée. Supposons que vous vouliez interpoler des nombres aléatoires dans une chaîne. Vous pouvez juste dire :

```
#!/usr/bin/perl
package InterpAleat;
sub TIEARRAY { bless \my $self };
sub FETCH { int rand $_[1] };

package main;
tie @rand, "InterpAleat";
for (1,10,100,1000) {
    print "Un entier aléatoire plus petit que $_ serait $rand[$_]\n";
}
$rand[32] = 5;    # Cela va-t-il reformater notre disque système ?
```

Lorsqu'il est exécuté, ce programme affiche :

```
Un entier aléatoire plus petit que 1 serait 0
Un entier aléatoire plus petit que 10 serait 5
Un entier aléatoire plus petit que 100 serait 81
Un entier aléatoire plus petit que 1000 serait 765
Can't locate object method "STORE" via package "InterpAleat" at foo line 11.
```

Comme vous le voyez, ce n'est pas si grave de ne pas avoir implémenté `STORE`. Cela explose simplement de façon normale.

Liaison de hachage

Une classe qui implémente un hachage lié doit définir huit méthodes. `TIEHASH` construit de nouveaux objets. `FETCH` et `STORE` donnent accès aux couples clef/valeur. `EXISTS` indique si une clef est présente ou non dans le hachage, et `DELETE` supprime une clef et sa valeur associée.[3] `CLEAR` vide le hachage en supprimant tous les couples clef/valeur. `FIRSTKEY` et `NEXTKEY` itèrent sur les couples clef/valeur lorsque vous appelez `keys`, `values` ou `each`. Et comme d'habitude, si vous voulez exécuter des actions spécifiques lorsque l'objet est désalloué, vous pouvez définir une méthode `DESTROY`. (Si ceci vous semble un grand nombre de méthodes, vous n'avez pas lu avec attention la section précédente sur les tableaux. En tout cas, n'hésitez pas à hériter de méthodes par défaut du module standard `Tie::Hash`, en ne redéfinissant que les méthodes intéressantes. Ici aussi, `Tie::StdHash` s'attend à ce que l'implémentation soit aussi un hachage.)

Par exemple, supposons que vous vouliez créer un hachage pour lequel, à chaque fois que vous affectez une valeur à une clef, au lieu d'écraser la valeur d'origine, la nouvelle valeur soit ajoutée à la fin d'un tableau de valeurs existantes. Comme cela lorsque vous dites :

```
$h{$c} = "un";
$h{$c} = "deux";
```

Cela fait en réalité :

```
push @{ $h{$c} }, "un";
push @{ $h{$c} }, "deux";
```

3. Souvenez-vous que Perl distingue entre une clef absente du hachage et une clef présente dans le hachage mais ayant `undef` comme valeur associée. Les deux possibilités peuvent être testées avec `exists` et `defined`, respectivement.

Cela n'est pas une idée très compliquée, donc vous devriez pouvoir employer un module assez simple, ce que vous pouvez faire en utilisant `Tie::StdHash` comme classe de base. Voici une classe `Tie::HachageRajout` qui fait précisément cela :

```
package Tie::HachageRajout;
use Tie::Hash;
our @ISA = ("Tie::StdHash");
sub STORE {
    my ($self, $clef, $valeur) = @_;
    push @{$self->{$clef}}, $valeur;
}
1;
```

Méthodes de liaison de hachage

Voici un exemple intéressant de classe de liaison de hachage : elle vous donne un hachage qui représente les « fichiers point » d'un utilisateur (c'est-à-dire les fichiers dont le nom commence avec un point, ce qui est une convention de nommage des fichiers de configuration sous Unix). Vous indexez dans le hachage avec le nom du fichier (sans le point) et vous récupérez le contenu du fichier point. Par exemple :

```
use FichiersPoint;
tie %dot, "FichiersPoint";
if ( $point{profile} =~ /MANPATH/ or
     $point{login}   =~ /MANPATH/ or
     $point{cshrc}   =~ /MANPATH/    ) {
    print "il semble que vous spécifiez votre MANPATH\n";
}
```

Voici une autre manière d'utiliser notre classe liée :

```
# Le troisième paramètre est le nom de l'utilisateur
# dont nous examinerons les fichiers point.
tie %lui, "FichiersPoint", "daemon";
foreach $f (keys %lui) {
    printf "le fichier point %s de daemon est de taille %d\n", $f,
            length $lui{$f};
}
```

Dans notre exemple `FichiersPoint` nous implémentons le hachage avec un hachage ordinaire comportant plusieurs champs importants, dont seul le champ `{CONTENU}` contient ce que voit l'utilisateur comme son hachage. Voici les vrais champs de l'objet :

Champ	Contenu
`UTILISATEUR`	Celui dont l'objet représente les fichiers point.
`REPERTOIRE`	Où se situent ces fichiers point.
`ECRASER`	Si l'on a le droit de modifier ou d'écraser ces fichiers point.
`CONTENU`	Le hachage de couples nom de fichier point et contenu.

Voici le début de *FichiersPoint.pm* :

```
package FichiersPoint;
use Carp;
sub quietaisje { (caller(1))[3] . "()" }
my $DEBOGUAGE = 0;
sub deboguage { $DEBOGUAGE = @_ ? shift : 1 }
```

Dans notre exemple, nous voulons pouvoir déclencher l'affichage de messages de débogage pendant le développement de la classe, d'où l'utilité de `$DEBOGUAGE`. Nous gardons aussi sous la main en interne une fonction pratique pour afficher les avertissements : `quietaisje` retourne le nom de la fonction ayant appelé la fonction courante (la fonction « grand-mère » de `quietaisje`).

Voici les méthodes pour le hachage lié `FichiersPoint` :

NOM_CLASSE`->TIEHASH(`*LISTE*`)`

Voici le constructeur de `FichiersPoint` :

```
sub TIEHASH {
    my $self        = shift;
    my $utilisateur = shift || $>;
    my $rep_point   = shift || "";

    croak "usage: @{[ &quietaisje ]} [UTILISATEUR [REP_POINT]]" if @_;

    $utilisateur = getpwuid($utilisateur) if $utilisateur =~ /^\d+$/;
    my $rep = (getpwnam($utilisateur))[7]
           or croak "@{ [&quietaisje] }: pas d'utilisateur $utilisateur";
    $rep .= "/$rep_point" if $rep_point;

    my $noeud = {
        UTILISATEUR => $utilisateur,
        REPERTOIRE  => $rep,
        CONTENU     => {},
        ECRASER     => 0,
    };

    opendir REP, $rep
            or croak "@{[&quietaisje]}: impossible d'ouvrir $rep: $!";
    for my $point ( grep /^\./ && -f "$rep/$_", readdir(REP)) {
        $point =~ s/^\.//;
        $noeud->{CONTENU}{$point} = undef;
    }
    closedir REP;

    return bless $noeud, $self;
}
```

Cela vaut sans doute la peine de mentionner que si vous allez appliquer des tests de fichier sur les valeurs retournées par le `readdir` ci-dessus, vous avez intérêt à les préfixer avec le répertoire concerné (comme nous le faisons). Sans quoi, en l'absence d'un `chdir`, vous risqueriez de ne pas tester le bon fichier.

SELF->FETCH(*CLEF*)

Cette méthode implémente la lecture d'un élément du hachage lié. Elle prend un paramètre après l'objet : la clef dont nous tentons de récupérer la valeur. Cette clef est une chaîne, et vous pouvez faire ce que vous voulez avec (qui soit cohérent avec sa nature de chaîne).

Voici le fetch de notre exemple FichiersPoint :

```
sub FETCH {
    carp &quietaisje if $DEBOGUAGE;
    my $self    = shift;
    my $point   = shift;
    my $rep     = $self->{REPERTOIRE};
    my $fichier = "$rep/.$point";

    unless (exists $self->{CONTENU}->{$point} || -f $fichier) {
        carp "@{[&quietaisje]}: pas de fichier $point" if $DEBOGUAGE;
        return undef;
    }
    # Implémenter un cache.
    if (defined $self->{CONTENU}->{$point}) {
        return $self->{CONTENU}->{$point};
    } else {
        return $self->{CONTENU}->{$point} = `cat $rep/.$point`;
    }
}
```

Nous avons un peu triché en exécutant la commande Unix *cat*(1), mais il serait plus portable (et plus efficace) d'ouvrir le fichier nous-mêmes. D'un autre côté, comme les fichiers point sont un concept unixien, nous ne sommes pas tellement concernés. Ou nous ne devrions pas l'être. Ou quelque chose comme ça...

SELF->STORE(*CLEF*,*VALEUR*)

Cette méthode fait ce qu'il faut lorsqu'un élément du hachage lié est modifié (écrit). Elle prend deux paramètres après l'objet : la clef sous laquelle nous stockons la nouvelle valeur, et la valeur elle-même.

Dans notre exemple FichiersPoint, nous ne permettons pas à un utilisateur d'écrire un fichier sans d'abord invoquer la méthode ecraser sur l'objet d'origine retourné par tie :

```
sub STORE {
    carp &quietaisje if $DEBOGUAGE;
    my $self    = shift;
    my $point   = shift;
    my $valeur  = shift;
    my $fichier = $self->{REPERTOIRE} . "/.$point";
    croak "@{[&quietaisje]}: $fichier non écrasable"
        unless $self->{ECRASER};
    open(F, "> $fichier") or croak "impossible d'ouvrir $fichier: $!";
    print F $valeur;
    close(F);
}
```

Si quelqu'un souhaite écraser quelque chose, il peut dire :

```
$ob = tie %points_daemon, "daemon";
$ob->ecraser(1);
$points_daemon{signature} = "Un vrai démon\n";
```

Mais il pourrait également modifier {ECRASER} à l'aide de `tied` :

```
tie %points_daemon, "FichiersPoint", "daemon";
tied(%points_daemon)->ecraser(1);
```

ou en une instruction :

```
(tie %points_daemon, "FichiersPoint", "daemon")->ecraser(1);
```

La méthode `ecraser` est simplement :

```
sub ecraser {
    my $self = shift;
    $self->{ECRASER} = @_ ? shift : 1;
}
```

SELF->DELETE(*CLEF*)

Cette méthode traite les tentatives de suppression d'un élément du hachage. Si votre hachage émulé utilise quelque part un vrai hachage, vous pouvez simplement appeler le vrai `delete`. De nouveau, nous prenons la précaution de vérifier que l'utilisateur souhaite réellement écraser des fichiers :

```
sub DELETE   {
    carp &quietaisje if $DEBOGUAGE;
    my $self = shift;
    my $point = shift;
    my $fichier = $self->{REPERTOIRE} . "/.$point";
    croak "@{[&quietaisje]}: ne supprime pas le fichier $fichier"
        unless $self->{ECRASER};
    delete $self->{CONTENU}->{$point};
    unlink $fichier or carp "@{[&quietaisje]}:
                              impossible de supprimer $fichier: $!";
}
```

SELF->CLEAR

Cette méthode est exécutée lorsque le hachage entier doit être vidé, habituellement en lui affectant la liste vide. Dans notre exemple, cela supprimerait tous les fichiers point de l'utilisateur ! C'est une chose tellement dangereuse que nous exigerons que ECRASER soit supérieur à 1 avant que cela ne puisse se produire :

```
sub CLEAR {
    carp &quietaisje if $DEBOGUAGE;
    my $self = shift;
    croak "@{[&quietaisje]}: ne supprime pas tous les fichiers point
           de $self->{UTILISATEUR}"
        unless $self->{ECRASER} > 1;
    for my $point ( keys %{$self->{CONTENU}}) {
        $self->DELETE($point);
    }
}
```

SELF->EXISTS(*CLEF*)

Cette méthode est exécutée lorsque l'utilisateur invoque la fonction `exists` sur un hachage donné. Dans notre exemple, nous examinons le hachage `{CONTENU}` pour trouver la réponse :

```
sub EXISTS   {
    carp &quietaisje if $DEBOGUAGE;
    my $self  = shift;
    my $point = shift;
    return exists $self->{CONTENU}->{$point};
}
```

SELF->FIRSTKEY

Cette méthode est appelée lorsque l'utilisateur commence à itérer sur le hachage, comme lors d'un appel à `keys`, `values` ou `each`. En appelant `keys` dans un contexte scalaire, nous réinitialisons son état interne pour nous assurer que le prochain `each` utilisé dans l'instruction `return` obtiendra bien la première clef.

```
sub FIRSTKEY {
    carp &quietaisje if $DEBOGUAGE;
    my $self = shift;
    my $temp = keys %{$self->{CONTENU}};
    return scalar each %{$self->{CONTENU}};
}
```

SELF->NEXTKEY(*CLEF_PRECEDENTE*)

Cette méthode est l'itérateur d'une fonction `keys`, `values` ou `each`. *CLEF_PRECEDENTE* est la dernière clef à laquelle on a accédé, ce que sait fournir Perl. Ceci est utile si la méthode `NEXTKEY` doit connaître l'état précédent pour calculer l'état suivant.

Dans notre exemple, nous utilisons un vrai hachage pour représenter les données du hachage lié, à ceci près que ce hachage est stocké dans le champ `CONTENU` du hachage plutôt que dans le hachage lui-même. Donc nous pouvons compter sur l'opérateur `each` de Perl :

```
sub NEXTKEY  {
    carp &quietaisje if $DEBOGUAGE;
    my $self = shift;
    return scalar each %{ $self->{CONTENU} }
}
```

SELF->DESTROY

Cette méthode est déclenchée lorsque l'objet du hachage lié est sur le point d'être désalloué. Elle n'est vraiment utile que pour le débogage et le nettoyage. En voici une version très simple :

```
sub DESTROY  {
    carp &quietaisje if $DEBOGUAGE;
}
```

Maintenant que nous vous avons donné toutes ces méthodes, voici un exercice à faire à la maison : reprendre tout par le début, trouver les endroits où nous avons interpolé `@{[&quietaisje]}`, et les remplacer par un simple scalaire lié nommé `$quietaisje` qui fait la même chose.

Liaison de handle de fichier

Une classe qui implémente un handle de fichier lié doit définir les méthodes suivantes : `TIEHANDLE`, et l'une au moins de `PRINT`, `PRINTF`, `WRITE`, `READLINE`, `GETC`, ou `READ`. La classe peut également fournir une méthode `DESTROY`, et la définition des méthodes `BINMODE`, `OPEN`, `CLOSE`, `EOF`, `FILENO`, `SEEK`, `TELL`, `READ` et `WRITE` permet au handle de fichier lié d'émuler les fonctions internes Perl correspondantes. (Enfin, ce n'est pas tout à fait vrai : `WRITE` correspond à `syswrite` et n'a rien à voir avec la fonction Perl interne `write`, qui fait de l'affichage à l'aide de déclarations `format`.)

Les handles de fichier liés sont particulièrement utiles lorsque Perl est intégré dans un autre programme (comme Apache ou *vi*) et que l'affichage par `STDOUT` ou `STDERR` doit être redirigé de manière spéciale.

Mais il n'est en fait même pas nécessaire que les handles de fichier soient liés à un fichier. Vous pouvez utiliser des instructions de sortie pour construire une structure de données en mémoire, et des instructions d'entrée pour la récupérer. Voici une manière simple d'inverser une série d'instructions `print` et `printf` sans inverser chaque ligne :

```
package InversePrint;
use strict;
sub TIEHANDLE {
    my $classe = shift;
    bless [], $classe;
}
sub PRINT {
    my $self = shift;
    push @$self, join '', @_;
}
sub PRINTF {
    my $self = shift;
    my $fmt = shift;
    push @$self, sprintf $fmt, @_;
}
sub READLINE {
    my $self = shift;
    pop @$self;
}

package main;
my $m = "--MORE--\n";
tie *INV, "InversePrint";

# Faire des print et des printf.
print INV "Le juge blond est soudainement très malade.$m";

printf INV <<"END", int rand 10000000;
Portez %d vieux whiskys
au juge blond qui fume !
END
```

```
print INV <<"END";
Portez ce vieux whisky
au juge blond qui fume.
END

# Maintenant on lit à partir du même handle.
print while <INV>;
```

Ceci affiche :

```
Portez ce vieux whisky
au juge blond qui fume.
Portez 7387770 vieux whiskys
au juge blond qui fume !
Le juge blond est soudainement très malade.--MORE--
```

Méthodes de liaison de handle de fichier

Pour notre exemple détaillé, nous créerons un handle de fichier qui met en majuscules les chaînes qui lui sont soumises. Pour nous amuser un peu, nous ferons commencer le fichier avec `<CRIER>` lorsqu'il est ouvert, et nous le fermerons avec `</CRIER>` lorsqu'il est fermé. Comme ça nous pourrons divaguer en XML bien formé.

Voici le début du fichier *Crier.pm* qui implémente la classe :

```
package Crier;
use Carp;      # Pour pouvoir signaler les erreurs avec croak
```

Listons à présent les définitions de méthode de *Crier.pm*.

NOM_CLASSE`->TIEHANDLE(`*LISTE*`)`

Ceci est le constructeur de la classe, qui comme d'habitude doit retourner une référence consacrée :

```
sub TIEHANDLE {
    my $classe = shift;
    my $forme = shift;
    open my $self, $forme, @_ or croak "impossible d'ouvrir $forme@_:
                                        $!";
    if ($forme =~ />/) {
        print $self  "<CRIER>\n";
        $$self->{ECRITURE} = 1;  # Se souvenir d'écrire la balise
                                 #de fin
    }
    return bless $self, $classe; # $self est une référence de glob
}
```

Ici, nous ouvrons un nouveau handle de fichier selon le mode et le nom de fichier passés à l'opérateur `tie`, nous écrivons `<CRIER>` dans le fichier, et nous retournons une référence consacrée. Il se passe beaucoup de choses dans cette instruction `open`, mais nous indiquerons simplement que, en plus de l'idiome habituel « open or die », le `my $self` passe un scalaire non défini à `open`, qui sait comment l'autovivifier en un typeglob. Sa nature de typeglob est également intéressante, car le typeglob contient non seulement le véritable objet entrée/sortie du fichier, mais aussi

diverses autres structures de données pratiques, comme un scalaire (`$$$self`), un tableau (`@$$self`), et un hachage (`%$$self`). (Nous ne mentionnerons pas la fonction, `&$$self`.)

La `$forme` est un paramètre indiquant soit le nom de fichier, soit le mode. Si c'est un nom de fichier, `@_` est vide, donc cela se comporte comme un open à deux paramètres. Autrement, `$forme` est le mode pour les paramètres restants.

Après l'ouverture, nous faisons un test pour déterminer si nous devons écrire la balise ouvrante, auquel cas nous le faisons. Et immédiatement nous utilisons une de ces structures de données susmentionnées. Ce `$$self->{ECRITURE}` est un exemple d'utilisation du glob pour stocker une information intéressante. Dans ce cas, nous retenons si nous avons écrit la balise ouvrante afin de savoir si nous devons écrire la balise fermante correspondante. Nous utilisons le hachage `%$$self`, donc nous pouvons donner au champ un nom compréhensible. Nous aurions pu utiliser le scalaire `$$$self`, mais une telle utilisation ne se documenterait pas elle-même. (Ou elle ne ferait *que* se documenter elle-même, selon votre point de vue.)

SELF`->PRINT(`*LISTE*`)`

Cette méthode implémente le `print` sur le handle lié. La *LISTE* contient ce qui a été passé à `print`. Notre méthode ci-dessous met en majuscules chaque élément de la *LISTE* :

```
sub PRINT {
    my $self = shift;
    print $self map {uc} @_;
}
```

SELF`->READLINE`

Cette méthode fournit les données lorsque le handle de fichier est lu avec l'opérateur angle (`<HF>`) ou avec `readline`. La méthode doit retourner `undef` lorsqu'il n'y a plus de données.

```
sub READLINE {
    my $self = shift;
    return <$self>;
}
```

Ici, nous faisons simplement `return <$self>` pour que la méthode se comporte correctement selon qu'elle est appelée en contexte scalaire ou en contexte de liste.

SELF`->GETC`

Cette méthode s'exécute à chaque fois qu'est appelée `getc` sur le handle de fichier lié.

```
sub GETC {
    my $self = shift;
    return getc($self);
}
```

Comme plusieurs des méthodes dans notre classe `Crier`, la méthode `GETC` ne fait qu'appeler sa fonction interne Perl correspondante et retourner le résultat.

SELF`->OPEN(`*LISTE*`)`

Notre méthode `TIEHANDLE` ouvre elle-même le fichier, mais un programme qui utilise la classe `Crier` et appelle ensuite `open` déclenche cette méthode.

```
sub OPEN {
    my $self = shift;
    my $forme = shift;
    my $nom = "$forme@_";
    $self->CLOSE;
    open($self, $forme, @_)  or croak "impossible de rouvrir $nom: $!";
    if ($forme =~ />/) {
        print $self "<CRIER>\n" or croak "impossible de commencer
                                          le print: $!";
        $$self->{ECRITURE} = 1; # Se souvenir d'écrire la balise de fin
    }
    else {
        $$self->{ECRITURE} = 0; # Se souvenir de ne pas écrire
                                # la balise de fin
    }
    return 1;
}
```

Nous invoquons notre propre méthode `CLOSE` pour fermer le fichier explicitement au cas où l'utilisateur ne se serait pas soucié de le faire lui-même. Ensuite nous ouvrons un nouveau fichier avec le nom de fichier spécifié dans le `open`, et nous crions dedans.

SELF->CLOSE

Cette méthode gère les requêtes de fermeture du handle. Ici, nous faisons un `seek` jusqu'à la fin du fichier et, s'il réussit, nous affichons `</CRIER>` avant d'appeler le `close` interne de Perl.

```
sub CLOSE {
    my $self = shift;
    if ($$self->{ECRITURE}) {
        $self->SEEK(0, 2)             or return;
        $self->PRINT("</CRIER>\n")    or return;
    }
    return close $self;
}
```

SELF->SEEK(*LISTE*)

Lorsque vous faites un `seek` sur un handle de fichier lié, la méthode `SEEK` est appelée.

```
sub SEEK {
    my $self = shift;
    my ($position, $depart) = @_;
    return seek($self, $position, $depart);
}
```

SELF->TELL

Cette méthode est invoquée lorsque `tell` est appelée sur le handle de fichier lié.

```
sub TELL {
    my $self = shift;
    return tell $self;
}
```

SELF->PRINTF(*LISTE*)

Cette méthode est exécutée lorsque `printf` est appelée sur le handle de fichier lié. La *LISTE* contient le format et les éléments à afficher.

```
sub PRINTF {
    my $self = shift;
    my $format = shift;
    return $self->PRINT(sprintf $format, @_);
}
```

Ici, nous employons `sprintf` pour produire la chaîne formatée et nous la passons à `PRINT` pour la mise en majuscules. Mais rien ne vous oblige à utiliser la fonction interne `sprintf`. Vous pourriez interpréter les codes pour-cent à votre manière.

SELF->READ(*LISTE*)

Cette méthode répond lorsque le handle est lu avec `read` ou `sysread`. Notez que nous modifions le premier paramètre de *LISTE* « sur place », imitant comment `read` peut remplir le scalaire fourni comme deuxième paramètre.

```
sub READ {
    my ($self, undef, $taille, $position) = @_;
    my $ref_tampon = \$_[1];
    return read($self, $$ref_tampon, $taille, $position);
}
```

SELF->WRITE(*LISTE*)

Cette méthode est invoquée lorsqu'on écrit au handle avec `syswrite`. Ici, nous mettons en majuscules la chaîne écrite.

```
sub WRITE {
    my $self = shift;
    my $chaine = uc(shift);
    my $taille = shift || length $chaine;
    my $position = shift || 0;
    return syswrite $self, $chaine, $taille, $position;
}
```

SELF->EOF

Cette méthode retourne une valeur booléenne lorsque l'état fin-de-fichier du handle de fichier lié à la classe `Crier` est testé avec `eof`.

```
sub EOF {
    my $self = shift;
    return eof $self;
}
```

SELF->BINMODE(*DISC*)

Cette méthode indique la discipline entrée/sortie qui doit être utilisée sur le handle de fichier. Si rien n'est spécifié, elle met le handle de fichier en mode binaire (la discipline `:raw`), pour les systèmes de fichiers qui distinguent entre les fichiers texte et les fichiers binaires.

```
sub BINMODE {
    my $self = shift;
    my $disc = shift || ":raw";
```

```
        return binmode $self, $disc;
    }
```

Voilà comment vous l'écririez, mais elle est en fait inutilisable dans notre cas car open a déjà écrit sur le handle. Donc nous devrions probablement lui faire dire :

```
sub BINMODE { croak("Trop tard pour utiliser binmode") }
```

SELF->FILENO

Cette méthode doit retourner le descripteur de fichier (fileno) associé au handle de fichier par le système d'exploitation.

```
sub FILENO {
    my $self = shift;
    return fileno $self;
}
```

SELF->DESTROY

Comme pour les autres types de liaison, cette méthode est déclenchée lorsque l'objet lié est sur le point d'être détruit. Ceci est utile pour permettre à l'objet de faire du nettoyage. Ici, nous nous assurons que le fichier est fermé, au cas où le programme aurait oublié d'appeler close. Nous pourrions simplement dire close $self, mais il est préférable d'invoquer la méthode CLOSE de la classe. De cette manière si le concepteur de la classe décide de modifier la façon dont les fichiers sont fermés, cette méthode DESTROY n'a pas à être modifiée.

```
sub DESTROY {
    my $self = shift;
    $self->CLOSE;   # Fermer le fichier avec la méthode CLOSE de Crier
}
```

Voici une démonstration de notre classe Crier :

```
#!/usr/bin/perl
use Crier;
tie(*FOO, Crier::, ">nom_fichier");
print FOO "bonjour\n";          # Affiche BONJOUR.
seek FOO, 0, 0;                 # Repart du début.
@lines = <FOO>;                 # Appelle la méthode READLINE.
close FOO;                      # Ferme le fichier explicitement.
open(FOO, "+<", "nom_fichier"); # Rouvre FOO, appelant OPEN.
seek(FOO, 8, 0);                # Saute "<CRIER>\n".
sysread(FOO, $tampon, 7);       # Lit 7 octets de FOO dans $tampon.
print "trouvé $tampon\n";       # Devrait afficher "BONJOUR".
seek(FOO, -7, 1);               # Recule par-dessus "bonjour".
syswrite(FOO, "ciao!\n", 6);    # Ecrit 6 octets dans FOO.
untie(*FOO);                    # Appelle la méthode CLOSE implicitement.
```

Après l'exécution de ceci, le fichier contient :

```
<CRIER>
CIAO!
</CRIER>
```

Voici d'autres choses étranges et merveilleuses à faire avec le glob interne. Nous employons le même hachage que précédemment, mais avec les nouvelles clefs

CHEMIN_FICHIER et DEBOGUAGE. D'abord nous installons une surcharge de conversion en chaîne pour que l'affichage de nos objets révèle le nom de fichier (voir le chapitre 13, *Surcharge*) :

```
# Ceci est vraiment trop génial !
use overload q("") => sub { $_[0]->chemin_fichier };

# Ceci doit être mis dans chaque fonction que vous voulez tracer.
sub trace {
    my $self = shift;
    local $Carp::CarpLevel = 1;
    Carp::cluck("\nméthode magique trace") if $self->deboguage;
}

# Handler de surcharge pour afficher le chemin d'accès au fichier.
sub chemin_fichier {
    my $self = shift;
    confess "je ne suis pas une méthode de classe" unless ref $self;
    $$self->{CHEMIN_FICHIER} = shift if @_;
    return $$self->{CHEMIN_FICHIER};
}
# Deux modes.
sub deboguage {
    my $self = shift;
    my $var = ref $self ? \$$self->{DEBOGUAGE} : \our $Deboguage;
    $$var = shift if @_;
    return ref $self ? $$self->{DEBOGUAGE} || $Deboguage : $Deboguage;
}
```

Ensuite nous appelons trace à l'entrée de toutes nos méthodes ordinaires, comme ceci :

```
sub GETC { $_[0]->trace;         # NOUVEAU
    my($self) = @_;
    getc($self);
}
```

Nous stockons le chemin d'accès du fichier dans les méthodes TIEHANDLE et OPEN :

```
sub TIEHANDLE {
    my $classe = shift;
    my $forme = shift;
    my $nom = "$forme@_";        # NOUVEAU
    open my $self, $forme, @_  or croak "impossible d'ouvrir $nom : $!";
    if ($forme =~ />/) {
        print $self  "<CRIER>\n";
        $$self->{ECRITURE} = 1;  # Se souvenir d'écrire la balise de fin
    }
    bless $self, $classe;        # $self est une référence de glob
    $self->chemin_fichier($nom); # NOUVEAU
    return $self;
}

sub OPEN { $_[0]->trace;         # NOUVEAU
```

```
    my $self = shift;
    my $forme = shift;
    my $nom = "$forme@_";
    $self->CLOSE;
    open($self, $forme, @_)  or croak "impossible de rouvrir $nom : $!";
    $self->chemin_fichier($nom); # NOUVEAU
    if ($forme =~ />/) {
        print $self "<CRIER>\n" or croak "impossible de commencer
                                          le print : $!";
        $$self->{ECRITURE} = 1;  # Se souvenir d'écrire la balise de fin
    }
    else {
        $$self->{ECRITURE} = 0;  # Se souvenir de ne pas écrire la
                                 # balise de fin
    }
    return 1;
}
```

Il faut aussi appeler `$self->deboguage(1)` quelque part pour démarrer le débogage. Lorsque nous faisons cela, tous les appels à `Carp::cluck` produiront des messages informatifs. En voici un que nous obtenons pendant la réouverture ci-dessus. Il nous situe à trois niveaux de profondeur d'appels de méthode, alors que nous fermons l'ancien fichier en préparation pour l'ouverture du nouveau :

```
méthode magique trace at foo line 87
    Crier::SEEK('>nom_fichier', '>nom_fichier', 0, 2) called at foo line 81
    Crier::CLOSE('>nom_fichier') called at foo line 65
    Crier::OPEN('>nom_fichier', '+<', 'nom_fichier') called at foo line 141
```

Handles de fichier originaux

Vous pouvez lier un même handle de fichier à l'entrée et à la sortie d'un tube (pipe) à deux bouts. Supposons que vous vouliez exécuter le programme *bc*(1) (calculateur à précision arbitraire) de cette manière :

```
use Tie::Open2;

tie *CALC, 'Tie::Open2', "bc -l";
$somme = 2;
for (1 .. 7) {
    print CALC "$somme * $somme\n";
    $somme = <CALC>;
    print "$_: $somme";
    chomp $somme;
}
close CALC;
```

Vous vous attendriez à ce qu'il affiche ceci :

```
1: 4
2: 16
3: 256
```

```
4: 65536
5: 4294967296
6: 18446744073709551616
7: 340282366920938463463374607431768211456
```

Vous auriez raison de vous attendre à cela si vous aviez le programme *bc*(1) sur votre ordinateur, et si vous aviez défini `Tie::Open2` de la manière suivante. Cette fois-ci nous utiliserons un tableau consacré comme objet interne. Il contient nos deux handles de fichier pour la lecture et pour l'écriture. (La tâche difficile d'ouverture d'un tube à deux bouts est faite par `IPC::Open2` ; nous ne faisons que la partie amusante.)

```
package Tie::Open2;
use strict;
use Carp;
use Tie::Handle;  # ne pas hériter de ceci !
use IPC::Open2;

sub TIEHANDLE {
    my ($classe, @cmd) = @_;
    no warnings 'once';
    my @paire_hf = \do { local(*LECTURE, *ECRITURE) };
    bless $_, 'Tie::StdHandle' for @paire_hf;
    bless(\@paire_hf => $classe)->OPEN(@cmd) || die;
    return \@paire_hf;
}

sub OPEN {
    my ($self, @cmd) = @_;
    $self->CLOSE if grep {defined} @{ $self->FILENO };
    open2(@$self, @cmd);
}

sub FILENO {
    my $self = shift;
    [ map { fileno $self->[$_] } 0,1 ];
}

for my $meth_sortie ( qw(PRINT PRINTF WRITE) ) {
    no strict 'refs';
    *$meth_sortie = sub {
        my $self = shift;
        $self->[1]->$meth_sortie(@_);
    };
}
for my $meth_entree ( qw(READ READLINE GETC) ) {
    no strict 'refs';
    *$meth_entree = sub {
        my $self = shift;
        $self->[0]->$meth_entree(@_);
    };
}
```

```
for my $meth_double ( qw(BINMODE CLOSE EOF)) {
    no strict 'refs';
    *$meth_double = sub {
        my $self = shift;
        $self->[0]->$meth_double(@_) && $self->[1]->$meth_double(@_);
    };
}
for my $meth_morte ( qw(SEEK TELL)) {
    no strict 'refs';
    *$meth_morte = sub {
        croak("impossible d'appliquer $meth_morte à un tube");
    };
}
1;
```

Les quatre dernières boucles sont incroyablement chouettes, à notre avis. Pour une explication de ce qui se passe, voir la section *Fermetures comme modèles de fonctions* au chapitre 8.

Voici un ensemble de classes encore plus bizarres. Les noms de paquetages devraient vous donner une idée de ce qu'elles font :

```
use strict;
package Tie::DevNull;

    sub TIEHANDLE {
        my $classe = shift;
        my $hf = local *HF;
        bless \$hf, $classe;
    }
    for (qw(READ READLINE GETC PRINT PRINTF WRITE)) {
        no strict 'refs';
        *$_  = sub { return };
    }

package Tie::DevAleatoire;

    sub READLINE { rand() . "\n"; }
    sub TIEHANDLE {
        my $classe = shift;
        my $hf = local *HF;
        bless \$hf, $classe;
    }

    sub FETCH { rand() }
    sub TIESCALAR {
        my $classe = shift;
        bless \my $self, $classe;
    }

package Tie::Tee;
```

```
sub TIEHANDLE {
    my $classe = shift;
    my @handles;
    for my $chemin (@_) {
        open(my $hf, ">$chemin") || die "impossible d'écrire $chemin";
        push @handles, $hf;
    }
    bless \@handles, $classe;
}

sub PRINT {
    my $self = shift;
    my $ok = 0;
    for my $hf (@$self) {
        $ok += print $hf @_;
    }
    return $ok == @$self;
}
```

La classe `Tie::Tee` émule le programme standard Unix *tee*(1) qui envoie un flux en sortie vers plusieurs différentes destinations. La classe `Tie::DevNull` émule le périphérique nul `/dev/null` des systèmes Unix. La classe `Tie::DevRandom` produit des nombres aléatoires soit comme handle, soit comme scalaire, selon que vous appelez `TIEHANDLE` ou `TIESCALAR` ! Voici comment vous les appelez :

```
package main;

tie *DISPERSE,  "Tie::Tee", qw(tmp1 - tmp2 >tmp3 tmp4);
tie *ALEATOIRE, "Tie::DevAleatoire";
tie *NULL,      "Tie::DevNull";
tie my $alea,   "Tie::DevAleatoire";

for my $i (1..10) {
    my $ligne = <ALEATOIRE>;
    chomp $ligne;
    for my $hf (*NULL, *DISPERSE) {
        print $hf "$i: $ligne $alea\n";
    }
}
```

Ce qui produit quelque chose comme ceci sur votre écran :

```
1: 0.124115571686165 0.20872819474074
2: 0.156618299751194 0.678171662366353
3: 0.799749050426126 0.300184963960792
4: 0.599474551447884 0.213935286029916
5: 0.700232143543861 0.800773751296671
6: 0.201203608274334 0.0654303290639575
7: 0.605381294683365 0.718162304090487
8: 0.452976481105495 0.574026269121667
9: 0.736819876983848 0.391737610662044
10: 0.518606540417331 0.381805078272308
```

Mais ce n'est pas tout ! Le programme a écrit sur votre écran à cause du - dans le `tie` de `*DISPERSE` ci-dessus. Mais cette ligne lui a également dit de créer les fichiers *tmp1*, *tmp2*, et *tmp4*, et d'ajouter à la fin de *tmp3*. (Nous avons aussi écrit au handle de fichier `*NULL` dans la boucle, mais bien sûr cela n'est apparu nulle part d'intéressant, à moins que vous vous intéressiez aux trous noirs.)

Un piège subtil du déliement

Si vous avez l'intention de vous servir de l'objet retourné par `tie` ou par `tied`, et que la classe définit un destructeur, il y a un piège subtil contre lequel vous devez vous protéger. Examinez cet exemple (tordu, avouons-le) d'une classe qui utilise un fichier pour tenir un log de toutes les valeurs affectées à un scalaire :

```
package Souvenir;

sub TIESCALAR {
    my $classe = shift;
    my $nom_fichier = shift;
    open(my $handle, ">", $nom_fichier)
         or die "Impossible d'ouvrir $nom_fichier: $!\n";
    print $handle "Le Début\n";
    bless {HF => $handle, VALEUR => 0}, $classe;
}

sub FETCH {
    my $self = shift;
    return $self->{VALEUR};
}

sub STORE {
    my $self = shift;
    my $valeur = shift;
    my $handle = $self->{HF};
    print $handle "$valeur\n";
    $self->{VALEUR} = $valeur;
}

sub DESTROY {
    my $self = shift;
    my $handle = $self->{HF};
    print $handle "La Fin\n";
    close $handle;
}

1;
```

Voici un exemple qui se sert de notre classe `Souvenir` :

```
use strict;
use Souvenir;
```

```
my $fred;
tie $fred, "Souvenir", "chameau.log";
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat chameau.log";
```

Voici ce qu'il affiche lorsqu'il est exécuté :

```
Le Début
1
4
5
La Fin
```

Jusqu'ici tout va bien. Rajoutons une méthode supplémentaire à la classe `Souvenir` qui permet les commentaires dans le fichier — disons, quelque chose comme ceci :

```
sub commenter {
    my $self = shift;
    my $message = shift;
    print { $self->{HF} } $handle $message, "\n";
}
```

Et voici l'exemple précédent, modifié pour utiliser la méthode `commenter` :

```
use strict;
use Souvenir;

my ($fred, $x);
$x = tie $fred, "Souvenir", "chameau.log";
$fred = 1;
$fred = 4;
commenter $x "change...";
$fred = 5;
untie $fred;
system "cat chameau.log";
```

Maintenant le fichier risque d'être vide ou incomplet, ce qui n'est probablement pas ce que vous souhaitiez faire. Voici pourquoi. Lier une variable l'associe à l'objet retourné par le constructeur. Cet objet n'a normalement qu'une référence : celle cachée derrière la variable liée. Appeler « `untie` » rompt l'association et élimine cette référence. Comme il ne reste pas de référence à l'objet, la méthode `DESTROY` est déclenchée.

Cependant, dans l'exemple ci-dessus nous avons stocké dans `$x` une deuxième référence à l'objet lié. Cela veut dire qu'il reste une référence valide à l'objet après le `untie`. `DESTROY` n'est pas déclenchée, et le fichier n'est ni enregistré, ni fermé. C'est pour cela que rien n'est sorti : le tampon du handle de fichier était encore en mémoire. Il n'atteindra le disque qu'à la sortie du programme.

Pour détecter ceci, vous pourriez utiliser l'option de ligne de commande **`-w`** ou inclure le pragma `use warnings "untie"` dans la portée lexicale courante. L'une ou l'autre technique décèlerait un appel à `untie` pendant qu'il reste des références à l'objet lié. Dans ce cas, Perl affiche cet avertissement :

```
untie attempted while 1 inner references still exist
```

Pour faire marcher correctement le programme et faire taire l'avertissement, éliminez les références supplémentaires à l'objet lié *avant* d'appeler `untie`. Vous pouvez le faire explicitement :

```
undef $x;
untie $fred;
```

Souvent vous pouvez résoudre le problème en vous assurant simplement que vos variables quittent leur portée au bon moment.

Modules de liaison sur CPAN

Avant de devenir tout inspiré à l'idée d'écrire votre propre module de liaison, vous devriez vérifier que quelqu'un ne l'a pas déjà fait. Il y a beaucoup de modules de liaison sur CPAN, et il y en a des nouveaux tous les jours. (Enfin, chaque mois, en tout cas.) Le *tableau 14-1* en liste quelques-unes.

Tableau 14-1. Modules de liaison sur CPAN

Module	Description
`GnuPG::Tie::Encrypt`	Lie une interface de handle de fichier avec GNU Privacy Guard.
`IO::WrapTie`	Emballe les objets liés avec une interface `IO::Handle`.
`MLDBM`	Stocke de manière transparente des données complexes, pas juste de simples chaînes, dans un fichier DBM.
`Net::NISplusTied`	Lie des hachages à des tables NIS+.
`Tie::Cache::LRU`	Implémente un cache utilisant l'algorithme « Least Recently Used » (moins récemment utilisé).
`Tie::Const`	Fournit des scalaires et des hachages constants.
`Tie::Counter`	Enchante une variable scalaire pour qu'elle s'incrémente à chaque fois qu'on y accède.
`Tie::CPHash`	Implémente un hachage insensible à la casse mais qui préserve la casse.
`Tie::DB_FileLock`	Fournit un accès verrouillé aux bases de données Berkeley DB 1.x.
`Tie::DBI`	Lie des hachages aux bases de données relationnelles DBI.
`Tie::DB_Lock`	Lie des hachages aux bases de données avec des verrouillages exclusifs et partagés.
`Tie::Dict`	Lie un hachage à un serveur dict RPC.
`Tie::Dir`	Lie un hachage pour lire des répertoires.
`Tie::DirHandle`	Lie des handles de répertoire.
`Tie::FileLRUCache`	Implémente un cache LRU léger, basé sur le système de fichiers.
`Tie::FlipFlop`	Implémente une liaison qui alterne entre deux valeurs.
`Tie::HashDefaults`	Permet à un hachage d'avoir des valeurs par défaut.
`Tie::HashHistory`	Garde l'historique de toutes les modifications d'un hachage.
`Tie::IxHash`	Fournit à Perl des tableaux associatifs ordonnés.

Tableau 14-1. Modules de liaison sur CPAN

Module	**Description**
`Tie::LDAP`	Implémente une interface à une base de données LDAP.
`Tie::Persistent`	Fournit des structures de données persistantes via `tie`.
`Tie::Pick`	Choisit (et supprime) aléatoirement un élément d'un ensemble.
`Tie::RDBM`	Lie des hachages aux bases de données relationnelles.
`Tie::SecureHash`	Permet l'encapsulation avec les espaces de noms.
`Tie::STDERR`	Envoie la sortie de votre `STDERR` à un autre processus tel qu'un système d'envoi de courrier.
`Tie::Syslog`	Lie un handle de fichier pour automatiquement enregistrer sa sortie avec syslog.
`Tie::TextDir`	Lie un répertoire de fichiers.
`Tie::TransactHash`	Modifie un hachage par transactions sans changer l'ordre pendant la transaction.
`Tie::VecArray`	Donne à un vecteur de bits une interface de tableau.
`Tie::Watch`	Place des témoins sur les variables Perl.
`Win32::TieRegistry`	Fournit des manières faciles et puissantes de manipuler la base de registres de Microsoft Windows.

III

Technique Perl

15

Unicode

Si vous ne savez pas encore ce qu'est Unicode, vous le saurez bientôt — même si vous ne lisez pas ce chapitre — car travailler avec Unicode devient indispensable. (Certains pensent que c'est une nécessité maléfique mais c'est bien davantage une nécessité bénéfique. Dans tous les cas, c'est un mal nécessaire.)

Historiquement, les gens ont construit des jeux de caractères qui reflétaient ce qu'ils devaient faire dans le contexte de leur propre culture. Comme les gens de toutes les cultures sont naturellement paresseux, ils ont eu tendance à n'inclure que les symboles dont ils avaient besoin en excluant ceux qui ne leur étaient pas utiles. Cela fonctionnait à merveille aussi longtemps que nous ne communiquions qu'avec d'autres gens appartenant à notre propre culture, mais maintenant que nous commençons à utiliser l'Internet pour des communications interculturelles, nous nous heurtons à des problèmes du fait de cette approche exclusive. Il est déjà difficile d'imaginer comment taper des caractères accentués sur un clavier américain. Comment dans le monde (littéralement) peut-on écrire une page web multilingue ?

Unicode est la solution, ou tout au moins une partie de la solution (voir également XML). Unicode est un jeu de caractères inclusif plutôt qu'exclusif. Alors que les gens peuvent chicaner, et ne s'en privent pas, sur les divers détails d'Unicode (et il y a une grande quantité de détails sur lesquels chicaner), l'intention générale est de contenter tout le monde de manière suffisante[1] avec Unicode pour qu'ils utilisent de leur propre gré Unicode comme le médium international d'échange de données textuelles. Personne ne vous force à utiliser Unicode, tout comme personne ne vous force à lire ce chapitre (du moins, nous l'espérons). Les gens pourront toujours utiliser leurs anciens jeux de caractères exclusifs dans leur propre culture. Mais dans ce cas (comme on dit), la portabilité en souffrira.

La loi de la conservation de la souffrance dit que si nous réduisons la souffrance quelque part, elle doit augmenter autre part. Dans le cas d'Unicode, nous devons souffrir de la migration d'une sémantique d'octet vers une sémantique de caractère. Puisque, par un

1. Ou dans certains cas, de mécontenter tout le monde de manière insuffisante.

accident de l'histoire, il a été inventé par un Américain, Perl a historiquement confondu les notions d'octet et de caractère. En migrant vers Unicode, Perl ne doit plus les confondre d'une manière ou d'une autre.

Paradoxalement, en amenant Perl à ne plus confondre les octets et les caractères, nous permettons aux programmeurs Perl de les confondre car ils savent que Perl rétablira la distinction, exactement comme ils peuvent confondrent confondre les nombres et les chaînes et s'en remettre à Perl pour convertir autant que nécessaire les uns vers les autres et réciproquement. Autant que possible, Perl a une approche d'Unicode semblable à son approche de tout le reste : Faire Ce Qu'il Faut Faire. Idéalement, nous aimerions parvenir à ces quatre objectifs :

Objectif n° 1

Les anciens programmes orientés octet ne devraient pas spontanément se planter avec les anciennes données orientées octet avec lesquelles ils travaillaient.

Objectif n° 2

Les anciens programmes orientés octet devraient par magie commencer à fonctionner avec les nouvelles données orientées caractère lorsque c'est approprié.

Objectif n° 3

Les programmes devraient tourner aussi vite avec les nouvelles données orientées caractère qu'avec les anciennes données orientées octet.

Objectif n° 4

Perl devrait rester un langage unifié plutôt que se scinder en un Perl orienté octet et un Perl orienté caractère.

Pris ensemble, ces objectifs sont pratiquement impossibles à atteindre. Mais nous nous en approchons relativement bien. Ou plutôt, nous sommes toujours dans le processus de rapprochement, puisqu'il s'agit d'une tâche en cours de réalisation. Perl continue d'évoluer en même temps qu'Unicode. Mais notre plan d'envergure est d'offrir une voie de migration qui soit sûre et qui puisse nous amener là où nous voulons aller en rencontrant le minimum d'accidents en cours de route. Comment y arriver est le sujet du prochain paragraphe.

Construction d'un caractère

Dans les versions de Perl antérieures à 5.6, toutes les chaînes étaient vues en tant que séquences d'octets. Dans les versions 5.6 et suivantes, une chaîne peut toutefois contenir des caractères d'une taille supérieure à un octet. Nous voyons maintenant les chaînes, non plus en tant que séquences d'octets, mais en tant que séquences de numéros compris dans l'intervalle `0 .. 2**32-1` (ou dans le cas d'ordinateurs 64 bits, `0 .. 2**64-1`). Ces numéros représentent des caractères abstraits et, dans un certain sens, plus grand est le numéro, plus la taille du caractère est grande ; mais au contraire de beaucoup de langages, Perl n'est pas lié à une taille particulière de la représentation d'un caractère. Perl utilise un encodage de taille variable (basé sur UTF8), ainsi ces numéros de caractères abstraits peuvent être empaquetés, ou non, à raison d'un numéro par octet. Évidemment, le numéro de caractère `18 446 744 073 709 551 515` (c'est-à-dire, « `\x{ffff_ffff_ffff_ffff}` ») ne tiendra jamais dans un octet (en fait, il prend 13 oc-

tets) mais si tous les caractères de votre chaîne se situent dans l'intervalle `0..127` en décimal, alors ils seront certainement empaquetés à raison d'un par octet, puisque l'UTF8 est identique à l'ASCII pour les sept premiers bits de poids faible.

Perl utilise de l'UTF8 lorsqu'il pense que c'est bénéfique, ainsi, si tous les caractères de votre chaîne sont dans l'intervalle `0..255`, il y a une grande chance que les caractères soient empaquetés dans des octets — mais en l'absence d'autre informations connues, vous ne pouvez être sûr car Perl fait en interne une conversion entre les caractères 8 bits et les caractères UTF8 de taille variable si nécessaire. L'essentiel est que vous ne devriez pas vous en soucier la plupart du temps car la sémantique de caractère est préservée à un niveau abstrait indépendamment de la représentation.

Dans tous les cas, si votre chaîne contient des numéros de caractères supérieurs à `255` en décimal, la chaîne est certainement stockée en UTF8. Plus exactement, elle est stockée dans la version étendue d'UTF8 de Perl que nous appelons *utf8*, en hommage au pragma du même nom, mais surtout parce que c'est plus simple à taper. (Et de plus, l'UTF8 « réel » n'est autorisé à contenir que des numéros de caractères consacrés par le Consortium Unicode. L'*utf8* de Perl est autorisé à contenir tous les caractères dont vous avez besoin pour accomplir votre travail. Perl se moque du fait que vos numéros de caractères soient officiellement corrects ou juste corrects.)

Nous avons dit que vous ne devriez pas vous en soucier la plupart du temps, mais les gens aiment de toute façon se faire du souci. Supposez que vous ayez une v-chaîne représentant une adresse IPv4 :

```
$adr_loc = v127.0.0.1;     # Sûrement stockée en octets.
$oreilly = v204.148.40.9;  # Pourrait être stockée en octets ou en utf8.
$adr_nok = v2004.148.40.9; # Sûrement stockée en utf8.
```

Tout le monde peut s'apercevoir que `$adr_nok` ne marchera pas en tant qu'adresse IP. Il est donc facile de penser que si l'adresse réseau d'O'Reilly est forcée à une représentation UTF8, elle ne marchera plus. Mais les caractères dans la chaîne sont des numéros abstraits, non des octets. Tout ce qui utilise une adresse IPv4, comme la fonction `gethostbyaddr`, devrait automatiquement forcer les numéros de caractères abstrait à être reconvertis vers une représentation en octets (et échouer avec `$adr_nok`).

Les interfaces entre Perl et le monde réel doivent s'accommoder des détails de la représentation. Jusqu'à un certain point, les interfaces existantes essaient de faire ce qu'il faut sans que vous ayez à leur dire quoi faire. Mais de temps en temps, vous devez donner des instructions à certaines interfaces (comme la fonction `open`) et si vous écrivez votre propre interface vers le monde réel, elle aura besoin d'être soit assez astucieuse pour arriver à comprendre les choses par elle-même, soit au moins assez élégante pour suivre les instructions lorsque que vous voulez qu'elle se comporte différemment de ce qu'elle aurait fait par défaut.[2]

2. Sur certains systèmes, il peut y avoir des moyens de basculer toutes vos interfaces en une seule fois. Si l'option de la ligne de commande `-C` est utilisée (ou la variable globale `${^WIDE_SYSTEM_CALLS}` est positionnée à `1`), tous les appels système utiliseront les API de « caractères larges » (*wide character*). (Ceci n'est actuellement implémenté que sur Microsoft Windows.) Le projet actuel de la communauté Linux est que toutes les interfaces basculeront en mode UTF8 si `$ENV{LC_TYPE}` est positionnée à « `UTF8` ». Les autres communautés peuvent adopter d'autres approches. Ce n'est pas une règle absolue.

Puisque Perl se soucie de maintenir une sémantique de caractère transparente à l'intérieur du langage lui-même, la seule place où vous devez vous inquiéter d'une sémantique d'octet, opposée à une sémantique de caractère, est dans vos interfaces. Par défaut, toutes vos anciennes interfaces Perl vers le monde extérieur sont orientées octet. C'est-à-dire qu'à un niveau abstrait, toutes vos chaînes sont des séquences de numéros dans l'intervalle `0..255`, ainsi, si rien dans le programme ne les force à des représentations en utf8, vos anciens programmes continueront à fonctionner avec des données orientées octet comme ils le faisaient auparavant. Vous pouvez donc cocher l'objectif n°1 ci-dessus.

Si vous voulez que vos anciens programmes fonctionnent avec les nouvelles données orientées caractère, vous devez marquer vos interfaces orientées caractère de manière que Perl s'attende à recevoir des données orientées caractère de ces interfaces. Une fois que c'est fait, Perl devrait automatiquement faire toutes les conversions nécessaires pour préserver l'abstraction des caractères. La seule différence est que vous devez introduire dans votre programme des chaînes qui soient marquées comme contenant potentiellement des caractères plus grands que `255`, ainsi, si vous effectuez une opération entre une chaîne d'octets et une chaîne utf8, Perl convertira en interne la chaîne d'octets en une chaîne utf8 avant d'effectuer l'opération. Généralement, les chaînes utf8 ne sont reconverties en chaînes d'octets que lorsque vous les envoyez à une interface orientée octet, auquel cas, si vous aviez une chaîne contenant des caractères plus grands que `255`, vous aurez un problème qui peut être résolu de diverses manières selon l'interface en question. Vous pouvez donc cocher l'objectif n°2.

Parfois, vous voulez mélanger du code comprenant une sémantique de caractère avec du code devant tourner avec une sémantique d'octet, comme du code d'entrées/sorties qui lit ou écrit des blocs de taille fixe. Dans ce cas, vous pouvez mettre une déclaration `use bytes` autour du code orienté octet pour le forcer à utiliser une sémantique d'octet même sur les chaînes marquées en tant que chaînes utf8. Les conversions sont alors sous votre responsabilité. Mais c'est un moyen de renforcer une lecture locale plus stricte de l'objectif n°1, au dépend d'une lecture globale plus relâchée de l'objectif n°2.

L'objectif n°3 a largement été atteint, en partie en faisant des conversions paresseuses entre les représentations en octets et en utf8 et en partie en étant discret sur la manière dont nous avons implémenté potentiellement des fonctionnalités lentes d'Unicode, comme la recherche des propriétés de caractères dans des tables énormes.

L'objectif n°4 a été atteint en sacrifiant un peu de la compatibilité des interfaces à la quête des autres objectifs. D'une certaine façon, nous n'avons pas bifurqué vers deux Perl différents ; mais en le considérant d'une autre manière, la révision 5.6 de Perl *est* une version bifurquée de Perl au regard des versions antérieures et nous ne nous attendons pas à ce que les gens migrent depuis leurs versions antérieures avant d'être sûrs que la nouvelle version fera bien ce qu'ils veulent. Mais c'est toujours le cas avec les nouvelles versions, nous nous permettrons donc de cocher également l'objectif n°4.

Conséquences de la sémantique de caractère

Le résultat de tout ceci est qu'un opérateur interne typique manipulera des caractères sauf s'il est dans la portée d'un pragma `use bytes`. Toutefois, même en dehors de la portée d'un `use bytes`, si tous les opérandes de l'opérateur sont stockés en tant que carac-

tères 8 bits (c'est-à-dire qu'aucune opérande n'est stockée en tant qu'utf8), alors la sémantique de caractère se confond avec la sémantique d'octet et le résultat de l'opérateur sera stocké en interne dans un format 8 bits. Ceci préserve une compatibilité antérieure pourvu que vous n'approvisionniez pas vos programmes avec des caractères plus larges qu'en Latin-1.

Le pragma `utf8` est avant tout un dispositif de compatibilité permettant la reconnaissance de l'UTF8 dans les littéraux et les identificateurs rencontrés par l'analyseur grammatical. Il peut également être utilisé pour permettre certaines fonctionnalités des plus expérimentales d'Unicode. Notre objectif à long terme est de changer le pragma utf8 en une opération ne faisant rien (*no-op*).

Le pragma `use bytes` ne sera jamais changé en une opération ne faisant rien. Non seulement il est nécessaire pour le code orienté octet, mais un de ses effets de bords est également de définir des enrobages orientés octet autour de certaines fonctions pour les utiliser en dehors d'une portée de `use bytes`. Au moment où nous écrivons ceci, le seul enrobage défini est pour la fonction `length` mais d'autres sont susceptibles de l'être au fil du temps. Pour utiliser cet enrobage, écrivez :

```
use bytes ();   # Charge les enrobages sans importer la sémantique d'octet.
...
$lg_car = length("\x{ffff_ffff}");         # Renvoie 1.
$lg_oct = bytes::length("\x{ffff_ffff}");  # Renvoie 7.
```

En dehors de la portée d'une déclaration `use bytes`, la version 5.6 de Perl fonctionne (ou du moins, est censée fonctionner) ainsi :

- Les chaînes et les motifs peuvent maintenant contenir des caractères ayant une valeur ordinale supérieure à 255 :

  ```
  use utf8;
  $convergence = "☞ ☜";
  ```

  En supposant que vous ayez un éditeur compatible avec Unicode pour éditer votre programme, de tels caractères se présenteront généralement directement à l'intérieur de chaînes littérales en tant que caractères UTF8. Pour le moment, vous devez déclarer un `use utf8` au début de votre programme pour permettre l'utilisation de l'UTF8 dans les littéraux.

  Si vous ne disposez pas d'éditeur Unicode, vous pouvez toujours spécifier un caractère particulier en ASCII avec une extension de la notation `\x`. Un caractère dans l'intervalle Latin-1 peut être écrit comme `\x{ab}` ou comme `\xab` mais si le numéro dépasse les deux chiffres hexadécimaux, vous devez utiliser des accolades. Les caractères Unicode sont spécifiés en mettant le code hexadécimal à l'intérieur des accolades après le `\x`. Par exemple, un smiley Unicode vaut `\x{263A}`. Il n'y a pas de construction syntaxique en Perl présumant que les caractères Unicode fassent exactement 16 bits, vous ne pouvez donc pas utiliser `\u263A` comme dans certains langages ; `\x{263A}` est l'équivalent le plus proche.

  Pour insérer des caractères nommés *via* `\N{`*`NOM_CAR`*`}`, voir le pragma `use charnames` au chapitre 31, *Modules de pragmas*.

- Les identificateurs à l'intérieur du script Perl peuvent contenir des caractères Unicode alphanumériques, y compris des idéogrammes :

  ```
  use utf8;
  ```

```
$人++; # Un enfant est né.
```

Ici encore, `use utf8` est nécessaire (pour l'instant) pour reconnaître de l'UTF8 dans votre script. Vous êtes actuellement livré à vous-même lorsqu'il s'agit d'utiliser les formes canoniques des caractères — Perl ne tente pas (encore) de canoniser pour vous les noms de variable. Voir *www.unicode.org* pour le dernier rapport sur la canonisation.

- Les expressions régulières correspondent à des caractères au lieu d'octets. Par exemple un point correspond à un caractère au lieu de correspondre à un octet. Si le Consortium Unicode arrive même à approuver l'emploi du twengar, alors (malgré le fait que de tels caractères sont représentés avec 4 octets en UTF8), ceci réussi la correspondance :

  ```
  "\N{TENGWAR LETTER SILME NUQUERNA}" =~ /^.$/
  ```

  Le motif `\C` est fourni pour forcer une correspondance à un seul octet (« `char` » en C, d'où `\C`). Utilisez `\C` avec prudence, puisqu'il peut désynchroniser des caractères dans votre chaîne et vous pouvez obtenir des erreurs « Caractère UTF8 mal formé ». Vous ne pouvez pas utiliser `\C` dans des crochets, puisqu'il ne représente aucun caractère particulier, ni aucun ensemble de caractères.

- Les classes de caractères dans les expressions régulières correspondent à des caractères au lieu d'octets et correspondent avec les propriétés des caractères spécifiées dans la base de données des propriétés Unicode. Ainsi, `\w` peut être utilisé pour correspondre à un idéogramme :

  ```
  "人" =~ /\w/
  ```

- Les propriétés nommées d'Unicode et les intervalles de blocs peuvent être utilisés comme classes de caractères avec les nouvelles constructions `\p` (correspond à la propriété) et `\P` (ne correspond pas à la propriété). Par exemple, `\p{Lu}` correspond à tout caractère ayant la propriété majuscule d'Unicode, alors que `\p{M}` correspond à tout caractère marqué. Les propriétés d'une seule lettre peuvent omettre les accolades, ainsi `\pM` peut également correspondre à des caractères marqués. Beaucoup de classes de caractères prédéfinies sont disponibles, comme `\p{IsMirrored}` et `\p{InTibetan}` :

  ```
  "\N{greek:Iota}" =~ /p{Lu}/
  ```

  Vous pouvez également utiliser `\p` et `\P` à l'intérieur des crochets des classes de caractères. (Dans la version 5.6.0 de Perl, vous avez besoin de faire `use utf8` pour que les propriétés des caractères fonctionnent bien. Cette restriction sera levée à l'avenir.) Voir le chapitre 5, *Correspondance de motifs*, pour plus de détails sur la correspondance de propriétés Unicode.

- Le motif spécial `\X` correspond à toute séquence Unicode étendue (une « séquence de caractères combinés » en langage standard), où le premier caractère est un caractère de base et les suivants des caractères marqués s'appliquant au caractère de base. C'est équivalent à `(?:\PM\pM*)` :

  ```
  "o\N{COMBINING TILDE BELOW}" =~ /\X/
  ```

  Vous ne pouvez pas utiliser `\X` à l'intérieur de crochets car il pourrait correspondre à plusieurs caractères et il ne représente aucun caractère particulier, ni aucun ensemble de caractères.

- L'opérateur `tr///` traduit des caractères au lieu d'octets. Pour changer tous les caractères hors de l'intervalle Latin-1 en points d'interrogation, vous pourriez écrire :

```
tr/\0-\x{10ffff}/\0-\xff?/; # caractère utf8 vers latin1
```

- Les opérateur de traduction de casse utilise des tables de traduction de casse Unicode lorsqu'on leur fournit des caractères en entrée. Remarquer que `uc` traduit en majuscules, alors qu'`ucfisrt` traduit l'initiale en majuscule de titre (*titlecase*), pour les langages qui font la distinction. Naturellement les séquences d'antislashs correspondantes ont la même sémantique :

```
$x = "\u$mot";     # met l'initiale de $mot en majuscule de titre
$x = "\U$mot";     # met $mot en majuscules
$x = "\l$mot";     # met l'initiale de $mot en minuscule
$x = "\L$mot";     # met $mot en minuscules
```

  Soyez prudent, car les tables de traduction de casse Unicode ne tentent pas de fournir des tables de correspondance bijectives pour chaque instance, particulièrement pour les langages utilisant un nombre de caractères différent pour les majuscules et les minuscules. Comme ils disent dans les standards, alors que les propriétés de casse sont elles-mêmes normatives, les tables de casse ne sont qu'informelles.

- La plupart des opérateurs manipulant des positions ou des longueurs dans une chaîne basculeront automatiquement à l'utilisation de positions de caractères, ce qui comprend : `chop`, `substr`, `pos`, `index`, `rindex`, `sprintf`, `write` et `length`. Les opérateurs qui ne basculent pas délibérément comprennent `vec`, `pack` et `unpack`. Les opérateurs qui s'en moquent vraiment comprennent `chomp`, ainsi que tout autre opérateur traitant une chaîne comme un paquet de bits, tels que le `sort` par défaut et les opérateurs manipulant des fichiers.

```
use bytes;
$lg_octets = length("Je fais de l'合氣道"); # 15 octets
no bytes;
$lg_caracs = length("Je fais de l'合氣道"); # mais 9 caractères
```

- Les lettres « `c` » et « `C` » de `pack`/`unpack` ne changent *pas*, puisqu'elles sont souvent utilisées pour des formats orientés octet. (Encore une fois, pensez « `char` » en langage C.) Toutefois, il existe un nouveau spécificateur « `U` », qui fera les conversions entre les caractères UTF8 et les entiers :

```
pack("U*", 1, 20, 300, 4000) eq v1.20.300.4000
```

- Les fonctions `chr` et `ord` fonctionnent avec des caractères :

```
chr(1).chr(20).chr(300).chr(4000) eq v1.20.300.4000
```

  En d'autres termes, `chr` et `ord` sont comme `pack("U")` et `unpack("U")` et non comme `pack("C")` et `unpack("C")`. En fait, ces derniers sont un moyen pour vous d'émuler les fonctions `chr` et `ord` orientées octet si vous êtes trop paresseux pour faire un `use bytes`.

- Enfin, un `reverse` en contexte scalaire inverse par caractère plutôt que par octet :

```
"☞ ☜" eq reverse "☜ ☞"
```

Si vous regardez dans le répertoire *`CHEMIN_BIBLIOTHEQUE_PERL/unicode`*, vous trouverez de nombreux fichiers en rapport avec la définition des sémantiques ci-dessus. La base de données des propriétés Unicode provenant du Consortium Unicode se situe

dans un fichier appelé *Unicode.300* (pour Unicode 3.0). Ce fichier a déjà été traité par *mktables.PL* pour donner un tas de petits fichiers *.pl* dans le même répertoire (et dans les sous-répertoires *Is/*, *In/* et *To/*), dont certains sont automatiquement avalés par Perl pour implémenter des choses comme `\p` (voir les répertoires *Is/* et *In/*) et `uc` (voir le répertoire *To/*). Les autres fichiers sont avalés par des modules comme le pragma `use charnames` (voir *Name.pl*). Mais au moment où nous écrivons ceci, il reste encore de nombreux fichiers qui sont juste assis là sans rien faire, en attendant que vous leur écriviez un module d'accès :

ArabLink.pl
ArabLnkGrp.pl
Bidirectional.pl
Block.pl
Category.pl
CombiningClass.pl
Decomposition.pl
JamoShort.pl
Number.pl
To/Digit.pl

Un résumé bien plus lisible d'Unicode, avec beaucoup de liens hypertextes, se trouve dans *CHEMIN_BIBLIOTHEQUE_PERL/unicode/Unicode3.html*.

Remarquez que lorsque le Consortium Unicode sort une nouvelle version, certains de ces noms de fichier sont susceptibles de changer, vous devrez donc fouiller un peu. Vous pouvez trouver *CHEMIN_BIBLIOTHEQUE_PERL* avec l'incantation suivante :

```
% perl -MConfig -le 'print $Config{privlib}'
```

Pour trouver tout ce que l'on peut trouver sur Unicode, vous devriez consulter *The Unicode Standard Version 3.0* (ISBN 0-201-61633-5).

Attention 人 au travail

Au moment où nous écrivons ceci (c'est-à-dire, en phase avec la version 5.6.0 de Perl), il existe encore des précautions à prendre pour utiliser Unicode. (Consultez votre documentation en ligne pour les mises à jour.)

- Le compilateur d'expressions régulières existant ne produit pas de codes d'opérations (*opcodes*) polymorphiques. Cela signifie que le fait de déterminer si un motif particulier correspondra ou non à des caractères Unicode est réalisé lorsque le motif est compilé (basé sur le fait que le motif contient ou non des caractères Unicode) et non lorsque la correspondance arrive à l'exécution. Ceci doit être changé pour correspondre de manière adaptée à de l'Unicode si la chaîne sur laquelle se fait la correspondance est en Unicode.
- Il n'y a actuellement pas de moyen facile de marquer une donnée lue depuis un fichier ou depuis tout autre source externe comme étant en utf8. Ceci sera un domaine sur lequel se concentrer dans un futur proche et ce sera probablement déjà corrigé au moment où vous lirez ceci.

- Il n'y a pas de méthode pour forcer automatiquement l'entrée et la sortie vers d'autres encodages que UTF8. Ceci est toutefois prévu dans un futur proche, consultez donc votre documentation en ligne.
- L'utilisation de locales avec utf8 peut conduire à d'étranges résultats. Actuellement, il existe quelques tentatives pour appliquer l' information d'une locale 8 bits à des caractères dans l'intervalle `0..255` mais ceci est de façon évidente incorrect pour des locales qui utilisent des caractères au-delà de cet intervalle (lorsqu'ils sont convertis en Unicode). Cela aura également tendance à tourner plus lentement. Le bannissement des locales est fortement encouragé.

L'Unicode est « fun » — il faut juste que vous définissiez le fun correctement.

16

Communication interprocessus

Les processus peuvent communiquer de presque autant de façons que les personnes. Mais les difficultés de la communication interprocessus ne devraient pas être sous-estimées. Il ne vous sert à rien d'attendre une réplique verbale si votre partenaire n'utilise que le langage des signes. De même, deux processus ne peuvent se parler que lorsqu'ils s'accordent sur la méthode de communication et sur les conventions bâties sur cette méthode. Comme avec tout type de communication, l'éventail des conventions sur lesquelles s'accorder va du lexical au pragmatique : tout, depuis quel jargon vous utiliserez jusqu'au choix de celui dont c'est le tour de parler. Ces conventions sont nécessaires car il est très difficile de communiquer avec une sémantique pure, dépouillée de tout contexte.

Dans notre jargon, la communication interprocessus s'épelle habituellement IPC (*Inter-Process Communication*). Les fonctionnalités d'IPC de Perl vont du très simple au très complexe. Votre choix dépend de la complexité des informations à communiquer. La forme la plus simple d'information est, d'une certaine manière, aucune information, sinon l'avertissement qu'un certain événement s'est produit à un certain moment. En Perl, ces événements sont communiqués par un mécanisme de signal inspiré de celui du système UNIX.

À l'autre extrême, les fonctionnalités de socket de Perl permettent de communiquer avec n'importe quel autre processus de l'Internet grâce à n'importe quel protocole mutuellement supporté. Cette liberté a bien sûr un prix : vous devez parcourir un certain nombre d'étapes pour mettre en place la connexion et vous assurer que vous parlez le même langage que le processus à l'autre bout. Ce qui peut vous forcer à adhérer à d'étranges coutumes, selon les conventions culturelles en usage. Il se peut même que pour être correct au niveau du protocole, vous deviez même vous mettre à parler un langage comme XML, Java ou Perl. Quelle horreur !

Entre ces deux extrêmes, on trouve des fonctionnalités intermédiaires conçues à l'origine pour faire communiquer des processus sur la même machine. Il s'agit notamment des bons vieux fichiers, des pipes (ou tubes), des FIFO et les divers appels système d'IPC System V. L'implémentation de ces fonctionnalités varie selon les plates-formes ; les sys-

tèmes Unix modernes (y compris le MAC OS X d'Apple) devraient toutes les implémenter et, mis à part les signaux et les IPC System V, la plupart des autres sont implémentées sur les système d'exploitation récents de Microsoft, y compris les pipes, les forks, les verrouillages de fichiers et les sockets.[1]

Vous pouvez trouver de plus amples informations sur le portage en général dans l'ensemble de documentations standard de Perl (dans n'importe quel format affiché par votre système) sous *perlport*. Les informations spécifiques à Microsoft peuvent être trouvées sous *perlwin32* et *perlfork*, qui sont installés même sur des systèmes non-Microsoft. Pour les livres publiés, nous vous suggérons les suivants :

- *Perl en action*, par Tom Christiansen et Nathan Torkington (O'Reilly Associates, 1998), chapitres 16 à 18.
- *Advanced Programming in the UNIX Environment*, par W. Richard Stevens (Addison-Wesley, 1992).
- *TCP/IP Illustrated*, par W. Richard Stevens, Volumes I-III (Addison-Wesley, 1992-1996).

Signaux

Perl utilise un modèle simple de gestion des signaux : le hachage `%SIG` contient des références (qu'elles soient symboliques ou en dur) à des gestionnaires de signaux définis par l'utilisateur. Certains événements entraînent le système d'exploitation à délivrer un signal au processus affecté. Le gestionnaire correspondant à cet événement est appelé avec un argument contenant le nom du signal qui l'a déclenché. Pour envoyer un signal à un autre processus, utilisez la fonction `kill`. Pensez-y comme à une information d'un bit envoyée à l'autre processus.[2] Si ce processus a installé un gestionnaire de signal pour le signal en question, il peut exécuter du code lorsqu'il le reçoit. Mais il n'y a aucun moyen pour le processus émetteur d'obtenir une quelconque sorte de valeur de retour, autre que de savoir si le signal a été légitimement envoyé. L'émetteur ne reçoit aucun retour lui disant ce que le processus récepteur a fait du signal, s'il en a fait quelque chose.

Nous avons classé cette fonctionnalité dans les IPC alors qu'en fait, les signaux peuvent provenir de diverses sources et pas seulement d'autres processus. Le signal pourrait également provenir du même processus, ou bien il pourrait être généré lorsque l'utilisateur tape au clavier une séquence particulière comme Ctrl-C ou Ctrl-Z, ou il pourrait être fabriqué par le noyau quand certains événements se produisent, comme la fin d'un processus fils, ou lorsque le processus n'a plus assez d'espace sur la pile, ou parce qu'un fichier a atteint sa taille limite en mémoire. Mais votre propre processus ne sait pas distinguer facilement ces différents cas. Un signal ressemble à un paquet qui arrive mystérieusement sur le pas de la porte sans l'adresse de l'expéditeur. Vous feriez mieux d'être prudent au moment de l'ouvrir.

Puisque les entrées dans le hachage `%SIG` peuvent être des références en dur, une tech-

1. Enfin, sauf les sockets `AF_UNIX`.

2. En fait, il s'agit plus de cinq ou six bits, selon le nombre de signaux définis par votre système d'exploitation et si l'autre processus utilise le fait que vous n'avez *pas* envoyé un signal différent.

nique courante est d'utiliser des fonctions anonymes pour les gestionnaires de signaux simples :

```
$SIG{INT}  = sub { die "\nIci\n" };
$SIG{ALRM} = sub { die "Votre minuterie a expiré" };
```

Ou vous pourriez créer une fonction nommée et affecter son nom ou une référence à l'entrée appropriée du hachage. Par exemple, pour intercepter les signaux d'interruption et de sortie (souvent associés aux touches Ctrl-C et Ctrl-\ sur votre clavier), mettez en place un gestionnaire comme ceci :

```
sub intercepte {
    my $nom_sig = shift;
    our $ben_voyons++;
    die "Quelqu'un m'a envoyé un SIG$nom_sig !";
}
$ben_voyons = 0;
$SIG{INT} = 'intercepte';    # signifie toujours &main::intercepte
$SIG{INT} = \&intercepte;    # la meilleure stratégie
$SIG{QUIT} = \&intercepte;   # en intercepte également un autre
```

Remarquez que tout ce que nous faisons dans le gestionnaire de signaux est de positionner une variable globale et de lever une exception avec `die`. Évitez autant que possible de faire quelque chose de plus compliqué que cela car sur la plupart des systèmes, la bibliothèque C n'est pas réentrante. Les signaux sont délivrés de manière asynchrone,[3] ainsi, l'appel à une quelconque fonction `print` (voire à tout ce qui a besoin d'allouer plus de mémoire avec *malloc*(3)) pourrait en théorie déclencher une erreur de mémoire et engendrer un core dump si vous étiez déjà dans une routine apparentée de la bibliothèque C lorsque le signal a été délivré. (Même la routine `die` est quelque peu dangereuse sauf si le processus est exécuté à l'intérieur d'un `eval`, ce qui supprime les entrées/sorties depuis `die` et par conséquent empêche d'appeler la bibliothèque C. Probablement.)

Une manière encore plus simple d'intercepter les signaux est d'utiliser le pragma `sigtrap` pour installer de simples gestionnaires de signaux par défaut :

```
use sigtrap qw(die INT QUIT);
use sigtrap qw(die untrapped normal-signals
   stack-trace any error-signals);
```

Le pragma est très utile lorsque vous ne voulez pas vous embêter à écrire votre propre gestionnaire mais que vous voulez toujours intercepter les signaux dangereux et tout fermer dans les règles. Par défaut, certains de ces signaux ont un caractère si fatal pour votre processus que votre programme s'arrêtera en pleine course lorsqu'il en recevra un. Malheureusement, cela signifie que les fonctions `END` gérant la terminaison du programme et les méthodes `DESTROY` s'occupant de l'achèvement d'un objet ne seront pas appelées. Mais *elles le seront* sur des exceptions Perl ordinaires (comme lorsque vous appelez `die`), vous pouvez donc utiliser ce pragma pour convertir sans douleur les signaux en

3. La synchronisation de l'émission de signaux avec les codes d'opérations (*opcodes*) au niveau de Perl est planifiée pour une prochaine version de Perl, ce qui devrait régler les questions de signaux et de core dumps.

exceptions. Même si vous ne manipulez pas de signaux vous-même, votre programme devra bien se comporter quand même. Voir la description de `use sigtrap` au Chapitre 31, *Modules de pragmas*, pour beaucoup plus de fonctionnalités de ce pragma.

Vous pouvez également choisir d'affecter au gestionnaire de `%SIG` les chaînes « `IGNORE` » ou « `DEFAULT` », auquel cas Perl essayera d'écarter le signal ou permettra que se produise l'action par défaut pour ce signal (bien que certains signaux ne peuvent être ni interceptés ni ignorés, comme les signaux `KILL` et `STOP` ; voir *signal*(3), s'il est implémenté, pour obtenir une liste des signaux disponibles sur votre système et leur comportement par défaut).

Le système d'exploitation voit les signaux comme des nombres plutôt que comme des noms, mais Perl, comme la plupart des gens, préfère les noms symboliques aux nombres magiques. Pour trouver les noms des signaux, passez en revue les clés du hachage `%SIG` ou utilisez la commande *kill -l* si vous en disposez sur votre système. Vous pouvez également utiliser le module standard `Config` pour déterminer la correspondance que fait votre système entre les noms et les numéros des signaux. Voir *Config*(3) pour des exemples.

Puisque `%SIG` est un hachage global, les affections que vous y effectuez se répercutent sur tout votre programme. Il est souvent plus respectueux pour le reste de votre programme de confiner l'interception des signaux à une portée réduite. Faites cela avec une affectation en `local` du gestionnaire de signaux, ce qui n'a plus d'effet une fois que l'on sort du bloc l'encadrant. (Mais souvenez-vous que les valeurs avec `local` sont visibles depuis les fonctions appelées depuis ce bloc.)

```
{
    local $SIG{INT} = 'IGNORE';
    ...     # Faites ce que vous voulez ici, en ignorant tous les SIGINT.
    fn();   # Les SIGINT sont également ignorés dans fn() !
    ...     # Et ici.
}           # La fin du bloc restaure la valeur précédente de $SIG{INT}.
fn();       # Les SIGINT ne sont pas ignorés dans fn() (a priori).
```

Envoi de signaux à des groupes de processus

Les processus (du moins sous Unix) sont organisés en groupes de processus, correspondant généralement à une tâche complète. Par exemple, lorsque vous lancez une seule commande shell consistant en une série de filtres envoyant les données de l'un à l'autre à travers des pipes, ces processus (et leurs fils) appartiennent au même groupe. Ce groupe de processus possède un numéro correspondant au numéro du processus leader de ce groupe. Si vous envoyez un signal à un numéro de processus positif, il n'est transmis qu'à ce processus ; mais si vous l'envoyez à un numéro négatif, le signal est transmis à tous les processus dont le numéro de groupe est le nombre positif correspondant, c'est-à-dire le numéro de processus du leader de ce groupe. (De façon bien pratique, pour le leader du groupe, l'ID du groupe de processus est simplement `$$`.)

Imaginez que votre programme veuille envoyer un signal « hang-up » à tous ses processus fils, ainsi qu'à tous les petits-fils lancés par ces fils, ainsi qu'à tous les arrières-petits-fils lancés par ces petits-fils, ainsi qu'à tous les arrières-arrières-petits-fils lancés par ces arrières-petits-fils, et ainsi de suite. Pour faire cela, votre programme appelle d'abord

`setpgrp(0,0)` pour devenir le leader d'un nouveau groupe de processus et tous les processus qu'il créera ensuite feront partie de ce nouveau groupe. Il importe peu que ces processus soient démarrés manuellement *via* `fork` ou automatiquement *via* des `open` sur des pipes ou encore en tant que tâches d'arrière-plan avec `system("cmd &")`. Même si ces processus ont leurs propres fils, l'envoi d'un signal « hang-up » au groupe de processus entier les trouvera tous (sauf pour les processus qui ont positionné leur propre groupe de processus ou changé leur UID pour s'auto-accorder une immunité diplomatique envers vos signaux).

```
{
    local $SIG{HUP} = 'IGNORE';    # m'exempte
    kill(HUP, -$$);                # envoie le signal à mon propre
                                   # groupe de processus
}
```

Un autre signal intéressant est le signal numéro `0`. En fait, ce dernier n'affecte pas le processus cible mais vérifie plutôt s'il est toujours en vie et s'il n'a pas changé son UID. C'est-à-dire qu'il vérifie s'il est légal de lui envoyer un signal sans le faire effectivement.

```
unless (kill 0 => $pid_fils) {
    warn "quelque chose d'affreux est arrivé à $pid_fils";
}
```

Le signal numéro `0` est le seul qui fonctionne de la même façon sous les portages sur Microsoft comme il le fait sous Unix. Sur les systèmes Microsoft, `kill` ne délivre pas effectivement de signal. À la place, il force le processus fils à se terminer avec le statut indiqué par le numéro de signal. Il se peut que cela soit corrigé un jour. Cependant, le signal magique `0` a toujours le comportement par défaut, non-destructeur.

Fossoyage des zombies

Lorsqu'un processus se termine, son père reçoit un signal `CHLD` envoyé par le noyau et le processus devient un zombie[4] jusqu'à ce que le parent appelle `wait` ou `waitpid`. Si vous démarrez un autre processus en Perl en utilisant n'importe quoi d'autre que `fork`, Perl s'occupe de fossoyer vos fils zombies mais si vous utilisez un `fork` pur, vous êtes censés nettoyer derrière vous. Sur la plupart des noyaux, mais pas tous, une bidouille simple pour fossoyer automatiquement les zombies consiste à positionner `$SIG{CHLD}` à `'IGNORE'`. Une approche plus souple (mais fastidieuse) est de les fossoyer vous-mêmes. Comme il se peut qu'il y ait plus d'un fils qui soit mort avant que vous ne vous en occupiez, vous devez rassembler vos zombies dans une boucle jusqu'à ce qu'il n'y en ait plus :

```
use POSIX ":queue";
sub FOSSOYEUR { 1 until (waitpid(-1, WNOHANG) == -1) }
```

Pour lancer ce code comme il se faut, vous pouvez soit positionnez un gestionnaire pour le signal `CHLD` :

```
$SIG{CHLD} = \&FOSSOYEUR;
```

ou, si vous êtes dans une boucle, arrangez-vous simplement pour appeler le fossoyeur

4. Si, si c'est vraiment le terme technique consacré.

de temps à autre. Ceci est la meilleure approche car elle n'est pas sujette au core dump occasionnel que les signaux peuvent parfois déclencher dans la bibliothèque C. Toutefois, elle s'avère coûteuse si on l'appelle dans une boucle concise, un compromis raisonnable consiste donc à utiliser une stratégie hybride dans laquelle vous minimisez le risque à l'intérieur du gestionnaire en en faisant le moins possible et en attendant sur l'extérieur pour fossoyer les zombies :

```
our $zombies = 0;
$SIG{CHLD} = sub { $zombies++ };
sub fossoyeur {
    my $zombie;
    our %Statut_Fils;   # Stocke chaque code de retour
    $zombies = 0;
    while (($zombie = waitpid(-1, WNOHANG)) != -1) {
        $Statut_Fils{$zombie} = $?;
    }
}
while (1) {
    fossoyeur() if $zombies;
    ...
}
```

Ce code présume que votre noyau implémente des signaux fiables. Ce n'était traditionnellement pas le cas pour les anciens SysV, ce qui rendait impossible l'écriture de gestionnaires de signaux corrects. Depuis la version 5.003, Perl utilise, là où il est disponible, l'appel système *sigaction*(2), qui est bien plus digne de confiance. Cela signifie qu'à moins que vous ne tourniez sur un ancien système ou avec un ancien Perl, vous n'aurez pas à réinstaller vos gestionnaires et risquez de manquer des signaux. Heureusement, tous les systèmes de la famille BSD (y compris Linux, Solaris et Mac OS X) et les systèmes compatibles POSIX fournissent des signaux fiables, le comportement dégradé des anciens SysV est donc plus une remarque historique qu'un problème d'actualité.

Avec ces nouveaux noyaux, beaucoup d'autres choses fonctionnent également mieux. Par exemple, les appels système « lents » (ceux qui peuvent bloquer, comme `read`, `wait` et `accept`) redémarreront automatiquement s'ils sont interrompus par un signal. Au mauvais vieux temps, le code de l'utilisateur devait se souvenir de vérifier explicitement si chaque appel système lent avait échoué ou non, avec `$!` (`$ERRNO`) positionné à `EINTR` et, si c'était le cas, le relancer. Ceci ne se produisait pas seulement avec les signaux `INT` ; même des signaux inoffensifs comme `TSTP` (provenant d'un Ctrl-Z) ou `CONT` (en passant la tâche en premier plan) auraient fait avorter l'appel système. Perl redémarre maintenant l'appel système automatiquement pour vous si le système d'exploitation le permet. Ceci est généralement décrit comme étant une fonctionnalité.

Vous pouvez vérifier si vous disposez ou non d'un comportement des signaux plus rigoureux, dans le style de POSIX, en chargeant le module `Config` et en testant si `$Config{d_sigaction}` est à vrai. Pour trouver si les appels système lents sont redémarrables ou non, vérifiez la documentation de votre système à propos de *sigaction*(2) ou *sigvec*(3) ou dénichez `SV_INTERRUPT` ou `SA_RESTART` dans votre fichier C *sys/signal.h*. Si vous trouvez l'un de ces symboles ou les deux, vous avez certainement des appels système redémarrables.

Minutage des opérations lentes

Une utilisation courante des signaux consiste à imposer une limite de temps aux opérations s'exécutant lentement. Si vous êtes sur un système Unix (ou sur tout autre système compatible POSIX implémentant le signal `ALRM`), vous pouvez demander au noyau d'envoyer un signal `ALRM` à votre processus à un moment à venir précis :

```
use Fcntl ':flock';
eval {
    local $SIG{ALRM} = sub { die "redémarrage de la minuterie" };
    alarm 10;                # planifie une alarme dans 10 secondes
    eval {
        flock(HF, LOCK_EX)    # un verrou exclusif bloquant
            or die "flock impossible : $!";
    };
    alarm 0;                 # annule l'alarme
};
alarm 0;              # protection contre une situation de concurrence
die if $@ && $@ !~ /redémarrage de la minuterie/;    # relance
```

Si l'alarme est déclenchée pendant que vous attendez le verrou et que vous ne faites qu'intercepter le signal avant de sortir, vous retournerez directement dans le `flock` car Perl relance automatiquement les appels système là où c'est possible. La seule solution est de lever une exception avec `die` et de laisser ensuite `eval` l'intercepter. (Ceci fonctionne car l'exception aboutit à un appel de la fonction *longjmp*(3) de la bibliothèque C et c'est réellement ce qui vous dispense de relancer l'appel système.)

La capture d'exception imbriquée existe car l'appel de `flock` lèverait une exception si `flock` n'était pas implémenté sur votre plate-forme et vous devez vous assurez d'annuler l'alarme dans tous les cas. Le second `alarm 0` sert dans le cas où le signal arrive après avoir lancé le `flock` mais avant d'atteindre le premier `alarm 0`. Sans le second `alarm 0`, vous prendriez le risque d'avoir une légère situation de concurrence (*N.d.T.* : *race condition*) — mais le poids ne compte pas dans les situations de concurrence ; soit il y en a, soit il n'y en a pas. Et nous préférons qu'il n'y en ait pas.

Blocage des signaux

De temps à autre, vous aimeriez bien retarder la réception d'un signal durant une section de code critique. Vous ne voulez pas ignorer le signal aveuglément mais ce que vous êtes en train de faire est trop important pour être interrompu. Le hachage `%SIG` de Perl n'implémente pas de blocage des signaux, contrairement au module `POSIX`, à travers son interface à l'appel système *sigprocmask*(2) :

```
use POSIX qw(:signal_h);
$ensemble_signaux = POSIX::SigSet->new;
$ensemble_bloques = POSIX::SigSet>new(SIGINT,SIGQUIT,SIGCHLD);
sigprocmask(SIG_BLOCK, $ensemble_bloques, $ensemble_signaux)
    or die "Impossible de bloquer les signaux INT,QUIT,CHLD : $!\n";
```

Une fois que les trois signaux sont bloqués vous pouvez faire ce que vous voulez sans craindre d'être embêté. Lorsque vous en avez fini avec votre section critique, débloquez les signaux en restaurant l'ancien masque de signaux :

```
sigprocmask(SIG_SETMASK, $ensemble_signaux)
    or die "Impossible de restaurer les signaux INT, QUIT, CHLD : $!\n";
```

Si l'un de ces trois signaux était arrivé alors qu'ils étaient bloqués, ils sont délivrés immédiatement. Si deux signaux ou plus sont en attente, l'ordre de délivrance est indéfini. De plus, on ne fait aucune distinction entre le fait de recevoir une seule fois un signal particulier que l'on a bloqué et le recevoir plusieurs fois.[5] Par exemple, si neuf processus fils se terminaient pendant que vous bloquiez les signaux `CHLD`, votre gestionnaire (si vous en aviez un) ne serait appelé qu'une seule fois après avoir débloqué les signaux. C'est pourquoi, lorsque vous fossoyez les zombies, vous devez toujours boucler jusqu'à ce qu'ils soient tous partis.

Fichiers

Peut-être n'aviez-vous jamais pensé aux fichiers comme étant un mécanisme d'IPC, mais ils se taillent la part du lion dans les communications interprocessus — loin devant tous les autres procédés réunis. Lorsqu'un processus dépose ses précieuses données dans un fichier et qu'un autre processus les extrait, ces processus ont communiqué entre eux. Les fichiers offrent quelque chose d'unique parmi toutes les formes d'IPC couvertes ici : comme un rouleau de papyrus enseveli dans le désert et déterré après des millénaires, un fichier peut être déterré et lu longtemps après le décès de celui qui l'a écrit.[6] En travaillant sur la persistance tout en étant d'une facilité d'utilisation comparable, il n'est pas surprenant que les fichiers soient toujours populaires.

L'emploi de fichiers, pour transmettre des informations depuis un passé révolu vers un futur incertain, amène quelques surprises. Vous écrivez le fichier sur un médium permanent, tel qu'un disque, et c'est tout. (Vous pourriez dire à un serveur web où le trouver, s'il contient du HTML.) Cela devient un challenge intéressant lorsque tous les protagonistes sont toujours en vie et essaient de communiquer entre eux. Sans une convention à propos de celui dont le temps de parole est arrivé, toute communication fiable est impossible ; l'accord peut être conclu *via* le verrouillage de fichier, qui sera abordé dans le prochain paragraphe. Dans le paragraphe suivant, nous parlerons de la relation spéciale existant entre un processus père et ses fils, qui permet à toutes les protagonistes d'échanger de l'information *via* l'héritage de l'accès aux mêmes fichiers.

Les fichiers présentent toutefois des limites quand il est question d'accès distant, de synchronisation, de fiabilité et de suivi des sessions. D'autres paragraphes de ce chapitre présentent divers mécanismes d'IPC inventés pour pallier ces limites.

Verrouillage de fichier

Dans un environnement multitâche, vous devez faire attention à ne pas rentrer en collision avec d'autres processus essayant d'utiliser le même fichier que vous. Tant que tous les processus ne font que lire, il n'y a aucun problème, mais dès qu'au moins un d'entre

5. C'est-à-dire, traditionnellement. Les signaux comptabilisables seront peut-être implémentés sur des systèmes temps-réel d'après les dernières spécifications mais nous n'avons encore jamais vus de tels systèmes.

6. Si tant est qu'un processus puisse décéder.

eux a besoin d'écrire dans le fichier, il s'en suit un chaos complet à moins qu'un quelconque mécanisme de verrouillage ne vienne faire la police.

Ne vous contentez pas de l'existence d'un fichier (c'est-à-dire, `-e $fichier`) comme une indication de verrouillage car il existe une situation de concurrence (*race condition*) entre le test de l'existence de ce nom de fichier et ce que vous avez prévu de faire avec (comme le créer, l'ouvrir ou l'effacer). Voir le paragraphe « Gérer les situations de concurrences » au chapitre 23, *Sécurité*, pour plus d'informations à ce sujet.

L'interface portable de Perl pour le verrouillage est la fonction `flock(`*`HANDLE, DRAPEAUX`*`)`, décrite au chapitre 29, *Fonctions*. Pour une portabilité maximale, Perl n'utilise que les fonctionnalités de verrouillage les plus simples et les plus largement répandues que l'on puisse trouver sur le plus grand éventail de plates-formes. Cette sémantique est suffisamment simple pour être émulée sur la plupart des systèmes, y compris ceux qui n'implémentent pas l'appel traditionnel du même nom, comme System V ou Windows NT. (Cependant, si vous tournez sur un système Microsoft antérieur à NT, vous n'avez probablement aucune chance, vous n'en auriez pas plus si vous étiez sur un système Apple antérieur à Mac OS X.)

Il existe deux variétés de verrous : les verrous partagés (le drapeau `LOCK_SH`) et les verrous exclusifs (le drapeau `LOCK_EX`). Malgré le nom intimidant « exclusif », les processus ne sont pas obligés d'obéir aux verrous sur les fichiers. C'est-à-dire que `flock` n'implémente qu'un *verrouillage consultatif*, signifiant que le verrouillage d'un fichier n'empêche pas un autre processus de lire, ni même d'écrire dans ce fichier. Une demande de verrou exclusif n'est qu'une manière pour un processus de permettre au système d'exploitation de bloquer cette demande jusqu'à ce qu'il n'y ait plus de verrous, qu'ils soient partagés ou exclusifs. De même, lorsqu'un processus demande un verrou partagé, il ne fait que se bloquer lui-même jusqu'à ce qu'il n'y ait plus de verrous exclusifs. Un fichier en concurrence ne peut être accessible de manière sûre que lorsque tous les protagonistes utilisent le mécanisme de verrouillage de fichier.

Ainsi, `flock` est une opération bloquante par défaut. C'est-à-dire que si vous ne pouvez obtenir immédiatement le verrou que vous désirez, le système d'exploitation bloque votre processus jusqu'à ce que ce soit possible. Voici comment obtenir un verrou partagé, bloquant, généralement utilisé pour lire un fichier :

```
use Fcntl qw(:DEFAULT :flock);
open(HF, "< nom_fichier") or die "impossible d'ouvrir nom_fichier : $!";
flock(HF, LOCK_SH)  or die "impossible de verrouiller nom_fichier : $!";
# maintenant, lisez le fichier
```

Vous pouvez essayer d'acquérir un verrou de manière non-bloquante en incluant le drapeau `LOCK_NB` dans la requête `flock`. S'il est impossible de vous donner le verrou immédiatement, la fonction échoue et renvoie instantanément faux. Voici un exemple :

```
flock(HF, LOCK_SH | LOCK_NB)
    or die "impossible de verrouiller nom_fichier : $!";
```

Vous pouvez désirer quelque chose d'autre que la levée d'une exception comme nous l'avons fait ici mais vous n'oserez certainement pas faire d'entrées/sorties sur le fichier. Si le verrou vous est refusé, vous ne devriez pas accéder au fichier jusqu'à ce que vous l'obteniez. Qui sait dans quel état de confusion vous pourriez trouver le fichier ? L'intérêt principal du mode non-bloquant est de vous laisser sortir faire quelque chose d'autre

pendant l'attente. Mais il peut également s'avérer utile pour produire des interactions plus amicales en avertissant les utilisateurs que cela pourrait prendre du temps pour obtenir le verrou afin qu'ils ne se sentent pas abandonnés :

```
use Fcntl qw(:DEFAULT :flock);
open(HF, "< nom_fichier") or die "impossible d'ouvrir nom_fichier : $!";
unless (flock(HF, LOCK_SH | LOCK_NB)) {
    local $| = 1;
    print "Attente d'un verrou sur nom_fichier...";
    flock(HF, LOCK_SH) or die "impossible de verrouiller nom_fichier : $!";
    print "obtenu.\n";
}
# maintenant, lisez dans HF
```

Certains seront tentés de mettre ce mode non-bloquant dans une boucle. Le problème majeur avec le mode non-bloquant est que, pendant que vous revenez pour vérifier à nouveau, quelqu'un d'autre a pu s'emparer du verrou parce que vous avez abandonné votre place dans la file d'attente. Parfois vous n'avez qu'à vous mettre dans la file et patienter. Si vous avez de la chance, il y aura quelques magazines à lire.

Les verrous s'appliquent sur les handles de fichiers, non sur les noms de fichiers.[7] Lorsque vous fermez le fichier, le verrou se dissout automatiquement, que vous ayez fermé le fichier explicitement en appelant `close` ou implicitement en le rouvrant ou en quittant votre processus.

Pour obtenir un verrou exclusif, généralement utilisé pour écrire dans un fichier, vous devez faire attention. Vous ne pouvez pas utiliser pour cela un `open` ordinaire ; si vous utilisez un mode d'ouverture de `<`, il échouera sur les fichiers qui n'existent pas encore et si vous utilisez `>`, il écrasera les fichiers existant déjà. Utilisez plutôt `sysopen` sur le fichier afin qu'il puisse être verrouillé avant d'être écrasé. Une fois que vous avez ouvert le fichier en toute sécurité pour y écrire mais que vous n'y avez pas encore touché, obtenez un verrou exclusif et *alors seulement*, tronquez le fichier. Maintenant vous pouvez l'écraser avec les nouvelles données.

```
use Fcntl qw(:DEFAULT :flock);
sysopen(HF, "nom_fichier", O_WRONLY | O_CREAT)
    or die "impossible d'ouvrir nom_fichier : $!";
flock(HF, LOCK_EX)
    or die "impossible de verrouiller nom_fichier : $!";
truncate(HF, 0)
    or die "impossible de tronquer nom_fichier : $!";
# maintenant, écrivez dans HF
```

7. En fait, les verrous ne s'appliquent pas sur des handles de fichiers — ils s'appliquent sur les descripteurs de fichiers associés aux handles puisque le système d'exploitation ne connaît pas les handles. Cela signifie que tous vos messages `die` portant sur l'échec de l'obtention d'un verrou sur le non de fichier sont techniquement inapropriés. Mais les messages d'erreur de la forme « Je ne peux pas obtenir de verrou sur le fichier représenté par le descripteur associé au handle ouvert à l'origine avec le chemin *nom_fichier*, bien que *nom_fichier* puisse maintenant représenter un fichier totalement différent de celui que notre handle représente » ne feraient que semer la confusion dans l'esprit de l'utilisateur (sans parler du lecteur).

Si vous voulez modifier le contenu d'un fichier sur place, utilisez encore `sysopen`. Cette fois-ci, vous demandez un accès à la fois en lecture et en écriture, en créant le fichier si besoin. Une fois le fichier ouvert mais avant d'avoir fait la moindre lecture ou écriture, obtenez le verrou exclusif et gardez-le durant toute votre transaction. Il vaut souvent mieux relâcher le verrou en fermant le fichier car cela garantit que tous les tampons seront vidés avant que le verrou ne soit relâché.

Une mise à jour implique la lecture d'anciennes valeurs et l'écriture de nouvelles. Vous devez accomplir ces deux opérations sous un seul verrou exclusif, de peur qu'un autre processus ne lise les valeurs (en passe de devenir incorrectes) après (voire avant) vous, mais avant que vous n'écriviez. (Nous reverrons cette situation quand nous aurons parlé de la mémoire partagée, plus loin dans ce chapitre.)

```
use Fcntl qw(:DEFAULT :flock);

sysopen(HF, "fichier_compteur", O_RDWR | O_CREAT)
    or die "impossible d'ouvrir fichier_compteur : $!";
flock(HF, LOCK_EX)
    or die "impossible de verrouiller en écriture
            fichier_compteur : $!";
$compteur = <HF> || 0; # serait indéfini la première fois
seek(HF, 0, 0)
    or die "impossible de rembobiner fichier_compteur : $!";
print HF $compteur+1, "\n"
    or die "impossible d'écrire dans fichier_compteur : $!";

# la prochaine ligne est superflue dans ce programme
# mais est une bonne idée dans le cas général
truncate(HF, tell(HF))
    or die "impossible de tronquer fichier_compteur : $!";
close(HF)
    or die "impossible de fermer fichier_compteur : $!";
```

Vous ne pouvez pas verrouiller un fichier que vous n'avez pas encore ouvert et vous ne pouvez pas avoir un verrou unique s'appliquant à plus d'un fichier. Cependant, ce qu'il vous est possible de faire est d'utiliser un fichier totalement séparé qui se comporterait comme une sorte de sémaphore, comme un feu de signalisation, et qui fournirait un accès contrôlé à quelque chose d'autre à travers des verrous partagés et exclusifs sur ce fichier sémaphore. Cette approche possède de multiples avantages. Vous pouvez avoir un fichier verrou qui contrôle l'accès à de multiples fichiers, évitant le genre de verrou mortel (*deadlock*) survenant lorsqu'un processus essaie de verrouiller ces fichiers dans un certain ordre pendant qu'un autre processus essaie de les verrouiller dans un ordre différent. Vous pouvez utiliser un fichier sémaphore pour verrouiller un répertoire entier. Vous pouvez même contrôler l'accès à quelque chose d'autre qui ne soit même pas dans le système de fichiers, comme un objet de mémoire partagée ou une socket sur laquelle plusieurs serveurs préforkés aimeraient appeler `accept`.

Si vous avez un fichier DBM ne fournissant pas son propre mécanisme de verrouillage explicite, un fichier verrou auxiliaire est la meilleure façon de contrôler les accès concurrents par de multiples agents. Sinon, le cache interne de votre bibliothèque DBM peut se désynchroniser du fichier sur disque. Avant d'appeler `dbmopen` ou `tie`, ouvrez et

verrouillez le fichier sémaphore. Si vous ouvrez la base de données avec O_RDONLY, vous devrez utiliser LOCK_SH pour le verrou. Sinon, utilisez LOCK_EX, pour un accès exclusif lors des mises-à-jour dans la base. (Encore une fois, cela ne fonctionne que si tous les protagonistes s'entendent pour accorder de l'attention au sémaphore.)

```
use Fcntl qw(:DEFAULT :flock);
use DB_File; # seulement pour la démonstration ; toute autre db convient

$NOM_DB = "chemin/a/la/base";
$VERROU = $NOM_DB . ".verrou";

# utilisez O_RDWR si vous pensez mettre des données dans le fichier verrou
sysopen(DB_VER, $VERROU, O_RDONLY | O_CREAT)
    or die "impossible d'ouvrir $VERROU : $!";

# vous devez obtenir le verrou avant d'ouvrir la base
flock(DB_VER, LOCK_SH)
    or die "impossible de verrouiller $VERROU avec LOCK_SH : $!";

tie(%hachage, "DB_File", $NOM_DB, O_RDWR | O_CREAT)
    or die "impossible de lier $NOM_DB : $!";
```

Maintenant vous pouvez en toute sécurité faire ce que vous voulez avec le %hachage lié. Lorsque vous en avez fini avec votre base de données, assurez-vous de relâcher explicitement ces ressources et dans l'ordre inverse de celui avec lequel vous les avez acquises :

```
untie %hachage;   # vous devez fermer la base avant le fichier verrou
close DB_VER;     # maintenant le verrou peut partir en toute sécurité
```

Si la bibliothèque GNU DBM est installée, vous pouvez utiliser le verrouillage implicite du module standard GDBM_File. Sauf si le tie initial contient le drapeau GDBM_NOLOCK, la bibliothèque s'assure que seul un écrivain à la fois puisse ouvrir le fichier GDBM et que les lecteurs et les écrivains n'aient pas à la fois la base de données ouverte.

Passage de handles de fichiers

À chaque fois que vous créez un processus fils en utilisant fork, ce nouveau processus hérite de tous les handles de fichiers ouverts de son père. L'utilisation des handles de fichiers pour les communications interprocessus plus facile à illustrer en utilisant tout d'abord des fichiers. La compréhension de ce mécanisme est essentielle pour maîtriser les mécanismes plus élaborés des pipes et des sockets, décrits plus loin dans ce chapitre.

L'exemple le plus simple consiste à ouvrir un fichier et à démarrer un processus fils. Le fils utilise alors le handle de fichier déjà ouvert pour lui :

```
open(ENTREE, "< /etc/motd") or die "/etc/motd : $!";
if ($pid = fork) { waitpid($pid,0) }
else {
    defined($pid) or die "fork : $!";
    while (<ENTREE>) { print "$. : $_" }
    exit; # empêche le fils de retomber dans le code principal
}
# le handle de fichier ENTREE se trouve maintenant à EOF pour le père
```

Une fois que l'accès à un fichier a été accordé par `open`, il le reste jusqu'à la fermeture du handle de fichier ; la modification des permissions du fichier ou des privilèges d'accès n'a aucun effet sur son accessibilité. Même si le processus altère ultérieurement son ID d'utilisateur ou de groupe ou si le fichier est transféré à un nouveau propriétaire, cela n'affecte pas les handles de fichiers déjà ouverts. Les programmes tournant sous des permissions accrues (comme les programmes set-id ou les démons du système) ouvrent souvent un fichier sous leurs droits accrus et délèguent ensuite le handle de fichier à un processus fils qui n'aurait pas pu ouvrir le fichier lui-même.

Bien que cette fonctionnalité soit très pratique lorsqu'elle est utilisée intentionnellement, elle peut également poser des problèmes de sécurité si les handles de fichiers ont des fuites accidentelles d'un programme à l'autre. Pour éviter d'accorder un accès implicite à tous les handles de fichiers possibles, Perl ferme automatiquement tout handle de fichier qu'il a ouvert (y compris les pipes et les sockets) à chaque fois que vous lancez avec `exec` un nouveau programme ou que vous en exécutez un implicitement avec un `open` sur un pipe, avec `system` ou avec `qx//` (apostrophes inverses). Les handles de fichiers du système `STDIN`, `STDOUT` et `STDERR` échappent à cette règle car leur fonction première est de fournir un lien dans les communications entre programmes. Un moyen de passer un handle de fichier à un nouveau programme consiste donc à copier ce handle vers un des handles standard :

```
open(ENTREE, "< /etc/motd")   or die "/etc/motd : $!";
if ($pid = fork) { wait }
else {
    defined($pid)             or die "fork : $!";
    open(STDIN, "<&ENTREE")   or die "dup : $!";
    exec("cat", "-n")         or die "exec cat : $!";
}
```

Si vous voulez vraiment que le nouveau programme acquière l'accès à un handle de fichier autre que ces trois-là, c'est possible mais vous devrez faire une ou deux petites choses. Lorsque Perl ouvre un nouveau fichier (ou un pipe, ou une socket), il vérifie la valeur courante de la variable `$^F` (`$SYSTEM_MAX_FD`). Si le descripteur numérique du fichier utilisé par ce nouveau handle de fichier est supérieur à `$^F`, il est marqué comme l'un des descripteur qu'il faudra fermer. Sinon, Perl le laisse tranquille et les nouveaux programmes que vous lancerez avec `exec` hériteront de l'accès.

Il n'est pas toujours aussi facile de prédire quel descripteur de fichier aura votre handle récemment ouvert mais vous pouvez positionner de manière temporaire votre numéro maximal pour les descripteurs de fichiers système à une valeur immodérément grande le temps de faire le `open` :

```
# ouvre le fichier et marque ENTREE comme devant être
# conservé entre les exec
{
    local $^F = 10_000;
    open(ENTREE, "< /etc/motd") or die "/etc/motd : $!";
} # l'ancienne valeur de $^F est restaurée lorsqu'on sort de la portée
```

Maintenant, tout ce qu'il vous reste à faire est d'obtenir que le nouveau programme accorde de l'attention au numéro de descripteur du handle de fichier que vous venez d'ouvrir. La solution la plus propre (sur les systèmes implémentant ceci) consiste à pas-

ser un nom de fichier spécial équivalant à un descripteur de fichier. Si votre système possède un répertoire nommé */dev/fd* ou */proc/$$/fd*, contenant des fichiers numérotés de 0 au numéro maximal de descripteurs supportés, vous pouvez probablement utiliser cette stratégie. (Beaucoup de systèmes d'exploitation Linux ont les deux mais seule la version */proc* a tendance à être correctement peuplée. Les systèmes BSD et Solaris préfèrent */dev/fd*. Vous devrez fureter dans votre système pour voir ce qui vous convient le mieux.) Ouvrez d'abord votre handle de fichier et marquez-le comme étant l'un de ceux à laisser ouverts entre les exec comme dans le code précédent, puis faites un `fork` comme ceci :

```
if ($pid = fork) { wait }
else {
    defined($pid)                    or die "fork : $!";
    $fichier_df = "/dev/fd/" . fileno(ENTREE);
    exec("cat", "-n", $fichier_df)   or die "exec cat : $!";
}
```

Si votre système implémente l'appel système `fcntl`, vous pouvez ajuster manuellement le drapeau de handle de fichier close-on-exec. Ceci est pratique pour les fois où vous n'avez pas réalisé lorsque vous avez créé le handle de fichier que vous voudriez le partager avec vos fils.

```
use Fcntl qw/F_SETFD/;

fcntl(ENTREE, F_SETFD, 0)
    or die "Impossible de mettre à zéro le flag close-on-exec
            pour ENTREE : $!\n";
```

Vous pouvez également forcer un handle de fichier à se fermer :

```
fcntl(ENTREE, F_SETFD, 1)
    or die "Impossible de positionner le flag close-on-exec
            pour ENTREE : $!\n";
```

Vous pouvez aussi demander le statut actuel :

```
use Fcntl qw/F_SETFD F_GETFD/;

printf("ENTREE sera %s entre les exec\n",
    fcntl(ENTREE, F_GETFD, 1) ? "fermé" : "laissé ouvert");
```

Si votre système n'implémente pas les descripteurs de fichiers nommés dans le système de fichiers et que vous voulez passer un handle de fichier autre que `STDIN`, `STDOUT` ou `STDERR`, vous pouvez toujours le faire, mais vous devrez procéder à quelques arrangements spéciaux dans votre programme. Les stratégies courantes pour ceci consistent à passer le numéro de descripteur par le biais d'une variable d'environnement ou d'une option de la ligne de commande.

Si le programme exécuté est en Perl, vous pouvez utiliser `open` pour convertir un descripteur de fichier en un handle de fichier. Au lieu de spécifier un nom de fichier, utiliser `"&="`, suivi du numéro de descripteur.

```
if (defined($ENV{no_df_entree}) && $ENV{no_df_entree} =~ /^d$/) {
    open(ENTREE, "<&=$ENV{no_df_entree}")
        or die "fdopen $ENV{no_df_entree} en entrée impossible : $!";
}
```

Cela devient encore plus facile que cela si vous allez lancer un sous-programme ou un programme Perl qui attend un nom de fichier comme argument. Vous pouvez utiliser la fonctionnalité d'ouverture de descripteur de la fonction ordinaire `open` de Perl (et non `sysopen` ou la forme à trois argument d'`open`) pour que cela se fasse automatiquement. Imaginez que vous ayez un simple programme Perl comme ceci :

```
#!/usr/bin/perl -p
# nl - numéro de lignes en entrée
printf "%6d ", $.;
```

En supposant que vous vous êtes arrangé pour que le handle `ENTREE` reste ouvert entre les `exec`, vous pouvez appeler ce programme de cette manière :

```
$df_spec = '<&=' . fileno(ENTREE);
system("nl", $df_spec);
```

ou pour capturer la sortie :

```
@lignes = `nl '$df_spec'`;  # les apostrophes empêchent le
                            # shell d'interpoler df_spec
```

Que vous fassiez ou non un `exec` d'un autre programme, si vous utilisez les descripteurs de fichiers hérités à travers un `fork`, il y a un petit pépin. Au contraire des variables copiées à travers un `fork`, qui deviennent en fait des copies dupliquées mais indépendantes, les descripteurs de fichiers *sont* vraiment les mêmes dans les deux processus. Si l'un des processus lit des données depuis le handle, le pointeur de `seek` (représentant la position dans le fichier) avance également dans l'autre processus et ces données ne sont plus accessibles par les deux processus. S'ils lisent chacun à leur tour, ils joueront ensemble à saute-mouton dans le fichier. Intuitivement, cela reste sensé pour les handles attachés à des périphériques séries, des pipes ou des sockets, puisque ceux-ci ont tendance à être des périphériques en lecture seule avec des données éphémères. Mais ce comportement peut vous surprendre avec des fichiers sur disque. Si c'est un problème, rouvrez tout fichier ayant besoin de traçages séparés après le `fork`.

L'opérateur `fork` est un concept tiré d'Unix, ce qui signifie qu'il peut ne pas être correctement implémenté sur toutes les plates-formes non-Unix/non-POSIX. Notamment, `fork` ne fonctionne sur les systèmes Microsoft que si vous lancer Perl 5.6 (ou une version supérieure) sur Windows 98 (ou une version ultérieure). Bien que `fork` soit implémenté sur ces systèmes *via* de multiples flux d'exécution concurrents dans le même programme, il ne s'agit pas des types de tâches (*threads*) où toutes les données sont partagées par défaut ; ici, seuls les descripteurs de fichiers le sont. Voir également le chapitre 17, *Threads*.

Pipes

Un *pipe* est un canal d'entrées/sorties unidirectionnel pouvant transférer un flux d'octets d'un processus à un autre. Il existe deux variétés de pipes : les pipes nommés et les pipes anonymes. Il se peut que vous connaissiez mieux les pipes anonymes, dont nous commencerons donc par parler.

Pipes anonymes

La fonction open de Perl ouvre un pipe plutôt qu'un fichier lorsque vous ajouter un symbole de pipe au début ou à la fin du deuxième argument d'open. Cela change le reste de l'argument en une commande, qui sera interprétée en tant que processus (ou en tant qu'ensemble de processus) depuis lequel vous voulez recevoir ou vers lequel vous voulez envoyer un flux de données *via* un pipe. Voici comment démarrer un processus fils auquel vous avez l'intention d'écrire :

```
open IMPRESSION, "| cat -v | lpr -h 2>/dev/null"
    or die "fork impossible : $!";
local $SIG{PIPE} = sub { die "pipe d'impression cassé" };
print IMPRESSION "trucs\n";
close IMPRESSION or die "mauvaise impression : $! $?";
```

Cet exemple démarre en fait deux processus, nous écrivons directement dans le premier (effectuant *cat*). Le second processus (effectuant *lpr*) reçoit ensuite la sortie du premier. Dans la programmation shell, on appelle fréquemment ceci un *pipeline*. Un pipeline peut contenir autant de processus à la suite que vous voulez, pourvu que ceux se trouvant au milieu sachent se comporter comme des *filtres* ; c'est-à-dire qu'ils sachent lire depuis l'entrée standard et écrire dans la sortie standard.

Perl utilise le shell par défaut de votre système (*/bin/sh* sur Unix) à chaque fois qu'une commande de pipe comprend des caractères spéciaux que le shell prend en compte. Si vous ne faites que démarrer une commande et si vous n'avez pas besoin d'utiliser un shell — ou si vous ne voulez pas —, vous pouvez plutôt employer la forme à plusieurs arguments d'un open sur un pipe :

```
open IMPRESSION, "|-", "lpr", "-h" # exige Perl 5.6.1
    or die "impossible de lancer lpr : $!";
```

Si vous rouvrez la sortie standard de votre programme en tant que pipe vers un autre programme, tout ce que vous afficherez ultérieurement avec print vers STDOUT deviendra l'entrée standard du nouveau programme. Ainsi, pour envoyer à un pager la sortie de votre programme,[8] vous devez utiliser :

```
if (-t STDOUT) {            # seulement si stdout est un terminal
    my $pager = $ENV{PAGER} || 'more';
    open(STDOUT, "| $pager")   or die "fork du pager impossible : $!";
}
END {
    close(STDOUT)              or die "impossible de fermer STDOUT : $!"
}
```

Lorsque vous écrivez dans un handle de fichier connecté à un pipe, fermez toujours ce handle explicitement avec close lorsque vous en avez fini avec lui. De cette manière, votre programme principal ne se terminera pas avant sa progéniture.

Voici comment démarrer un processus fils dans lequel vous avez l'intention de lire :

```
open STATUT, "netstat -an 2>/dev/null |"
```

8. C'est-à-dire, permettre de la consulter un écran par écran et non déclencher des bips aléatoires.

```
    or die "fork impossible : $!";
while (<STATUT>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUT or die "mauvais netstat: $! $?";
```

Vous pouvez ouvrir un pipeline sur plusieurs niveaux en entrée exactement comme vous le pouvez en sortie. Et comme précédemment, vous pouvez éviter le shell en utilisant une autre forme d'`open` :

```
open STATUT, "-|", "netstat", "-an" # exige Perl 5.6.1
    or die "impossible de lancer netstat : $!";
```

Mais dans ce cas, vous n'aurez pas de redirection des entrées/sorties, ni d'expansion de jokers (*wilcard*), ni de pipes sur plusieurs niveaux, puisque Perl se base sur votre shell pour ces fonctionnalités.

Vous pourriez avoir remarqué que vous pouvez utiliser des apostrophes inverses pour obtenir le même effet qu'en ouvrant un pipe en lecture :

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "mauvais netstat" if $?;
```

Si les apostrophes sont extrêmement commodes, le programme lit tout en mémoire en une seule fois, il est donc souvent plus efficace d'ouvrir votre propre handle de fichier sur un pipe et de traiter le fichier ligne par ligne ou enregistrement par enregistrement. Cela vous donne un contrôle plus fin sur l'ensemble de l'opération, en vous laissant tuer le processus fils plus tôt si ça vous plaît. Vous pouvez également être plus efficace en traitant l'entrée au fur et à mesure, puisque les ordinateurs savent intercaler diverses opérations lorsque deux processus ou plus tournent en même temps. (Même sur une machine avec une seule CPU, les opérations d'entrées/sorties peuvent survenir lorsque la CPU fait autre chose.)

Comme vous lancez deux processus ou plus en concurrence, une catastrophe peut frapper le processus fils à n'importe quel moment entre l'ouverture avec `open` et la fermeture avec `close`. Cela signifie que le père doit vérifier la valeur de retour des deux fonctions, `open` et `close`. La seule vérification d'`open` n'est pas suffisante, puisqu'elle vous dirait seulement si la commande a été lancée avec succès ou non. (Elle ne peut vous dire ceci que dans les versions récentes de Perl et seulement si la commande est exécutée directement par le fils et non *via* le shell.) Le fils rapporte à son père les catastrophes qui interviennent après cela avec un statut de fin non nul. Lorsque la fonction `close` voit cela, elle sait renvoyer une valeur fausse, indiquant que le véritable statut devrait être lu à partir de la variable `$?` (`$CHILD_ERROR`). Vérifier la valeur renvoyée par `close` est donc aussi important que celle renvoyée par `open`. Si vous écrivez dans un pipe, vous devriez également vous préparer à gérer le signal `PIPE`, qui vous est envoyé si le processus à l'autre extrémité meurt avant que vous ayez fini de lui transmettre des données.

Soliloque

Une autre approche de la communication interprocessus consiste à faire soliloquer le programme, pour ainsi dire. En fait, votre processus discute avec une copie de lui-même par le biais de pipes. Cela fonctionne largement comme l'`open` sur un pipe dont nous

avons parlé au paragraphe précédent, hormis le fait que le processus fils continue à exécuter votre script plutôt qu'une autre commande.

Pour indiquer ceci à la fonction `open`, utilisez une pseudo-commande composée d'un signe moins. Le second argument d'`open` ressemble donc à « `-|` » ou à « `|-` », selon que vous voulez transmettre des données depuis ou vers vous-même. Comme avec une commande `fork` ordinaire, la fonction `open` renvoie l'ID du processus fils dans le processus père, mais `0` dans le processus fils. Une autre asymétrie est que le handle de fichier cité dans l'`open` n'est utilisé que dans le processus père. L'extrémité du pipe pour le fils est rattachée soit à `STDIN`, soit à `STDOUT`, selon le cas. C'est-à-dire que si vous ouvrez un pipe *vers* le signe moins, vous pouvez écrire des données dans le handle de fichier que vous avez ouvert et votre fils les lira dans `STDIN` :

```
if (open(VERS, "|-")) {
    print VERS $depuis_pere;
}
else {
    $vers_fils = <STDIN>;
    exit;
}
```

Si vous ouvrez un pipe *depuis* le signe moins, vous pouvez lire dans le handle de fichier ouvert ce que le fils écrit dans `STDOUT` :

```
if (open(DEPUIS, "-|"))
    $vers_pere = <DEPUIS>;
}
else {
    print STDOUT $depuis_fils;
    exit;
}
```

Une application courante de cette construction est de court-circuiter le shell lorsque vous voulez ouvrir un pipe depuis une commande. Vous pourriez vouloir faire ceci car vous ne voulez pas que le shell interprète les métacaractères éventuels dans les noms de fichiers que vous essayer de passer à la commande. Si vous tournez sous la version 5.6.1 de Perl, ou une version supérieure, vous pouvez utiliser la forme à plusieurs arguments d'`open` pour obtenir le même résultat.

Une autre utilisation d'un `open` sur un `fork` consiste à ouvrir un fichier ou une commande en toute sécurité alors que vous tournez sous un certain UID ou GID. Le fils que vous lancez avec `fork` abandonne tous les droits d'accès spéciaux, puis ouvre en toute sécurité le fichier ou la commande et se comporte comme un intermédiaire, en passant les données entre son père plus puissant et le fichier ou la commande qu'il a ouvert. Vous pouvez trouver des exemples dans la section *Accéder aux commandes et aux fichiers avec des privilèges restreints*, au chapitre 23.

Une utilisation pleine de créativité d'un `open` sur un `fork` est le filtrage de votre propre sortie. Certains algorithmes sont beaucoup plus faciles à implémenter en deux passes séparées plutôt qu'en une seule. Voici un exemple simple dans lequel nous émulons le programme *tee*(1) d'Unix en envoyant notre sortie ordinaire vers un pipe. L'agent à l'autre extrémité du pipe (l'un de vos propres sous-programmes) distribue notre sortie vers tous les fichiers spécifiés :

```
tee("/tmp/truc", "/tmp/machin", "/tmp/bidule");

while (<>) {
    print "$ARGV à la ligne $. => $_";
}
close(STDOUT) or die "impossible de fermer STDOUT : $!";

sub tee {
    my @sortie = @_;
    my @handles = ();
    for my $chemin (@sortie) {
        my $hf; # open remplira ceci
        unless (open ($hf, ">", $chemin)) {
            warn "impossible d'écrire dans $chemin : $!";
            next;
        }
        push @handles, $hf;
    }
    # rouvre STDOUT dans le père et renvoie
    return if my $pid = open(STDOUT, "|-");
    die "fork impossible : $!" unless defined $pid;

    # traite STDIN dans le fils
    while (<STDIN>) {
        for my $hf (@handles) {
            print $hf $_ or die "échec de la sortie de tee : $!";
        }
    }
    for my $hf (@handles) {
        close($hf) or die "échec de la fermeture de tee : $!";
    }
    exit; # empêche le fils de renvoyer au programme principal !
}
```

Cette technique peut être appliquée en boucle pour mettre autant de filtres que vous le désirez sur votre sortie. Continuez simplement d'appeler les fonctions qui font un `fork-open` sur `STDOUT`, ayez le fils qui lit depuis son père (qu'il voit en tant que `STDIN`) et passez la sortie triturée à la prochaine fonction de la chaîne.

Une autre application intéressante du soliloque avec `fork-open` consiste à capturer la sortie d'une fonction ayant de mauvaises manières en envoyant tous ses résultats éclabousser `STDOUT`. Imaginez Perl qui n'aurait qu'une fonction `printf`, sans fonction `sprintf`. Ce dont vous auriez besoin est d'une fonction marchant comme les apostrophes inverses mais avec des fonctions Perl au lieu de commandes externes :

```
fonction_mechante("arg");                       # zut, ça nous a échappé !
$chaine = sub_fork(\&fonction_mechante, "arg"); # capturez-la dans une
                                                # chaîne
@lignes = sub_fork(\&fonction_mechante, "arg"); # dans des lignes séparées

sub sub_fork {
```

```
    my $pid_fils = open my $self, "-|";
    defined $pid_fils       or die "fork impossible : $!";
    shift->(@_), exit       unless $pid_fils;
    local $/                unless wantarray;
    return <$self>;         # ferme à la sortie de la portée
}
```

Nous ne prétendons pas que ceci est efficace ; un handle de fichier lié avec `tie` irait certainement bien plus vite. Mais c'est beaucoup plus facile à coder si vous êtes plus pressé que votre ordinateur.

Communication bidirectionnelle

Bien que l'emploi d'`open`, pour se connecter à une autre commande *via* un pipe, fonctionne raisonnablement bien pour les communications unidirectionnelles, *quid* des communications bidirectionnelles ? L'approche la plus évidente ne fonctionne pas :

```
open(PROG_EN_LECTURE_ECRITURE, "| un programme |") # MAUVAIS !
```

et si vous oubliez d'activer les avertissements, le message de diagnostic passera à la trappe :

```
Can't do bidirectional pipe at mon_prog line 3.
```

La fonction `open` ne l'autorise pas car cette approche est tout à fait encline à causer des verrous mortels (*deadlocks*) si vous n'êtes pas très attentif. Mais si vous êtes déterminé,vous pouvez utiliser le module standard de bibliothèque `IPC::Open2` pour rattacher deux pipes aux `STDIN` et `STDOUT` d'un sous-processus. Il existe également `IPC::Open3` pour les entrées/sorties tridirectionnelles (vous permettant également de capturer le `STDERR` de votre fils fils) mais ce dernier exige soit une boucle `select` embarrassée, soit le module IO::Select, plus commode. Mais vous devrez alors éviter les opérations dans le tampon d'entrée de Perl, comme `<>` (`readline`).

Voici un exemple utilisant `open2` :

```
use IPC::OPen2;
local (*Lecteur, *Ecrivain);
$pid = open2(\*Lecteur, \*Ecrivain, "bc -l");
$somme = 2;
for (1 .. 5) {
    print Ecrivain "$somme * $somme\n";
    chomp($somme = <Lecteur>);
}
close Ecrivain;
close Lecteur;
waitpid($pid, 0);
print "La somme vaut $somme\n";
```

Vous pouvez également autovivifier des handles de fichiers lexicaux :

```
my ($hf_lecture, $hf_ecriture);
$pid = open2($hf_lecture, $hf_hf_ecriture, "cat -u -n");
```

Généralement le problème avec ceci est que le vidage de tampon standard des entrées/sorties va vraiment gâcher votre journée. Même si votre handle de fichier de sortie est

automatiquement vidé (la bibliothèque le fait pour vous) de sorte que le processus à l'autre extrémité recevra vos données de manière opportune, vous ne pouvez généralement rien faire pour l'obliger à vous retourner la politesse. Dans ce cas particulier, nous avons de la chance : `bc` s'attend à manipuler un pipe et sait vider (*flush*) chaque ligne de sortie. Mais peu de commandes sont conçues de la sorte, cela fonctionne donc rarement à moins d'écrire vous-même le programme à l'autre extrémité du pipe bidirectionnel. Même les simples programmes apparemment interactifs, comme *ftp*, échouent ici car ils ne videront pas leur tampon ligne par ligne dans un pipe. Il ne le feront que sur un périphérique tty.

Les modules de CPAN `IO::Pty` et `Expect` peuvent vous venir en aide car ils fournissent un vrai tty (en fait, un vrai pseudo-tty mais se comportant comme un vrai). Cela vous donne un vidage de tampon ligne par ligne dans l'autre processus, sans modifier son programme.

Si vous éclatez votre programme en plusieurs processus et que vous voulez qu'ils aient tous ensemble une conversation allant dans les deux sens, vous ne pouvez pas utiliser les interfaces de haut niveau de Perl, car ces dernières sont toutes unidirectionnelles. Vous devrez utilisez deux appels de fonction `pipe`, de bas niveau, chacun manipulant une direction dans la conversation :

```
pipe(DEPUIS_PERE, VERS_FILS)     or die "pipe : $!";
pipe(DEPUIS_FILS, VERS_PERE)     or die "pipe : $!";
select((select(VERS_FILS), $| = 1))[0]);   # vidage automatique
select((select(VERS_PERE), $| = 1))[0]);   # vidage automatique

if ($pid = fork) {
    close DEPUIS_PERE; close VERS_PERE;
    print VERS_FILS "Pid $$ du père envoie ceci\n";
    chomp($ligne = <DEPUIS_FILS>);
    print "Pid $$ du père vient de lire ceci : ‘$ligne’\n";
    close DEPUIS_FILS; close VERS>FILS;
    waitpid($pid,0);
} else {
    die "fork impossible : $!" unless defined $pid;
    close DEPUIS_FILS; close VERS_FILS;
    chomp($ligne = <DEPUIS_PERE>);
    print "Pid $$ du fils vient de lire ceci : ‘$ligne’\n";
    print VERS_PERE "Pid $$ du fils envoie ceci\n";
    close DEPUIS_PERE; close VERS_PERE;
    exit;
}
```

Sur la plupart des systèmes Unix, vous n'avez en fait pas à faire deux appels `pipe` séparés pour accomplir une communication bidirectionnelle complète (*full duplex*) entre un père et son fils. L'appel `socketpair` fourni des connexions bidirectionnelles entre des processus associés sur la même machine. Ainsi, plutôt que deux `pipe`, vous n'avez besoin que d'un `socketpair` :

```
use Socket;
socketpair(Fils, Pere, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";
```

```
# ou laissez Perl choisir les handles de fichiers pour vous
my ($hf_fils, $hf_pere);
socketpair($hf_fils, $hf_pere, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";
```

Après le `fork`, le père ferme le handle `Pere`, puis lit et écrit *via* le handle `Fils`. Pendant ce temps, le fils ferme le handle `Fils`, puis lit et écrit *via* le handle `Pere`.

Pipes nommés

Un pipe nommé (souvent appelé FIFO) est un mécanisme pour mettre en place un conversation entre des processus sans rapport les uns avec les autres, sur la même machine. Les noms en question existent dans le système de fichiers, ce qui est une manière amusante de dire que vous pouvez mettre un fichier spécial dans le système de fichiers, soustendant un processus plutôt qu'un disque.[9]

Un FIFO est pratique lorsque vous voulez connecter un processus à un autre qui n'a aucun rapport. Lorsque vous ouvrez un FIFO, votre processus bloquera jusqu'à ce qu'il y ait un processus à l'autre extrémité. Ainsi, si un lecteur ouvre en premier un FIFO, il bloquera jusqu'à ce que l'écrivain apparaisse — et vice-versa.

Pour créer un pipe nommé, utilisez la fonction `mkfifo` de POSIX — enfin, si vous êtes sur un système POSIX. Sur les systèmes Microsoft, vous pourrez à la place regarder le module `Win32::Pipe`, qui, malgré les apparences, crée des pipes nommés. (Les utilisateurs de Win32 créent des pipes anonymes en employant `pipe`, tout comme le reste d'entre nous.)

Par exemple, mettons que vous aimeriez que votre fichier *.signature* produise une réponse différente à chaque fois qu'il est lu. Vous n'avez qu'à le mettre en pipe nommé avec un programme Perl à l'autre extrémité crachant des aphorismes au hasard. Maintenant, à chaque fois qu'un programme quelconque (comme en logiciel envoyant des mails, un lecteur de forums de discussion, un programme *finger*, et ainsi de suite) essayera de lire dans ce fichier, ce programme se connectera au vôtre et lira une signature dynamique.

Dans l'exemple suivant, nous utilisons l'opérateur de test de fichier `-p`, que l'on rencontre rarement, pour déterminer si quelqu'un (ou quelque chose) a supprimé accidentellement notre FIFO.[10] Si c'est le cas, il n'y a aucune raison pour essayer de l'ouvrir, nous traiterons donc ceci comme une demande de terminaison. Si nous utilisions une simple fonction open avec un mode valant « `+>` `$chemin_f` », il y aurait une minuscule situation de concurrence (*race condition*) qui risquerait de créer la signature en tant que simple fichier s'il disparaissait entre le test `-p` et l'ouverture. Nous ne pourrions pas non plus utiliser un mode valant « `+<` `$chemin_f` » car l'ouverture d'une FIFO en lecture-écriture est non-bloquante (ceci n'est vrai que pour les FIFO). En utilisant `sysopen` et en omettant le drapeau `O_CREAT`, nous évitons ce problème en ne créant jamais de fichier

9. Vous pouvez utiliser la même chose avec des sockets dans le domaine Unix mais vous ne pouvez pas utiliser `open` sur ces derniers.

10. Une autre utilisation consiste à voir si un handle de fichier est connecté à un pipe, nommé ou anonyme, comme dans `-p STDIN`.

par accident.

```
use Fcntl;                # pour sysopen
chdir;                    # au répertoire « maison »
$chemin_f = '.signature';
$ENV{PATH} .= ":/usr/games";

unless (-p $chemin_f) {   # pas un pipe
    if (-e _) {           # mais quelque chose d'autre
        die "$0 : n'écrasera pas .signature\n";
    } else {
        require POSIX;
        POSIX::mkfifo($chemin_f, 0666)
            or die "mknod $chemin_f impossible: $!";
        warn "$0 : a créé $chemin_f en tant que pipe nommé\n";
    }
}
while (1) {
    # quitte si le fichier signature est supprimé manuellement
    die "Le fichier pipe a disparu" unless -p $chemin_f;
    # la prochaine ligne bloque s'il n'y a pas de lecteur
    sysopen(FIFO, $chemin_f, O_WRONLY)
        or die "Impossible d'écrire dans $chemin_f: $!";
    print FIFO "Gérard Lambert (lambert\@machine.org)\n", `fortune -s`;
    close FIFO;
    select(undef, undef, undef, 0.2); # dort 1/5ème de seconde
}
```

La courte sieste après la fermeture du FIFO est nécessaire pour laisser au lecteur une chance de lire ce qui a été écrit. Si nous avions recommencé la boucle immédiatement et rouvert le FIFO avant que notre lecteur ait fini de lire les données que nous venions d'envoyer, nous n'aurions pas vu de fin de fichier (*end-of-file*) car il y aurait encore un écrivain. Nous aurions bouclé tous les deux encore et encore jusqu'à ce qu'à une itération donnée, l'écrivain arrive un peu après et que le lecteur voie cette fin de fichier insaisissable. (Et nous nous sommes inquiétés à propos des situations de concurrence ?)

IPC System V

Tout le monde déteste les IPC System V. Elles sont plus lentes que des cartes perforées, se taillent de petits espaces de noms insidieux sans aucun rapport avec le système de fichiers, utilisent des numéros hostiles à l'entendement humain pour nommer leurs objets et perdent constamment le fil de leurs propres pensées. Et de temps à autre, votre administrateur système doit se lancer dans une opération de ratissage pour rechercher et de détruire ces objets perdus d'IPC SysV avec *ipcs*(1) et les tuer avec *ipcrm*(1), avant que le système n'ait plus de mémoire, si tout va bien.

Malgré tous ces maux, les antiques IPC SysV ont toujours des applications valides. Les trois sortes d'objets d'IPC sont la mémoire partagée, les sémaphores et les messages. Pour la transmission de messages, on préférera de nos jours le mécanisme de sockets qui est en outre beaucoup plus portable. Pour les utilisations simples des sémaphores, on a

plutôt tendance à utiliser le système de fichiers. Quant à la mémoire partagée — en fait, vous avez maintenant un problème. Si vous l'avez, l'appel système *mmap*(2), plus moderne, remplit ce contrat[11] mais la qualité de l'implémentation varie d'un système à l'autre. Il requiert en outre un peu d'attention pour que Perl évite de réallouer vos chaînes depuis l'endroit où *mmap*(2) les a mises. Mais lorsque les programmeurs envisagent d'utiliser *mmap*(2), ils entendent marmonner de manière incohérente les gourous locaux sur le fait que *mmap* souffre de problèmes épineux avec la cohérence de cache s'il manque quelque chose appelé un « cache de tampon unifié » (*unified buffer cache*) — à moins que ce ne soit une « tâche d'unité de canton » (*flycatcher unibus*) — et, réfléchissant sur le fait que l'on sait ce qu'on perd mais qu'on ne sait pas ce qu'on trouve, ils retournent vite aux IPC SysV qu'ils connaissent et détestent pour tous leurs besoins en mémoire partagée.

Voici un petit programme démontrant l'accès contrôlé à un tampon en mémoire partagé par une nichée de processus apparentés. Les objets d'IPC SysV peuvent également être partagés entre des processus *sans rapport* les uns avec les autres, sur le même ordinateur, mais vous devez alors imaginer comment ils vont se trouver les uns les autres. Nous créerons un sémaphore par entité, qui servira d'intermédiaire pour un accès sécurisé.[12]

À chaque fois que vous voulez récupérer ou mettre une nouvelle valeur en mémoire partagée, vous devez passer d'abord par le sémaphore. Ceci peut devenir un peu pénible, nous enroberons donc l'accès dans une classe d'objets. Le module `IPC::Shareable` va encore plus loin en enrobant sa classe d'objets dans une interface `tie`.

Ce programme tourne jusqu'à ce que vous l'interrompiez avec un Ctrl-C ou un équivalent :

```
#!/usr/bin/perl -w
use v5.6.0;         # ou une version supérieure
use strict;
use sigtrap qw(die INT TERM HUP QUIT);
my $PROGENITURE = shift(@ARGV) || 3;
eval { main() }; # voir DESTROY ci-dessous pour comprendre pourquoi
die if $@ && $@ !~ /^SIG reçu/;
print "\nFini.\n";
exit;

sub main {
    my $mem = ShMem->allouer("Création Originale à " . localtime);
    my(@rejetons, $fils);
    $SIG{CHLD} = 'IGNORE';
    for (my $embryon = $PROGENITURE; $embryon > 0; $embryon--) {
```

11. Il y a même un module `Mmap` sur CPAN.

12. Il serait plus réaliste de créer une paires de sémaphores pour chaque morceau de mémoire partagée : un pour la lecture et l'autre pour l'écriture. Et, en fait, c'est ce que fait le module de CPAN `IPC::Shareable`. Mais nous essayons ici de rester simples. Cependant il vaut mieux admettre qu'avec un couple de sémaphores, vous pourriez alors faire usage d'à peu près le seul bénéfice des IPC SysV : vous pourriez exécuter des opérations atomiques sur des ensembles entiers de sémaphores comme s'il s'agissait d'un seul élément, ce qui est parfois bien utile.

```
        if ($fils = fork) {
            print "$$ engendra $fils\n";
            next;
        }
        die "fork impossible : $!" unless defined $fils;
        eval {
            while (1) {
                $mem->verrouiller();
                $mem->positionner("$$ " . localtime)
                unless $mem->consulter =~ /^$$\b/o;
                $mem->deverrouiller();
            }
        };
        die if $@ && $@ !~ /^SIG reçu/;
        exit; # mort du fils
    }
    while (1) {
        print "Le tampon vaut ", $mem->recuperer, "\n";
        sleep 1;
    }
}
```

Et voici le paquetage `ShMem`, que ce programme utilise. Vous n'avez qu'à l'épingler à la fin du programme ou le mettre dans son propre fichier (avec un « `1;` » à la fin) et le charger avec `require` depuis le programme principal. (On trouve les deux modules d'IPC qu'il utilise à son tour dans la distribution standard de Perl.)

```
package ShMem;
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);
use IPC::Semaphore;
sub MAXBUF() { 2000 }

sub allouer {     # constructeur
    my $classe = shift;
    my $valeur = @_ ? shift : ”;

    my $clef = shmget(IPC_PRIVATE, MAXBUF, S_IRWXU) or die "shmget : $!";
    my $sem = IPC::Semaphore->new(IPC_PRIVATE, 1, S_IRWXU | IPC_CREAT)
                        or die "IPC::Semaphore->new : $!";
    $sem->setval(0,1)   or die "sem setval : $!";

    my $obj = bless {
        PROPRIETAIRE => $$,
        SHMKEY => $clef,
        SEMA => $sem,
    } => $classe;

    $obj->mettre($valeur);
    return $obj;
}
```

Maintenant, passons aux méthodes de collecte et de stockage. Les méthodes `recolter` et `mettre` verrouillent le tampon, contrairement aux méthodes `consulter` et `positionner`, ces dernières ne devraient donc être utilisées que lorsque l'objet est verrouillé manuellement — ce que vous devez faire lorsque vous voulez aller chercher une ancienne valeur et remettre une version modifiée, tout cela avec le même verrou. Le programme de démonstration fait ceci dans sa boucle `while (1)`. La transaction complète doit se dérouler avec le même verrou sinon le test et le positionnement ne seraient pas atomiques et pourraient exploser.

```
sub recolter {
    my $obj = shift;
    $obj->verrouiller;
    my $valeur = $obj->consulter(@_);
    $obj->deverrouiller;
    return $valeur;
}
sub consulter {
    my $obj = shift;
    shmread($obj-&>{SHMKEY}, my $tampon=”, 0, MAXBUF)
        or die "shmread : $!";
    substr($tampon, index($tampon, "\0")) = ”;
    return $tampon;
}
sub mettre {
    my $obj = shift;
    $obj->verrouiller;
    $obj->positionner(@_);
    $obj->deverrouiller;
}
sub positionner {
    my($obj,$msg) = @_;
    shmwrite($obj->{SHMKEY}, $msg, 0, MAXBUF) or die "shmwrite : $!";
}
sub verrouiller {
    my $obj = shift;
    $obj->{SEMA}->op(0,-1,0) or die "semop : $!";
}
sub deverrouiller {
    my $obj = shift;
    $obj->{SEMA}->op(0,1,0) or die "semop : $!";
}
```

Finalement, la classe a besoin d'un destructeur pour que, lorsque l'objet disparaît, nous puissions manuellement désallouer la mémoire partagée et le sémaphore stocké à l'intérieur de l'objet. Sinon, ils survivront à leur créateur et vous devrez recourir à *ipcs* et *ipcrm* (ou un administrateur système) pour vous débarrasser d'eux. C'est pourquoi nous somme passés par les enrobages élaborés dans le programme principal pour convertir les signaux en exceptions : pour que tous les destructeurs soient lancés, que tous les objets d'IPC SysV soient désalloués et que les administrateurs système ne s'occupent pas de notre cas.

```
sub DESTROY {
    my $obj = shift;
    return unless $obj->{PROPRIETAIRE} == $$;   # évite les désallocations
                                                # redondantes
    shmctl($obj->{SHMKEY}, IPC_RMID, 0) or warn "shmctl RMID : $!";
    $obj->{SEMA}->remove()               or warn "sema->remove : $!";
}
```

Sockets

Les mécanismes d'IPC dont nous avons discuté auparavant souffrent tous d'une importante restriction : ils sont conçus pour un communication entre des processus tournant sur le même ordinateur. (Même si les fichiers peuvent parfois être partagés entre les machines à travers des mécanismes comme NFS, le verrouillage échoue lamentablement sur la plupart des implémentations de NFS, ce qui annihile une grande part des plaisirs des accès concurrents.) Pour des communications génériques sur le réseau, les sockets sont la solution de choix. Bien que les sockets aient été inventées sur BSD, elles se sont rapidement répandues aux autres formes d'Unix et de nos jours, vous pouvez trouver une interface pour les sockets sur à peu près tous les systèmes d'exploitations viables ici-bas. Si vous n'avez pas de sockets sur votre machine, vous allez rencontrer d'énormes difficultés pour utiliser l'Internet.

Avec les sockets, vous pouvez communiquer à la fois avec des circuits virtuels (les *flux* ou *streams* TCP) et avec des datagrammes (les paquets UDP). Votre système vous permet peut-être d'en faire plus encore. Mais le type de programmation socket le plus courant utilise TCP sur des sockets du domaine Internet, c'est donc celui que nous détaillerons ici. De telles sockets fournissent des connexions fiables fonctionnant un peu comme deux pipes bidirectionnels qui ne seraient pas limités à une seule machine. Les deux applications reines (*killer apps*) de l'Internet, l'email et le surf sur le Web, reposent presqu'exclusivement sur des sockets TCP.

Vous utilisez aussi largement UDP sans le savoir. Chaque fois que votre machine essaie de trouver un site sur l'Internet, elle envoie des paquets UDP à votre serveur DNS pour lui demander la véritable adresse IP. Vous pourriez utiliser UDP vous-même lorsque vous voulez envoyer et recevoir des datagrammes. Les datagrammes sont plus économiques que les connexions TCP précisément parce qu'ils ne sont pas orientés connexion ; c'est-à-dire qu'ils ressemblent moins à un appel téléphonique qu'à un dépôt de lettre dans une boîte. Mais il manque également à UDP la fiabilité offerte par TCP, ce qui le rend plus approprié pour les situations où vous ne vous souciez pas si un ou deux paquets se retrouvent happés, broyés et engloutis. Ou lorsque vous savez qu'un protocole de plus haut niveau fera respecter un certain degré de redondance ou d'échec en douceur (ce que fait DNS).

D'autres choix sont disponibles mais bien moins courants. Vous pouvez utilisez des sockets du domaine Unix mais elles ne fonctionnent que pour les communications locales. Divers systèmes implémentent divers autres protocoles qui ne sont pas basés sur IP. Ceux-ci intéressent sans doute quelqu'un quelque part pour faire quelque chose mais nous devons nous restreindre quelquefois.

Les fonctions Perl s'occupant des sockets ont les mêmes noms que les appels système

correspondants en C, mais leurs arguments ont tendance à différer pour deux raisons : premièrement, les handles de fichiers de Perl fonctionnent différemment des descripteurs C ; deuxièmement, Perl connaît déjà la longueur de ses chaînes et il est donc inutile de passer cette information. Voir les détails sur chaque appel système relatif aux sockets au chapitre 29.

Avec du code Perl ancien, on devait utiliser des valeurs codées en dur pour passer des constantes aux fonctions des sockets, ce qui annihilait la portabilité. Comme la plupart des appels système, ceux relatifs aux sockets renvoient discrètement mais poliment `undef` lorsqu'ils échouent, au lieu de lever une exception. Il est donc essentiel de vérifier les valeurs renvoyées par ces fonctions, puisque si vous les jetez à la poubelle, elles n'iront pas crier cet échec sur les toits. Si jamais vous voyez du code faisant quelque chose comme positionner explicitement `$AF_INET = 2`, vous savez que vous êtes dans les problèmes jusqu'au cou. Une approche d'une supériorité incommensurable consiste à utiliser le module `Socket` ou le module encore plus sympathique `IO::Socket`, tous deux standard. Ces modules fournissent diverses constantes et des fonctions d'aide dont vous aurez besoin pour mettre en place des clients et des serveurs. Pour une réussite optimale, vos programmes de sockets devraient toujours commencer ainsi (n'oubliez pas non plus d'ajouter l'option de vérification de marquage **`-T`** sur la ligne *shebang* (commençant par `#!`) pour les serveurs) :

```
#!/usr/bin/perl -w
use strict;
use sigtrap;
use Socket; # ou IO::Socket
```

Comme nous l'avons fait remarquer ailleurs, Perl est à la merci de vos bibliothèques C pour la plupart de ses fonctionnalités concernant le système, et tous les systèmes n'implémentent pas toutes les sortes de sockets. Il est probablement plus sûr de s'en tenir avec les opérations sur les sockets normales TCP et UDP. Par exemple, si vous voulez que votre code ait une chance d'être portable vers des systèmes auxquels vous n'avez pas pensé, ne vous attendez pas à ce qu'ils implémentent un protocole fiable avec des paquets séquencés. Ne vous attendez pas non plus à passer des descripteurs de fichiers ouverts entre des processus n'ayant aucun rapport les uns avec les autres sur des sockets du domaine Unix. (Si, si, vous pouvez vraiment faire cela sur de nombreuses machines Unix — voir *recvmsg*(2) dans votre page de manuel locale.)

Si vous voulez juste utiliser un service standard de l'Internet comme le mail, les forums de discussion, le service de noms de domaine, FTP, Telnet, le Web et ainsi de suite, alors plutôt que de repartir de zéro, essayez d'utiliser les modules existants de CPAN. Les modules tout faits, conçus pour cela, comprennent : `Net::SMTP` (ou `Mail::Mailer`), `Net:NNTP`, `Net::DNS`, `Net::FTP`, `Net::Telnet` et les divers modules relatifs à HTTP. Les familles de modules `libnet` et `libwww` comprennent toutes deux de nombreux modules individuels pour la communication en réseau. Les zones de modules sur CPAN qui peuvent vous intéresser sont dans la section 5 sur la Communication en réseau et dans la section 15, sur les Utilitaires de serveurs et de démons.

Dans les paragraphes qui suivent, nous présentons plusieurs exemples de clients et de serveurs sans grande quantité d'informations sur chaque fonction utilisée car ce serait en grande partie une recopie pure et simple de ce que nous avons déjà écrit au chapitre 29, *Fonctions*.

Clients en réseau

Utilisez les sockets du domaine Internet lorsque vous voulez une communication client/serveur fiable entre des machines potentiellement différentes.

Pour créer un client TCP se connectant à un serveur quelque part, il est d'ordinaire plus facile d'utiliser le module standard `IO::Socket::INET` :

```
use IO::Socket::INET;

$socket = IO::Socket::INET->new(PeerAddr => $hote_distant,
                                PeerPort => $port_distant,
                                Proto    => "tcp",
                                Type     => SOCK_STREAM)
    or die "N'a pas pu se connecter à $hote_distant:$port_distant : $!\n";
# envoie quelque chose sur la socket,
print $socket "Pourquoi ne m'appelles-tu plus jamais ?\n";

# lit la réponse distante,
$reponse = <$socket>;

# et termine la connexion lorsque nous avons fini
close($socket);
```

Un raccourci pour l'appel est suffisant lorsque vous n'avez que l'hôte et le port auxquels se connecter et que vous voulez utiliser les valeurs par défaut pour les autres champs :

```
$socket = IO::Socket::INET->new("www.google.org:80")
    or die "N'a pas pu se connecter au port 80 de google: $!";
```

Pour se connecter en utilisant le module `Socket` :

```
use Socket;

# crée une socket
socket(Serveur, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

# construit l'adresse de la machine distante
$adr_internet = inet_aton($hote_distant)
    or die "N'a pas pu convertir $hote_distant en adresse Internet : $!\n";
$adr_p = sockaddr_in($port_distant, $adr_internet);

# se connecte
connect(Serveur, $adr_p)
    or die "N'a pas pu se connecter à $hote_distant:$port_distant : $!\n";

select((select(Serveur), $| = 1)[0]);   # active la mise en tampon
                                        # des commandes

# envoie quelque chose sur la socket
print Serveur "Pourquoi ne m'appelles-tu plus jamais ?\n";

# lit la réponse distante
$reponse = <Serveur>;
```

```
# et termine la connexion lorsque nous avons fini
close(Serveur);
```

Si vous ne voulez fermer que votre côté de la connexion, pour que l'extrémité distante reste reçoive une fin de fichier mais que vous puissiez continuer à lire les données venant du serveur, utilisez l'appel système `syscall` pour une demi-fermeture :

```
# plus d'écriture vers le serveur
shutdown(Serveur, 1);     # constante Socket::SHUT_WR dans la v5.6
```

Serveurs en réseau

Voici le serveur correspondant qui va avec. Il est assez simple grâce à la classe standard `IO::Socket::INET` :

```
use IO::Socket::INET;

$serveur = IO::Socket::INET->new(LocalPort => $port_serveur,
                                 Type      => SOCK_STREAM,
                                 Reuse     => 1,
                                 Listen    => 10 )     # ou SOMAXCONN
    or die "N'a pas pu être un serveur TCP sur le port $port_serveur : $!\n";

while ($client = $serveur->accept()) {
    # $client est la nouvelle connexion
}

close($serveur);
```

Vous pouvez également écrire cela en utilisant le module `Socket`, de plus bas niveau :

```
use Socket;

# crée la socket
socket(Serveur, PF_INET, SOCK_STREAM, getprotobyname('tcp'));

# pour pouvoir redémarrer rapidement notre serveur
setsockopt(Serveur, SOL_SOCKET, SO_REUSEADDR, 1);

# construit l'adresse de la socket
$mon_adr = sockaddr_in($port_serveur, INADDR_ANY);
bind(Serveur, $mon_adr)
    or die "N'a pas pu s'attacher au port $port_serveur : $!\n";

# établit une file pour connexions entrantes
listen(Serveur, SOMAXCONN)
    or die "N'a pas pu écouter sur le port $port_serveur : $!\n";

# accepte et traite les connexions
while (accept(Client, Serveur)) {
    # Fait quelque chose avec la nouvelle connexion Client
}

close(Server);
```

Le client n'a pas besoin de s'attacher avec `bind` à une adresse particulière, par contre c'est le cas pour le serveur. Nous avons spécifié son adresse en tant que `INADDR_ANY`, ce qui signifie que les clients peuvent se connecter depuis n'importe quelle interface réseau disponible. Si vous voulez vous placer devant une interface particulière (comme le côté externe d'une passerelle ou d'une machine pare-feux (*firewall*)), utilisez plutôt l'adresse réelle de cette interface. (Les clients peuvent également faire cela mais en ont rarement besoin.)

Si vous voulez savoir quelle machine s'est connectée avec vous, appelez `getpeername` sur la connexion du client. Ceci renvoie une adresse IP, que vous devrez convertir vous-même vers un nom (si vous le pouvez) :

```
use Socket;
$autre_extremite = getpeername(Client)
    or die "N'a pas pu identifier l'autre extrémité : $!\n";
($port, $adr_i) = unpack_sockaddr_in($autre_extremite);
$ip_reelle = inet_ntoa($adr_i);
$nom_hote_pretendu = gethostbyaddr($adr_i, AF_INET);
```

Ceci être usurpé (*spoofable*) de manière triviale car le propriétaire de cette adresse IP peut initialiser ses tables d'inversion (*reverse tables*) pour qu'elles disent ce qu'il veut. Une petite mesure pour s'assurer de la validité de l'adresse IP consiste à faire la conversion inverse :

```
@recherche_noms = gethostbyname($nom_hote_pretendu)
    or die "N'a pas pu faire la conversion inverse de $nom_hote_pretendu :
$!\n";
@ips_resolues = map{inet_ntoa($_)} $recherche_noms[ 4 .. $#recherche_noms ];
$usurpation_potentielle = !gre             e eq $_ } @ips_resolues;
```

Une fois qu'un client s'est connecté à votre serveur, ce dernier peut faire des entrées/sorties à la fois depuis et vers ce handle client. Mais pendant que le serveur est ainsi occupé, il ne peut plus servir de requêtes entrantes provenant d'autres clients. Pour éviter de rester bloqué sur un seul client à la fois, de nombreux serveurs se clônent, avec `fork`, pour s'occuper de chaque connexion entrante. (D'autres anticipent les `fork` ou multiplexent les entrées/sorties entre plusieurs clients en utilisant l'appel système `select`.)

```
REQUETE:
while (accept(Client, Serveur)) {
    if ($pid_fils = fork) {
        close Client;      # le père ferme le handle inutilisé
        next REQUETE;
    }
    defined($pid_fils)    or die "fork impossible : $!" ;

    close Serveur;         # le fils ferme le handle inutilisé

    select(Client);        # nouveau handle par défaut pour les affichages
    $| = 1;                # vidage de tampon automatique

    # le code par connexion des fils fait ses entrées/sorties avec le
    # handle Client
    $entree = <Client>;
```

```
        print Client "affichage\n";   # ou STDOUT, c'est la même chose

        open(STDIN, "<&Client")
                    or die "impossible de dupliquer le client : $!";
        open(STDOUT, ">&Client")
                    or die "impossible de dupliquer le client : $!";
        open(STDERR, ">&Client")
                    or die "impossible de dupliquer le client : $!";

        # lance la calculatrice, ce n'est qu'un exemple
        system("bc -l");              # ou ce que vous préférez, pourvu
                                      # qu'il n'y ait pas de séquences
                                      # d'échappement du shell !
        print "fini\n"; # toujours au client
        close Client;
        exit;    # empêche le fils de retourner à accept
    }
```

Ce serveur clône un fils avec fork pour chaque requête entrante. De cette manière, il peut s'occuper de plusieurs requêtes à la fois, tant que vous pouvez créer d'autres processus. (Il se pourrait que vous vouliez fixer une limite à ceci.) Même si vous ne faites pas de `fork`, le `listen` permettra d'avoir jusqu'à `SOMAXCONN` (généralement cinq ou plus) connexions en attente. Chaque connexion utilise des ressources, toutefois moins qu'un processus. Les serveurs faisant des `fork` doivent prendre soin de nettoyer une fois que leurs fils (appelés des zombies dans le jargon d'Unix) ont terminé sinon il rempliraient vite votre table de processus. Le code `FOSSOYEUR`, présenté au paragraphe « Signaux » s'en occupera pour vous ou vous pouvez affecter `$SIG{CHLD} = 'IGNORE'`.

Avant de lancer une autre commande, nous connectons l'entrée et la sortie (et la sortie d'erreurs) standard à la connexion du client. De cette manière, toute commande lisant depuis `STDIN` et écrivant vers `STDOUT` peut également parler à la machine distante. Sans la réaffectation, la commande ne pourrait pas trouver le handle du client — qui, par défaut, est de toute façon fermé après l'`exec`.

Lorsque vous écrivez un serveur en réseau, nous vous encourageons fortement à activer la vérification du marquage par **`-T`** même si vous ne tournez pas sous setuid ou setgid. C'est toujours une bonne idée pour les serveurs et tout autre programme qui est lancé pour le compte de quelqu'un d'autre (comme tous les scripts CGI) car cela réduit les chances que des étrangers soient capables de compromettre votre système. Voir la section *Gérer des données non sûres* au chapitre 23, *Sécurité,* pour plus d'informations à ce sujet.

Une autre chose à prendre en considération lors de l'écriture de programme Internet : plusieurs protocoles spécifient que la fin de ligne devrait être `CRLF`, ce qui peut se spécifier de différentes façons : `"\015\012"` ou `"\xd\xa"` ou même `chr(13).chr(10)`. Depuis la version 5.6 de Perl, écrire `v13.10` produit également la même chaîne. (Sur plusieurs machines, vous pouvez également employer `"\r\n"` pour signifier `CRLF` mais ne le faites pas si vous voulez être portable vers les Mac car la signification de `\r` et de `\n` est inversée !) Plusieurs programmes Internet accepte en fait un simple `"\012"` comme fin de ligne mais c'est parce que les programmes Internet essaient en général d'être tolérants dans ce qu'ils acceptent et rigides dans ce qu'ils émettent. (Si seulement nous arrivions à pousser les gens à agir de la sorte...)

Passage de message

Comme nous l'avons évoqué auparavant, les communications UDP impliquent une surcharge bien moindre mais ne fournisse aucune fiabilité, puisqu'il n'y a aucune promesses que les messages arriveront dans l'ordre — ou même qu'il arriveront tout court. On dit souvent qu'UDP signifie *Unreliable Datagram Protocol* (protocole de datagrammes non fiable).

Néanmoins, UDP apporte des avantages par rapport à TCP, y compris la capacité de faire de la diffusion large (*broadcast*) ou multiple (*multicast*) à un ensemble complet d'hôtes destinataires en une seule fois (généralement utilisée sur votre sous-réseau local). Si vous vous trouvez excessivement soucieux à propos de la fiabilité et que vous commencez à construire des vérifications dans votre système de messages, vous devriez probablement commencer par utiliser TCP. Il est vrai qu'il est plus coûteux de mettre en place et de se séparer d'une connexion TCP mais si vous pouvez amortir cela sur plusieurs messages (ou un message très long), cela importe peu.

De toute façon, voici un exemple d'un programme UDP. Il contacte le port UDP *time* des machines passées comme paramètres dans la ligne de commande ou toutes les personnes qu'il peut trouver en utilisant l'adresse universelle de *broadcast* si aucun argument n'a été donné.[13] Toutes les machines n'ont pas de serveur d'horloge activé mais celles pour qui c'est le cas vous renverront un entier de 4 octets, empaqueté dans l'ordre « réseau », représentant le nombre de secondes depuis 1900. Vous devez soustraire le nombre de secondes entre 1900 et 1970 pour fournir cette date au fonctions de conversion `localtime` ou `gmtime`.

```
#!/usr/bin/perl
# retardhorloge - compare les horloges d'autres systèmes avec celle-ci
#                 sans arguments, broadcast à quiconque est à l'écoute.
#                 attend une réponse pendant une demi-seconde.

use v5.6.0; # or une version supérieure
use warnings;
use strict;
use Socket;

unshift(@ARGV, inet_ntoa(INADDR_BROADCAST))
    unless @ARGV;

socket(my $sock_mess, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "socket : $!";

# Certains machines ont besoin de ceci. Ça ne devrait pas faire de
# mal aux autres.
setsockopt($sock_mess, SOL_SOCKET, SO_BROADCAST, 1)
    or die "setsockopt : $!";
```

13. Si cela ne fonctionne pas, lancez `ifconfig -a` pour trouvé l'adresse de *broadcast* locale appropriée.

```
my $no_port = getservbyname("time", "udp")
    or die "pas de port udp time";

for my $cible (@ARGV) {
    print "Envoi à $cible:$no_port\n";
    my $adr_dest_p = sockaddr_in($no_port, inet_aton($cible));
    send($sock_mess, "x", 0, $adr_dest_p)
        or die "send : $!";
}

# le service d'horloge renvoie une heure sur 32 bits en secondes
# depuis 1900
my $DEPUIS_1900_A_ORIGINE = 2_208_988_800;
my $fmt_heure = "N"; # et ceci en format binaire
my $lg_heure = length(pack($fmt_heure, 1)); # n'importe quel nombre
                                            # convient

my $masque_entree = ''; # chaîne pour stocker les bits de fileno bits
                        # pour le select
vec($masque_entree, fileno($sock_mess), 1) = 1;

# n'attend une entrée qu'une demi-seconde
while (select(my $masque_sortie = $masque_entree, undef, undef, 0.5)) {
    defined(my $adr_src_p = recv($sock_mess, my $heure_bin, $lg_heure, 0))
        or die "recv : $!";
    my($port, $adr_ip) = sockaddr_in($adr_src_p);
    my $hote_emet = sprintf "%s [%s]",
                            gethostbyaddr($adr_ip, AF_INET) || 'UNKNOWN',
                            inet_ntoa($adr_ip);
    my $delta = unpack($fmt_heure, $heure_bin) -
                    $DEPUIS_1900_A_ORIGINE - time();
    print "L'horloge sur $hote_emet est en avance de $delta
           secondes sur celle-ci.\n";
}
```

17

Threads

La programmation parallèle est beaucoup plus compliquée qu'elle ne semble l'être. Imaginez que vous preniez une recette dans un livre de cuisine et que vous la convertissiez en quelque chose sur lequel plusieurs douzaines de chefs cuisiniers pourraient travailler en même temps. Vous pouvez adopter deux approches.

La première consiste à donner à chaque chef sa cuisine privée, complètement équipée avec son propre matériel et ses propres ustensiles. Pour les recettes qui peuvent être facilement décomposées et pour les aliments qui peuvent être transportés d'une cuisine à une autre, cette approche fonctionne bien car elle empêche chaque chef d'empiéter sur la cuisine de son voisin.

Sinon, il vous suffit de mettre tous les chefs dans une seule cuisine et de les laisser se débrouiller en décidant qui utilise le mixer et quand. Cela peut se transformer en pagaille, surtout lorsque les couteaux de boucher commencent à voler.

Ces deux approches correspondent à deux modèles de programmation parallèle sur les ordinateurs. Le premier est le modèle multiprocessus, typique des systèmes Unix traditionnels, dans lequel chaque thread[1] de contrôle possède son propre jeu de ressources. Le second modèle est le modèle multithread, dans lequel chaque thread de contrôle partage des ressources avec tous les autres threads de contrôle. Ou ne les partage pas, comme cela peut parfois arriver (et comme cela doit parfois arriver).

Nous savons tous que les chefs aiment prendre le contrôle ; ça ne pose pas de problème car les chefs *ont besoin* de prendre le contrôle pour accomplir ce que nous voulons qu'ils accomplissent. Mais les chefs doivent s'organiser, d'une manière ou d'une autre.

Perl implémente les deux modèles d'organisation. Dans ce chapitre nous les appellerons le *modèle de processus* et le *modèle de thread*.

1. *N.d.T.* : La traduction française d'un *thread* est « une tâche légère » ou une unité élémentaire de processus, mais comme le terme est généralement connu de tous les programmeurs et que les programmeurs Perl sont particulièrement impatients, paresseux et orgueilleux, nous emploierons littéralement le mot *thread* dans cet ouvrage, ce qui nous permet d'aller plus vite, d'en écrire moins et d'en être particulièrement fiers !

Le modèle de processus

Nous ne nous étendrons pas trop sur le modèle de processus ici, simplement parce qu'il est dominant tout au long du reste de ce livre. Perl est issu des systèmes Unix, il a donc adopté la notion que chaque processus accomplit son propre travail. Si un processus veut démarrer un traitement parallèle, il doit logiquement démarrer un processus parallèle ; c'est-à-dire qu'il doit faire un `fork` pour avoir un nouveau processus poids lourd, qui ne partage par défaut que très peu de choses avec son processus père, si ce n'est certains descripteurs de fichiers. (Le père et le fils peuvent sembler partager beaucoup plus, mais une grande partie de l'état du processus père est dupliquée dans le processus fils que réellement partagée au sens logique du terme. Le système d'exploitation peut bien entendu faire preuve de paresse quand il s'agit de forcer cette séparation logique, auquel cas nous appelons cela une sémantique de copie lors d'écritures (*copy-on-write*) mais nous ne ferions pas de copie du tout, s'il n'y avait pas en premier lieu de séparation logique.)

Historiquement, cette vision industrielle du traitement multiprocessus a posé de petits problèmes sur les systèmes Microsoft car Windows n'avait pas de modèle multiprocessus bien développé (et il ne se reposait pas souvent sur ce qu'il avait à cet égard pour la programmation parallèle). Il a typiquement adopté à la place une approche multithread.

Toutefois, au prix d'efforts héroïques, la version 5.6 de Perl implémente maintenant l'opération `fork` sur Windows en clônant un nouvel objet interpréteur à l'intérieur du même processus. Cela signifie que la plupart des exemples utilisant `fork` dans le reste de ce livre fonctionneront maintenant sur Windows. L'interpréteur clôné partage du code immuable avec les autres interpréteurs mais reçoit sa propre copie des données avec lesquelles il s'amusera. (Bien entendu, il peut toujours y avoir des problèmes avec les bibliothèques C qui ne comprennent pas les threads.)

Cette approche du traitement multiprocessus a été baptisée *ithreads*, comme raccourci pour « threads d'interpréteurs ». L'émulation de `fork` pour les systèmes Microsoft a déclenché l'implémentation des ithreads. Toutefois, nous avons rapidement réalisé que, bien les autres interpréteurs tournaient sous des threads distincts, ils tournaient dans le même processus, il serait donc facile de faire partager des données à ces interpréteurs séparés, même s'ils ne les partageaient pas par défaut.

C'est exactement l'inverse du modèle multithread typique, dans lequel tout est partagé par défaut et où vous devez vous échiner à *ne pas* partager quelque chose. Mais vous ne devriez pas voir ces deux modèles comme étant complètement distincts l'un de l'autre car ils essaient tous deux de construire un pont au-dessus de la même rivière ; ils ne font que le construire sur des berges opposées. La véritable solution à tout problème de traitement parallèle est de mêler un certain degré de partage dans un certain degré d'égoïsme.

Ainsi, le but à long terme est d'étendre le modèle d'ithreads pour permettre autant de partage que vous avez besoin ou que vous voulez. Toutefois, à l'heure où nous écrivons ces lignes, la seule interface pour les ithreads visible par l'utilisateur est l'appel `fork` sous les portages Microsoft de Perl. Nous pensons que cette approche produira éventuellement des programmes plus propres que l'approche multithread standard. À la base, il est plus facile de gérer une économie où vous présumez que chacun possède ce qu'il a, plutôt que de présumer que tout le monde possède tout. Ce n'est pas que l'on attend des gens qu'ils ne partagent pas dans une économie capitaliste, ou qu'ils ne détournent pas

de fonds publics[2] dans une économie communiste. Ces choses tendent à s'équilibrer. Il arrive que l'on soit dans une économie socialiste. Mais avec d'inportant groupes de personnes, le partage de tout par défaut ne fonctionne que si vous avez un « chef en chef » avec un grand couteau de boucher, pensant que tout lui appartient.

Bien entendu, le véritable gouvernement de tout ordinateur est géré par un dictateur fasciste connu sous le nom de système d'exploitation. Mais un dictateur avisé sait lorsqu'il faut faire croire aux gens qu'ils sont capitalistes — et lorsqu'il faut leur faire croire qu'ils sont communistes.

Le modèle de thread

Le modèle de thread pour le traitement parallèle a tout d'abord été introduit dans Perl comme une fonctionnalité expérimentale dans la version 5.005. (Nous voulons dire par « modèle de thread », des threads partageant des ressources de donnée par défaut, non les nouveaux ithreads de la version 5.6.) Par certains côtés, ce modèle de thread est toujours une fonctionnalité expérimentale même en 5.6 car Perl est un langage riche et le traitement multithread peut mettre un fouillis même dans le plus simple des langages. Il existe toujours des coins et recoins de la sémantique de Perl qui n'interagissent pas très bien avec la notion que tout soit partagé. Le nouveau modèle des ithreads est une tentative pour contourner ces problèmes et, à un moment à l'avenir, il se peut que le modèle de thread actuel soit englobé dans le modèle d'ithread (lorsque nous aurons une interface aux ithreads qui dira « partagez tout ce que vous pouvez par défaut »). Mais malgré ses problèmes, l'actuel modèle de thread « expérimental » continue d'être très utile dans plusieurs situations du mode réel où il est préférable d'être un cobaye plutôt que de ne pas avoir de threads. Des applications raisonnablement robustes peuvent être écrites en Perl avec des threads mais vous devez faire très attention. Vous devriez au moins envisager d'utiliser plutôt `fork`, si vous pouvez trouver un moyen de résoudre votre problème avec des pipes à la place de structures de données partagées.

Mais certains algorithmes sont plus simples à exprimer si de multiples tâches peuvent accéder facilement et efficacement au même jeu de données.[3] Cela produit du code plus petit et plus simple. Et puisque le noyau n'a pas à copier des tableaux de pages pour les données (même s'il fait de la copie lors d'écriture (*copy-on-write*)) lors de la création de thread, il devrait être plus rapide de démarrer une tâche de cette manière. De même, le changement de contexte peut être plus rapide si le noyau n'a pas besoin d'échanger (*swap*) des tableaux de pages. (En fait, pour les threads au niveau de l'utilisateur, le noyau n'est pas impliqué du tout — bien que les threads au niveau de l'utilisateur aient bien entendu des problèmes que les threads du noyau n'ont pas.)

Voici pour les bonnes nouvelles. Maintenant, passons à d'autres plus sujettes à réclamations. Nous avons déjà mentionné que le traitement des threads était quelque peu expérimental en Perl mais, même s'il ne l'était pas, la programmation avec des threads est

2. Détourner des fonds publics : piquer la propriété publique dans les fonds au milieu de la nuit ; détourner du public quelque chose qui n'est pas forcément de l'argent, *cf.* adopter, étendre, GPL.

3. Le modèle de mémoire partagée des IPC System V, dont nous avons parlé dans le chapitre précédent, ne peut pas être qualifié exactement comme étant « facile et efficace ».

assez traître. La capacité d'un flux d'exécution à placer, qu'on le veuille ou non, des trous dans l'espace des données d'un autre, est susceptible de provoquer des désastres, plus que vous ne pouvez l'imaginer. Vous pourriez vous dire : « C'est facile à corriger, je n'ai qu'à mettre des verrous sur les données partagées. » D'accord, le verrouillage des données partagées est indispensable mais disposer de protocoles de verrouillages qui soient corrects est une difficulté de notoriété publique, avec des erreurs générant des verrous mortels (*deadlocks*) ou des résultats non déterministes. Si vous avez des problèmes de synchronisation dans votre programme, l'emploi de threads va non seulement les aggraver mais les rendra également plus difficiles à localiser.

Vous êtes non seulement responsable de l'honnêteté de vos propres données partagées, mais également de celle des modules Perl et des bibliothèques C que vous appelez. Votre code Perl a beau être sécurisé au niveau des threads à 100 %, si vous appelez un module ou un sous-programme C qui ne soit pas sûr au niveau des threads, vous êtes grillé. Vous devriez présumer qu'aucun module n'est sûr au niveau des threads à moins de le prouver. Ceci inclut même certains des modules standard. Peut-être même la plupart d'entre eux.

Nous ne vous avons pas déjà découragé ? Non ? Alors nous mettrons l'accent sur le fait que vous êtes quelque peu à la merci des bibliothèques de votre systèmes d'exploitation pour le traitement des threads lorsqu'il s'agit de politiques d'ordonnancement et de préemption. Certaines bibliothèques de thread ne font du basculement de thread que sur les appels système bloquants. Certaines bloquent le processus entier si un seul thread fait un appel système bloquant. Certaines ne basculent les threads que sur l'expiration d'un délai quantifiable (soit d'un thread, soit d'un processus). Certaines ne basculent les threads qu'explicitement.

Oh, et pendant qu'on y est, si votre processus reçoit un signal, déterminer à quel thread ce signal est délivré est complètement dépendant du système.

Pour faire de la programmation de thread en Perl, vous devez compilez une version spéciale de Perl en suivant les instructions fournies dans le fichier *README.threads* dans le répertoire des sources de Perl. Il est à peu près garanti que ce Perl spéciale tournera moins vite que votre exécutable Perl standard.

Ne présupposez pas que, puisque vous savez comment les threads sont programmés dans d'autres modèles (POSIX, DEC, Microsoft, etc.), vous savez comment les threads fonctionnent en Perl. Comme avec d'autres choses en Perl, Perl est du Perl, non du C++ ou du Java ou quoi que ce soit d'autre. Par exemple, il n'y a pas de priorités de thread en temps réel (et aucun moyen de contourner leur absence). Il n'y a pas non plus de verrous d'exclusion mutuelle (*mutex*). Utilisez simplement le verrouillage ordinaire ou peut-être le module `Thread::Semaphore` ou les mécanismes de `cond_wait`.

Toujours pas découragé ? Bien ! Car les threads sont vraiment sympas. Vous pouvez prévoir de vous amuser.

Le module Thread

L'interface actuelle pour les threads en Perl est définie par le module `Thread`. De plus, un nouveau mot-clé de Perl a été ajouté, l'opérateur `lock`. Nous parlerons de `lock` ultérieurement dans ce chapitre. Les autres modules standard de thread sont construits à partir de cette interface de base.

Le module `Thread` fournit ces méthodes de classe :

Méthode	Utilisation
`new`	Construit un nouvel objet `Thread`.
`self`	Renvoie mon objet `Thread` courant.
`list`	Renvoie la liste des objets `Thread`.

Et pour les objets `Thread`, il fournit ces méthodes d'objet :

Méthode	Utilisation
`join`	Moissonne un thread (propage les erreurs).
`eval`	Moissonne un thread (capture les erreurs).
`equal`	Compare l'identité de deux threads.
`tid`	Renvoie l'ID de thread interne.

De plus, le module `Thread` fournit ces fonctions que l'on peut importer :

Fonction	Utilisation
`yield`	Dit à l'ordonnanceur de lancer un thread différent.
`async`	Construit un nouveau thread *via* une fermeture.
`cond_signal`	Réveille exactement un thread qui fait un `cond_wait()` sur une variable.
`cond_broadcast`	Réveille tous les threads qui font un `cond_wait()` sur une variable.
`cond_wait`	Attend sur une variable jusquà être réveillé par un `cond_signCal()` ou un `cond_broadcast()` sur cette variable.

Création de thread

Vous pouvez engendrer un thread de deux manières, soit en utilisant la méthode de classe `Thread->new`, soit en utilisant la fonction `async`. Dans tous les cas, la valeur renvoyée est un objet `Thread`. `Thread->new` prend une référence de code indiquant la fonction à lancer et des arguments à passe à cette fonction :

```
use Thread;
...
$t = Thread->new(\&fonc, $arg1, $arg2);
```

Vous vous retrouvez souvent à vouloir passer une fermeture comme premier argument sans fournir d'arguments supplémentaires :

```
my $quelque_chose;
$t = Thread->new( sub { dit($quelque_chose) } );
```

Dans ce cas spécial, la fonction `async` donne un relief dans l'écriture (c'est-à-dire du sucre de syntaxe) :

```
use Thread qw(async);
```

```
...
my $quelque_chose;
$t = async {
    dit($quelque_chose);
};
```

Vous remarquerez que nous avons importé explicitement la fonction `async`. Bien entendu, vous pouvez également utiliser le nom pleinement qualifié `Thread::async` à la place, mais alors votre sucre de syntaxe n'est plus aussi doux. Puisqu'`async` ne prend qu'une fermeture, tout ce que vous voulez lui passer doit être une variable lexicale qui soit à ce moment dans la portée.

Destruction de thread

Une fois commencé — et sujet aux lubies de votre bibliothèque de thread — le thread continuera à tourner tout seul jusqu'à ce que sa fonction de plus haut niveau (celle que vous avez passée au constructeur) renvoie un résultat. Si vous voulez terminer un thread plus tôt, il vous suffit de faire `return` depuis l'intérieur de cette fonction de plus haut niveau.[4]

Maintenant, tout est parfait pour que votre thread se termine en renvoyant un résultat, mais *à qui* le renvoyer ? Le thread qui a engendré ce thread-ci est vraisemblablement parti faire d'autres choses et n'attend plus de réponse à l'appel de la méthode. La réponse est assez simple : le thread attend jusqu'à ce que quelqu'un émette un appel de méthode qui *fasse* l'attente d'une valeur de retour. Cet appel de méthode s'appelle `join` (joindre) car il fait se rejoindre de manière conceptuelle deux threads en un seul :

```
$val_ret = $t->join();   # moissonne le thread $t
```

L'opération `join` est une réminiscence de `waitpid` sur un processus fils. Si le thread a déjà terminé, la méthode `join` rend la main immédiatement avec la valeur valeur renvoyée par la fonction principale du thread. Si le thread n'a pas fini, `join` se comporte comme un appel bloquant qui interrompt le thread appelant indéfiniment. (Il n'y a pas de mécanisme de minuterie (*time-out*).) Lorsqu'éventuellement le thread s'achève, le `join` renvoie une valeur.

Toutefois, au contraire de `waitpid`, qui ne peut moissonner que les propres fils d'un processus, n'importe quel thread peut faire un `join` sur n'importe quel autre thread à l'intérieur du processus. C'est-à-dire qu'il n'est pas nécessaire que le thread qui fait le `join` soit le thread principal ou le thread parent. La seule restriction est qu'un thread ne peut pas faire de `join` sur lui-même (ce qui reviendrait à officier pour vos propres funérailles) et qu'un thread ne peut pas faire de `join` sur un thread qui a déjà répondu à un `join` (ce qui équivaudrait à deux directeurs des pompes funèbres se battant au-dessus de la dépouille). Si vous essayez de faire l'une de ces choses, une exception sera levée.

La valeur de retour de `join` ne doit pas forcément être une valeur scalaire — elle peut également être une liste :

4. N'appelez pas `exit` ! Cela essayerait de quitter votre processus entier et y arriverait peut-être. Mais en fait, le processus ne se terminera pas avant que tous les threads ne le fassent et certains d'entre eux peuvent refuser de se terminer sur un `exit`. Nous donnerons plus d'informations là-dessus plus tard.

```
use Thread 'async';

$t1 = async {
    my @trucs = getpwuid($>);
    return @trucs;
};

$t2 = async {
    my $mess = `cat /etc/motd`;
    return $mess;
};

@liste_ret = $t1->join();
$val_ret = $t2->join();

print "Le 1er fils a renvoyé @liste_ret\n";
print "Le 2ème fils a renvoyé $val_ret\n";
```

En fait, l'expression renvoyée par un thread est toujours évaluée dans un contexte de liste, même si `join` est appelé dans un contexte scalaire, auquel cas la dernière valeur de la liste est renvoyée.

Capture des exceptions avec join

Si un thread se termine avec une exception non capturée, cela ne tuera pas immédiatement le programme entier. Cela serait embêtant. À la place, lorsqu'un `join` est lancé sur ce thread, le `join` lève lui-même l'exception. L'emploi de `join` sur un thread indique une volonté de propager toute exception levée par ce thread. Si vous préférez capturer l'exception à ce moment et à cet endroit, utilisez la méthode `eval`, qui comme la fonction interne du même nom, fait que l'exception est mise dans `$@` :

```
$val_ret = $t->eval();      # capture les erreurs lorsque les
                            # threads se rejoignent
if ($@) {
    warn "échec du thread : $@";
}
else {
    print "le thread a renvoyé $ret_val\n";
}
```

Bien qu'il n'y ait pas de règle à cet effet, vous pourriez vouloir adopter l'habitude de ne rejoindre un thread que depuis le thread qui a créé celui que vous rejoignez. C'est-à-dire que vous ne moissonnez un thread fils que depuis le thread père qui l'a engendré. Cela rend plus facile la conservation d'une trace indiquant quelles exceptions vous auriez besoin de capturer et quand le faire.

La méthode detach

Comme autre solution pour arrêter les threads, si vous n'avez pas prévu de rejoindre un thread plus tard avec `join` pour obtenir sa valeur de retour, vous pouvez appeler la méthode `detach` sur ce thread pour que Perl le nettoie à votre place. C'est un peu comme lorsqu'un processus est hérité par le programme *init* sur Unix, sauf que le seul moyen

de faire cela sur Unix est que le processus père meure.

La méthode `detach` ne met pas le thread « en arrière plan » ; si vous essayez de sortir du programme principal et qu'un thread détaché est toujours en train de tourner, la fin du programme bloquera jusqu'à ce que le thread lui-même se termine. Mais plutôt, la méthode `detach` vous épargne les tâches de nettoyage. Elle indique purement et simplement à Perl de ne pas garder la valeur de retour, ni la valeur de sortie du thread après qu'il a fini. En un sens, `detach` dit à Perl de faire un `join` implicite lorsque le thread finit puis de jeter les résultats. Cela peut avoir de l'importance : si vous n'avez jamais fait de `join`, ni de `detach`, sur un thread qui renvoie une liste très grande, cet espace de stockage sera perdu jusqu'à la fin. En effet, Perl devrait s'agripper à lui dans la perspective lointaine (mais alors très lointaine, dans ce cas), où quelqu'un voudrait faire un `join` sur ce thread à un moment à l'avenir.

Une exception levée dans un thread fils détaché ne se propage plus non plus durant un `join`, puisque les threads ne s'uniront jamais. Utilisez sagement `eval {}` dans la fonction de plus haut niveau et trouvez d'autres moyens de rapporter les erreurs.

Identification des threads

Chaque thread de Perl possède un numéro d'identification de thread distinct, qui est celui que la méthode `tid` renvoie :

```
$son_num_tid = $t1->tid();
```

Un thread peut accéder à son propre objet thread par l'intermédiaire de l'appel `Thread->self`. Ne confondez pas cela avec l'ID du thread : pour calculer son propre ID, un thread fait ceci :

```
$mon_tid = Thread->self->tid();  # $$ pour les threads, pour ainsi dire
```

Pour comparer un objet thread avec un autre, employez l'une de ces techniques :

```
Thread::equal($t1, $t2)
$t1->equal($t2)
$t1->tid() == $t2->tid()
```

Liste des threads courants

Vous pouvez obtenir une liste des objets threads courants dans le processus courant en utilisant l'appel de méthode de classe `Thread->list`. La liste comprend à la fois les threads en train de tourner et ceux qui ont fini mais sur lesquels aucun `join` n'a encore été appliqué. Vous pouvez faire ceci depuis n'importe quel thread.

```
for my $t (Thread->list()) {
    printf "$t a un tid = %d\n", $t->tid();
}
```

Priorité au processeur

Le module `Thread` implémente une fonction, que vous pouvez importer, appelée `yield`. Elle fait en sorte que le thread appelant abandonne le processeur. Malheureusement, les détails de ce mécanisme sont complètement dépendants votre version d'implémentation des threads. Néanmoins, abandonner le contrôle de la CPU occasionnellement est considérée comme un geste d'amabilité :

```
use Thread 'yield';
yield();
```

L'utilisation des parenthèses n'est pas une obligation. C'est même encore plus sûr, syntaxiquement parlant, car la faute de frappe apparemment inévitable « yeild » est alors interceptée :

```
use strict;
use Thread 'yield';
yeild;         # Le compilateur se désole, puis arrête tout
yield;         # Ok.
```

Accès au données

Ce que nous savons passé en revue jusqu'ici n'était pas bien compliqué; mais nous sommes sur le point d'y remédier. Rien de ce que nous avons fait ne mettait en pratique la nature parallèle des threads. L'accès aux données partagées change tout cela.

Le code mis dans un thread en Perl possède les mêmes limites en ce qui concerne la visibilité des données que tout autre morceau de code en Perl. On a toujours accès aux variables globales *via* des tables de symboles globales et aux variables lexicales *via* une portée lexicale (mémoire de travail, *scratchpad*).

Toutefois, le fait que de multiples threads de contrôle existent dans le programme génère des erreurs dans les travaux. Deux threads ne peuvent être autorisés à accéder à la même variable globale simultanément ou ils peuvent s'écraser l'un l'autre. (Le résultat de l'écrasement dépend de la nature de l'accès.) De même, deux threads ne peuvent être autorisés à accéder à la même variable lexicale simultanément car les variables lexicales se comportent également comme les globales si elles sont déclarées en dehors de la portée de fermetures utilisées par des threads. Démarrer des threads *via* des références de sous-programmes (en utilisant `Thread->new`) plutôt que *via* des fermetures (en utilisant `async`) peut aider à limiter l'accès aux variables lexicales, si c'est ce que vous voulez. (Parfois ce n'est cependant pas le cas.)

Perl résout le problème pour certaines variables spéciales internes, comme `$!`, `$_`, `@_` et les autres variables comme celles-ci, en en faisant des données spécifiques au thread. La mauvaise nouvelle est que toutes vos variables de paquetage, basiques, que vous utilisez quotidiennement, ne sont pas protégées des écrasements.

La bonne nouvelle est que vous n'avez généralement pas à vous inquiéter à propos de vos variables lexicales, en supposant qu'elles soient déclarées à l'intérieur du thread courant, puisque chaque thread instanciera l'entrée de sa propre portée lexicale, qui est distincte de celle de tout autre thread. Vous ne devez vous soucier des variables lexicales que si elles sont partagées entre les threads, par exemple en passant des références ou en référençant les variables lexicales à l'intérieur de fermetures utilisées par de multiples threads.

Synchronisation des accès avec lock

Lorsque plusieurs agents peuvent accéder au même élément en même temps, des collisions surviennent, exactement comme à un carrefour. Le verrouillage du carrefour est votre seule défense.

La fonction interne `lock` est le mécanisme de Perl correspondant à un feu tricolore pour le contrôle des accès. Bien que `lock` soit en quelque sorte un mot-clé, c'est un mot-clé timide car la fonction interne n'est *pas* utilisée si le compilateur a déjà vu un définition `sub lock {}` dans le code de l'utilisateur. Ceci est destiné à assurer une compatibilité antérieure. Cependant, `CORE::lock` désigne toujours la fonction interne. (Dans un *perl* qui n'a pas été compilé pour le traitement des threads, l'appel de `lock` n'est pas une erreur ; c'est une opération sans danger ne faisant rien (*no-op*), du moins dans les versions récentes.)

Exactement de la même manière que l'opérateur `flock` ne bloque que d'autres instances de `flock`, et non les véritables entrées/sorties, l'opérateur `lock` également, ne bloque que d'autres instances de `lock`, et non les accès ordinaires aux données. Ce sont, en effet, des verrous *consultatifs*. Exactement comme les feux tricolores.[5]

Vous pouvez faire un `lock` sur des variables scalaires individuelles aussi bien que sur des tableaux ou des hachages entiers.

```
lock $var;
lock @valeurs;
lock %hachage;
```

Cependant, l'emploi de `lock` sur des valeurs agrégées ne verrouille pas implicitement les composants scalaires de ces valeurs agrégées :

```
lock @valeurs;       # dans le thread 1
...
lock $valeurs[23];   # dans le thread 2 --- ne verrouille pas !
```

Si vous verrouillez un référence, cela verrouille automatiquement l'accès au référant. C'est-à-dire que vous obtenez gratuitement une déréférence. Ceci est bien commode car les objets sont toujours cachés derrière un référence et vous voulez souvent verrouiller des objets. (Et vous ne voulez presque jamais verrouiller des références.)

Le problème avec les feux tricolores, bien entendu, est qu'ils sont rouge la plupart du temps et que vous devez attendre. De même, `lock` est un appel bloquant — votre thread sera suspendu à cet endroit jusqu'à ce que le verrou soit accordé. Il n'y a pas de mécanisme de minuterie (*time-out*). Il n'y a pas non plus de mécanisme de déverrouillage car les verrous ont une portée dynamique. Ils durent jusqu'à ce que leur bloc, leur fichier ou leur `eval` soit fini. Lorsqu'ils sortent de la portée, il sont automatiquement libérés.

Les verrous sont également récursifs. Cela signifie que si vous verrouillez une variable dans une fonction et que cette fonction s'appelle récursivement pendant que vous détenez le verrou, le même thread pourra verrouiller avec succès la même variable à nouveau. Le verrou est finalement abandonné lorsque l'on est sorti de tous les niveaux détenant les verrous.

Voici une démonstration simple de ce qui peut se passer si vous n'avez pas verrouillé. Nous forçons un changement de contexte en utilisant `yield` pour montrer le genre de

5. Certains feux de passage à niveau sont obligatoires (ceux avec une barrière) et certaines personnes pensent que les verrous devraient l'être également. Mais imaginez seulement un monde dans lequel chaque carrefour aurait des armes se levant et se baissant lorsque les feux changeraient.

problème qui peut également arriver accidentellement avec un ordonnanceur préemptif :

```
use Thread qw/async yield/;
my $var = 0;
sub une_inflation {
    if ($var == 0) {
        yield;
        $var++;
    }
}

my $t1 = new Thread \&une_inflation;
my $t2 = new Thread \&une_inflation;

for my $t ($t1, $t2) { $t->join }
print "var vaut $var\n";
```

Ce code affiche toujours `2` (pour une certaine définition de « toujours ») car nous avons décidé de l'inflation après avoir vu que la variable valait `0` mais avant que nous puissions le faire, un autre thread a décidé d'en faire autant.

Nous pouvons résoudre cette collision par l'ajout trivial d'un verrou avant d'examiner `$var`. Maintenant ce code affiche toujours `1` :

```
sub une_inflation {
    lock $var;
    if ($var == 0) {
        yield;
        $var++;
    }
}
```

Souvenez-vous qu'il n'y a pas de fonction `unlock` explicite. Pour contrôler le déverrouillage, vous n'avez qu'à ajouter un autre niveau de portée, qui soit imbriqué, pour que le verrou soit relâché lorsque cette portée se termine :

```
sub une_inflation {
    {
        lock $var;
        if ($var == 0) {
            yield;
            $var++;
        }
    } # le verrou est relâché ici !
    # autre code avec $var déverrouillée
}
```

Verrous mortels (deadlocks)

Le verrou mortel (*deadlock*) est le fléau des programmeurs de threads car il est facile d'en créer par accident et difficile de l'éviter. Voici un exemple simple de verrou mortel :

```
my $t1 = async {
    lock $a; yield; lock $b;
    $a++; $b++
```

```
};
my $t2 = async {
    lock $b; yield; lock $a;
    $b++; $a++
};
```

La solution ici est que tous les protagonistes qui on besoin d'un ensemble de verrous particulier s'en saisissent dans le même ordre.

C'est également une bonne chose de minimiser la durée pendant laquelle vous détenez des verrous. (Au moins, c'est une bonne chose de le faire pour des questions de performance. Mais si vous le faites pour réduire les risques de verrou mortel, tout ce que vous faites est de le rendre plus difficile à reproduire et à diagnostiquer le problème.)

Verrouillage des sous-programmes

Vous pouvez également mettre un verrou sur un sous-programme :

```
lock &fonc;
```

Au contraire des verrous sur les données, qui ne sont que consultatifs, les verrous de sous-programmes sont *obligatoires*. Personne d'autre que le thread détenant le verrou peut entrer dans le sous-programme.

Considérez le code suivant, qui contient des situations de concurrence (*race conditions*) impliquant la variable `$fini`. (Les `yield` ne sont mis qu'à titre de démonstration.)

```
use Thread qw/async yield/;
my $fini = 0;
sub bricole {
    my $arg = shift;
    my $tid = Thread->self->tid;
    print "thread $tid : bricole $arg\n";
    yield;
    unless ($fini) {
        yield;
        $fini++;
        bricole($arg + 10);
    }
}
```

Si vous le lancez de cette façon :

```
my @t;
for my $i (1..3) {
    push @t, Thread->new(\&bricole, $i);
}
for (@t) { $_->join }
print "fini vaut $fini\n";
```

voici l'affichage (enfin, certaines fois — ce n'est pas déterministe) :

```
thread 1 : bricole 1
thread 2 : bricole 2
thread 3 : bricole 3
thread 1 : bricole 11
```

```
thread 2 : bricole 12
thread 3 : bricole 13
fini vaut 3
```

Toutefois, si vous le lancez de cette façon :

```
for my $i (1..3) {
    push @t, async {
        lock &bricole;
        bricole($i);
    };
}
for (@t) { $_->join }
print "fini vaut $fini\n";
```

voici l'affichage :

```
thread 1 : bricole 1
thread 1 : bricole 11
thread 2 : bricole 2
thread 3 : bricole 3
fini vaut 1
```

L'attribut locked

Bien qu'il soit obligatoire d'obéir à un verrou de sous-programme, rien ne force personne à les verrouiller en premier lieu. Mais certains sous-programmes aimeraient vraiment être capables d'exiger d'être verrouillés avant d'être appelés.

L'attribut de sous-programme `locked` remplit cette fonction. Il est plus rapide que l'appel `lock &fonc` car il est connu à la compilation, non seulement à l'exécution. Mais le comportement est le même que lorsque nous l'avons verrouillé explicitement auparavant. Voici la syntaxe :

```
sub bricole : locked {
    # comme avant
}
```

Si vous avez un prototype de fonction, il vient entre le nom et tous les attributs :

```
sub bricole ($) : locked {
    # comme avant
}
```

Verrouillage des méthodes

Le verrouillage automatique d'un sous-programme est vraiment très pratique mais il en fait parfois un peu trop. Lorsque vous invoquez une méthode d'objet, cela n'a généralement aucune importance si de multiples méthodes sont lancées simultanément pour autant qu'elles le sont toutes pour le compte d'objets différents. Vous aimeriez donc vraiment verrouiller plutôt l'objet sur lequel cette méthode est appelée. L'ajout d'un attribut `method` à la définition du sous-programme fait ceci :

```
sub bricole : locked method {
    # comme avant
}
```

Si ce code est appelé en tant que méthode, l'objet qui l'invoque est verrouillé, fournissant un accès séquentiel à cet objet mais permettant à la méthode d'être appelée sur d'autres objets. Si la méthode n'est pas appelée sur un objet, l'attribut essaie toujours de faire ce qu'il faut : si vous appelez une méthode verrouillée comme une méthode de classe (`Paquetage->new` plutôt que `$obj->new`), la table de symboles du paquetage est verrouillée. Si vous appelez une méthode verrouillée comme un sous-programme normal, Perl lèvera une exception.

Variables de condition

Une variable de condition permet à un thread d'abandonner le processeur jusqu'à ce qu'un certain critère soit satisfait. Les variables de conditions indiquent des points de coordination entre les threads lorsque vous avez besoin de plus de contrôle que ce qu'un seul verrou fournit. D'un autre côté, vous n'avez pas vraiment besoin de plus de temps système (*overhead*) qu'un verrou ne fournit et les variables de condition sont conçues dans cet esprit. Vous n'avez qu'à utiliser des verrous ordinaires plus des conditions ordinaires. Si la condition échoue, vous devrez alors prendre des mesures extraordinaires *via* la fonction `cond_wait` ; mais nous optimisons pour les chances de réussite, puisque dans une application bien conçue, nous ne devrions pas de toute façon nous trouver dans un goulot d'étranglement sur la condition courante.

Outre le verrouillage et le test, les opération de base sur les variables de condition consistent à envoyer ou recevoir un événement « signal » (non un signal réel au sens de `%SIG`). Soit, vous suspendez votre exécution pour attendre la réception d'un événement, soit vous envoyez un événement pour réveiller d'autres threads attendant une condition particulière. Le module `Thread` fournit trois fonctions, que vous pouvez importer, pour faire cela : `cond_wait`, `cond_signal` et `cond_broadcast`. Ce sont les mécanismes primaires sur lesquels se basent des modules plus abstraits comme `Thread::Queue` et `Thread::Semaphore`. Il est souvent plus commode d'utiliser ces abstractions, lorsque c'est possible.

La fonction `cond_wait()` prend une variable déjà verrouillée par le thread courant, déverrouille cette variable, puis bloque jusqu'à ce qu'un autre thread fasse un `cond_signal` ou un `cond_broadcast` pour la même variable verrouillée.

La variable bloquée par `cond_wait` est reverrouillée après que le retour de `cond_wait`. Si de multiples threads font un `cond_wait` sur la même variable, tous rebloquent sauf un car ils ne peuvent regagner le verrou sur la variable. Ainsi, si vous n'utilisez que `cond_wait` pour la synchronisation, abandonnez le verrou dès que possible.

La fonction `cond_signal` prend une variable déjà verrouillée par le thread courant et débloque l'un des threads qui était actuellement dans un `cond_wait` sur cette variable. Si plus d'un thread bloquait dans un `cond_wait` sur cette variable, un seul est débloqué et vous ne pouvez pas prédire lequel. Si aucun thread ne bloquait dans un `cond_wait` sur cette variable, l'événement est éliminé.

La fonction `cond_broadcast` fonctionne comme `cond_signal` mais elle débloque toutes les threads bloqués dans un `cond_wait` sur la variable verrouillée, non un seul thread. (Bien entendu, c'est toujours le cas s'il n'y a qu'un seul thread détenant le verrou.)

La fonction `cond_wait` est destinée à être le genre de chose qu'un thread n'emploie en dernier ressort que si la condition qu'il veut n'est pas remplie. Le `cond_signal` et le

`cond_broadcast` indiquent que la condition est en train de changer. Le déroulement des actions est supposé être le suivant : verrouillez, puis vérifiez pour voir si la condition que vous voulez est remplie ou non ; si c'est le cas, c'est bon, sinon, faites un `cond_wait` jusqu'à ce que ce *soit* bon. L'accent devrait être mis sur le fait d'éviter de bloquer autant que possible. (C'est généralement un bon conseil lorsque l'on manipule des threads.)

Voici un exemple où le contrôle passe d'un thread à l'autre. Ne soyez pas bluffé par le fait que les conditions réelles sont reléguées dans la partie droite des modificateurs d'instruction ; `cond_wait` ne sera jamais appelé sauf si la condition que nous attendons est fausse.

```
use Thread qw(async cond_wait cond_signal);
my $var_wait = 0;
async {
    lock $var_wait;
    $var_wait = 1;
    cond_wait $var_wait until $var_wait == 2;
    cond_signal($var_wait);
    $var_wait = 1;
    cond_wait $var_wait until $var_wait == 2;
    $var_wait = 1;
    cond_signal($wait_var);
};

async {
    lock $var_wait;
    cond_wait $var_wait until $var_wait == 1;
    $var_wait = 2;
    cond_signal($var_wait);
    cond_wait $var_wait until $var_wait == 1;
    $var_wait = 2;
    cond_signal($wait_var);
    cond_wait $var_wait until $var_wait == 1;
};
```

Autres modules de thread

Plusieurs modules sont construits au-dessus de la primitive `cond_wait`.

Files d'attente (queues)

Le module standard `Thread::Queue` fournit deux moyens de passer des objets entre des threads sans s'inquiéter des verrous ou de la synchronisation. Cette interface est bien plus facile :

Méthode	Utilisation
`new`	Construit un nouvel objet `Thread::Queue`.
`enqueue`	Pousse un ou plusieurs scalaires à la fin de la file.
`dequeue`	Enlève le premier scalaire du début de la file. La méthode `dequeue` bloque s'il n'y a plus d'éléments.

Remarquez comme une file d'attente est similaire à un pipe ordinaire, sauf qu'au lieu d'envoyer des octets, vous arrivez à passer des scalaires purs et durs, y compris des références et des objets consacrés avec `bless`.

Voici un exemple dérivé de la page de manuel de *perlthrtut* :

```
use Thread qw/async/;
use Thread::Queue;

my $Q = Thread::Queue->new();
async {
    while (defined($donnee = $Q->dequeue)) {
        print "$donnee retiré de la file\n";
    }
};

$Q->enqueue(12);
$Q->enqueue("A", "B", "C");
$Q->enqueue($thr);
sleep 3;
$Q->enqueue(\%ENV);
$Q->enqueue(undef);
```

Et voici ce que vous obtenez comme affichage :

```
12 retiré de la file A retiré de la file
B retiré de la file
C retiré de la file
Thread=SCALAR(0x8117200) retiré de la file
HASH(0x80dfd8c) retiré de la file
```

Remarquez comment `$Q` était dans la portée lorsque le thread asynchrone a été lancé *via* une fermeture de `async`. Les threads respectent les mêmes règles concernant les portées que n'importe quoi d'autre en Perl. L'exemple ci-dessus n'aurait pas fonctionné si `$Q` avait été déclaré après l'appel à `async`.

Sémaphores

`Thread::Semaphore` vous fournit des objets sémaphores, sûrs vis-à-vis des threads et capables de compter, pour implémenter vos opérations favorites `p()` et `v()`. Comme la plupart d'entre nous n'associent pas ces opérations avec les mots hollandais *passeer* (« passer ») et *verlaat* (« laisser »), le module appelle ces opérations « *down* » (diminuer) et « *up* » (augmenter), respectivement. (On trouvera dans la littérature, « *wait* » (attendre) ou « *signal* » (signaler).) Les méthodes suivantes sont implémentées :

Méthode	Utilisation
`new`	Construit un nouvel objet `Thread::Semaphore`.
`down`	Alloue un ou plusieurs éléments.
`up`	Désalloue un ou plusieurs éléments.

La méthode `new` crée un nouveau sémaphore et initialise son compteur au nombre spé-

cifié. Si aucun nombre n'est spécifié, le compteur du sémaphore est positionné à 1. (Le nombre représente une certaine réserve d'élément qui peut « s'épuiser » si tous les éléments sont désalloués.)

```
use Thread::Semaphore;
$mutex = Thread::Semaphore->new($MAX);
```

La méthode `down` décrémente le compteur du sémaphore du nombre spécifié, ou de 1 si aucun nombre n'est donné. Elle peut être interprétée comme une tentative d'allocation d'une partie ou de la totalité d'une ressource. Si le compteur du sémaphore descend en dessous de zéro, cette méthode bloque jusqu'à ce qu'elle compteur soit supérieur ou égal à la quantité que vous avez demandée. Appelez-le comme ceci :

```
$mutex->down();
```

La méthode `up` incrémente le compteur du sémaphore du nombre spécifié, ou de 1 si aucun nombre n'est donné. Elle peut être interprétée comme une tentative de désallocation d'une certaine quantité d'une ressource précédemment allouée. Ceci débloque au moins un thread qui était bloqué en essayant de faire un `down` sur le sémaphore, pourvu que le `up` augmente le compteur du sémaphore au-dessus de ce que le `down` essaie de le diminuer. Appelez-le comme ceci :

```
$mutex->up();
```

Autres modules standard pour le traitement des threads

`Thread::Signal` vous permet de démarrer un thread qui est désigné pour recevoir les signaux `%SIG` de votre processus. Ceci traite le problème toujours vexant des signaux qui ne sont pas fiables tels qu'ils sont actuellement implémentés dans Perl et leur utilisation imprudente peut de temps en temps causer des *core dumps*.

Ces modules sont toujours en développement et peuvent ne pas entraîner l'effet escompté sur votre système. D'un autre côté, ils peuvent produire les résultats désirés. Si ce n'est pas le cas, c'est parce que quelqu'un comme vous ne les a pas corrigés. Peut-être que quelqu'un comme vous devrait se lancer et donner un coup de main.

18

Compilation

Si vous êtes venu ici pour trouver un compilateur Perl, vous serez peut-être surpris d'apprendre que vous en avez déjà un — votre programme perl (généralement /usr/bin/perl) renferme déjà un compilateur Perl. Ce n'est peut-être pas ce que vous pensiez et si tel est le cas, vous serez heureux d'apprendre que nous fournissons également des générateurs de code (que certaines personnes bien pensantes appellent « compilateurs ») et nous en discuterons vers la fin de ce chapitre. Mais nos voulons tout d'abord parler de ce que nous pensons être Le Compilateur. Il y aura inévitablement dans ce chapitre, une certaine quantité de petits détails qui intéresseront certains et ennuieront d'autres. Si vous pensez que vous n'êtes pas intéressé, considérez cela comme l'occasion d'entraîner vos facultés de lecture à grande vitesse.

Imaginez que vous êtes un chef d'orchestre ayant commandé la partition pour une œuvre immense. Lorsque le coffret arrive, vous trouvez plusieurs douzaines de livrets, un pour chaque membre de l'orchestre, ne contenant que sa propre partition. Mais il manque curieusement votre copie principale avec toutes les partitions. Encore plus curieusement, les partitions que vous avez écrites *vous-même* sont retranscrites en français au lieu d'être en notation musicale. Avant que vous puissiez mettre en place un concert susceptible d'être présenté au public, ou même simplement donner la musique à jouer à votre orchestre, il vous faudra traduire les descriptions en prose vers le système normal de notes et de mesures. Puis vous aurez besoin de compiler les partitions individuelles en une seule gigantesque pour que vous ayez une idée du programme dans son ensemble.

De même, lorsque vous remettez le code source de votre script Perl à l'exécutable *perl* pour qu'il l'exécute, il n'est pas plus utile à l'ordinateur que ne l'était la description en français de la symphonie aux musiciens. Avant que votre programme puisse tourner, Perl a besoin de compiler[1] ces directives ressemblant à un langage naturel vers une représentation symbolique spéciale. Votre programme ne tourne pourtant toujours pas car le compilateur ne fait que compiler. Comme la partition du chef d'orchestre, même

1. Ou traduire, ou transformer, ou transfigurer, ou transmuter, ou métamorphoser.

après que votre programme a été converti en un format d'instructions convenant à l'interprétation, il a encore besoin d'un agent actif pour interpréter ces instructions.

Cycle de vie d'un programme Perl

Vous pouvez décomposer le cycle de vie d'un programme Perl en quatre phases distinctes, chacune avec ses propres étapes indépendantes. La première et la dernière sont les plus intéressantes et les deux du milieu sont optionnelles. Ces phases sont décrite à la *figure 18-1*.

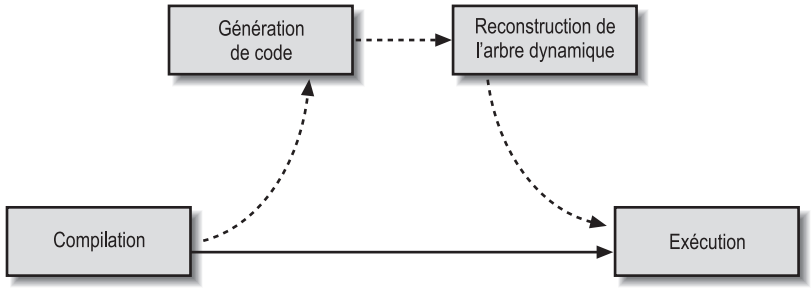


Figure 18-1. Le cycle de vie d'un programme Perl

1. *La phase de compilation*

   Durant la phase 1, la *phase de compilation*, le compilateur convertit votre programme vers une structure de données appelée un *arbre syntaxique*. En plus des techniques traditionnelles d'analyse syntaxique, Perl en emploie une bien plus puissante : il utilise les blocs `BEGIN` pour guider toute compilation plus approfondie. Les blocs `BEGIN` sont passés à l'interpréteur pour être lancés dès qu'ils ont été analysés, ce qui les lance effectivement dans l'ordre du premier arrivé (*FIFO* : *first in, first out*). Ceci comprend toutes les déclarations `use` et `no` ; celles-ci ne sont que des blocs `BEGIN` déguisés. Tous les blocs `CHECK`, `INIT` et `END` sont programmés par le compilateur pour une exécution différée.

   Les déclarations lexicales sont notées mais leurs affectations ne sont pas exécutées. Toutes les constructions `eval` *`BLOC`*, `s///e` et les expressions régulières non interpolées sont compilées ici et les expressions constantes sont pré-évaluées. Le compilateur a maintenant fini, à moins qu'il ne soit rappelé au travail plus tard. À la fin de cette phase, l'interpréteur est à nouveau appelé pour exécuter tous les blocs `CHECK` programmés dans l'ordre du dernier arrivé (*LIFO* : *last in, first out*). La présence ou l'absence d'un bloc `CHECK` détermine si nous allons ensuite à la phase 2 ou si nous sautons directement à la phase 4.

2. *La phase de génération de code (optionnelle)*

   Les blocs `CHECK` sont installés par les générateurs de code ; cette phase optionnelle n'intervient donc que si vous avez utilisé explicitement l'un de ces générateurs de code (décrit plus tard dans *Générateurs de codes*). Ceux-ci convertissent le programme compilé (mais non encore lancé) soit en code source en C, soit en bytecode

Perl sérialisé — une séquence de valeurs exprimant des instructions Perl internes. Si vous choisissez de générer du code source en C, il peut éventuellement produire un fichier appelé une *image exécutable* en langage machine natif.[2]

Arrivé à ce point, votre programme se met en hibernation. Si vous avez fait une image exécutable, vous pouvez directement aller à la phase 4 ; sinon, vous devez reconstruire les bytecodes lyophilisés en phase 3.

3. *La phase de reconstruction de l'arbre syntaxique (optionnelle)*

   Pour réanimer le programme, son arbre syntaxique doit être reconstruit. Cette phase n'existe que si la génération de code a eu lieu et si vous avez choisi de générer du bytecode. Perl doit tout d'abord reconstituer ses arbres syntaxiques à partir de cette séquence de bytecode avant que le programme puisse être lancé. Perl ne se lance pas directement à partir des bytecodes ; ce serait trop lent.

4. *La phase d'exécution*

   Finalement, ce que vous attendiez tous : faire tourner votre programme.[3] L'interpréteur prend l'arbre syntaxique (qu'il obtient directement depuis le compilateur ou indirectement depuis la génération de code et la reconstruction consécutive de l'arbre) et l'exécute. (Ou, si vous avez généré un fichier image exécutable, il peut être lancé en tant que programme indépendant puisqu'il contient un interpréteur Perl intégré.)

   Au début de cette phase, avant que ce ne soit à votre programme principal de se lancer, tous les blocs `INIT` programmés sont exécutés dans l'ordre du premier arrivé (*FIFO*). Puis, votre programme principal est lancé. L'interpréteur peut faire à nouveau appel au compilateur comme cela est nécessaire lorsqu'il rencontre un `eval` *`CHAÎNE`*, un `do` *`FICHIER`* ou une instruction `require`, une construction `s///ee` ou une reconnaissance de motif avec une variable interpolée contenant une assertion de code valide.

   Lorsque votre programme principal a fini, tous les blocs `END` différés sont finalement exécutés, cette fois-ci dans l'ordre du dernier arrivé (*LIFO*). Le bloc vu en tout premier sera exécuté en dernier et ensuite vous aurez fini. (Les blocs `END` ne sont court-circuités que si vous appelez un `exec` ou si votre processus est balayé par une erreur catastrophique non interceptée. Les exceptions ordinaires ne sont pas considérées comme étant catastrophiques.)

Maintenant nous allons discuter de ces phases plus en détail et dans un ordre différent.

Compilation du code

Perl est toujours dans l'un des deux modes d'opération : soit il est en train de compiler votre programme, soit il est en train de l'exécuter — jamais les deux en même temps.

---

2. Votre script original est également un fichier exécutable mais il n'est pas en langage machine, nous ne l'appelons donc pas une image. Un fichier image est appelé ainsi car il s'agit d'une copie exacte des codes machine que votre CPU sait directement exécuter.

3. *N.d.T.* : en anglais, faire tourner se dit *run* et par conséquent, cette phase s'appelle également *run phase*.

Tout au long de ce livre, nous nous référons à certains événements comme se déroulant à la compilation (*compile time*) ou alors nous disons que « le compilateur Perl fait ceci ou cela ». À d'autres endroits, nous mentionnons qu'autre chose se déroule à l'exécution (*run time*) ou que « l'interpréteur Perl fait ceci ou cela ». Bien que vous pouvez continuer à vivre en pensant que « Perl » désigne à la fois le compilateur et l'interpréteur, comprendre lequel de ces deux rôles est en train de jouer Perl à un moment donné est essentiel pour comprendre pourquoi certaines choses se déroulent comme elles le font et non autrement. L'exécutable *perl* implémente les deux rôles : tout d'abord le compilateur, puis l'interpréteur. (D'autres rôles sont également possibles ; *perl* est aussi un optimisateur et un générateur de code. Parfois, c'est même un tricheur — mais tout cela dans la joie et la bonne humeur.)

Il est également important de comprendre la distinction entre la phase de compilation et la compilation (*compilation time*) ainsi qu'entre la phase d'exécution et l'exécution (*run time*). Une « phase » est un concept à grande échelle. Alors que la compilation et l'exécution sont des concepts à petite échelle. Une phase de compilation donnée fait surtout des travaux de compilation, mais elle fait également quelques travaux d'exécution *via* les blocs `BEGIN`. Une phase d'exécution donnée fait surtout des travaux d'exécution, mais elle fait également quelques travaux de compilation par l'intermédiaire d'opérateurs comme `eval` *`CHAÎNE`*.

Dans le déroulement typique des événements, le compilateur Perl lit le source de votre programme en entier avant que l'exécution ne commence. C'est ici que Perl analyse les déclarations, les instructions et les expressions pour s'assurer qu'elles sont syntaxiquement valides.[4] S'il trouve une erreur de syntaxe, le compilateur tente de la corriger pour qu'il puisse rendre compte des erreurs suivantes dans le code source. Parfois, ceci fonctionne et parfois non ; les erreurs de syntaxe ont une tendance fâcheuse à déclencher une cascade de fausses alertes. Perl abandonne, avec un sentiment de frustration, après environ 10 erreurs.

En plus de l'interpréteur qui traite les blocs `BEGIN`, le compilateur traite votre programme avec la connivence de trois agents notionnels. Ceux-ci sont parfois appelés « lexèmes » mais vous les rencontrerez plus souvent sous le nom de *tokens* dans les textes parlant des langages de programmation. L'analyseur lexical est parfois appelé un *tokener* ou un *scanner* et ce qu'il fait est parfois appelé de la lexicalisation (*lexing*) ou de la tokenisation (*tokening*). L'*analyseur syntaxique* (*parser*) essaie alors de dégager un sens des groupes de tokens en les assemblant dans des constructions plus larges, comme des expressions et des instructions, basées sur la grammaire du langage Perl. L'*optimisateur* réarrange et réduit ces regroupements plus larges en séquences plus efficaces. Il choisit ses optimisations précautionneusement, sans perdre de temps sur des optimisations marginales car le compilateur Perl doit être rapide comme l'éclair puisqu'il est utilisé en mode *load-and-go* (c'est-à-dire que l'exécution d'un programme est toujours précédée de sa compilation).

4. Non, il n'y a pas de diagramme de syntaxe formelle comme en syntaxe BNF (Backus-Naur Form), mais nous vous invitons à lire attentivement, dans l'arborescence des sources de Perl, le fichier *perly.y*, qui contient la grammaire *yacc*(1) que Perl utilise. Nous vous recommandons de vous tenir le plus loin possible de l'analyseur lexical, qui est connu pour avoir provoqué des troubles alimentaires chez des rats de laboratoire.

Ceci ne se produit pas dans des étapes indépendantes mais d'un seul coup avec énormément de dialogues croisés entre les agents. L'analyseur lexical a parfois besoin d'indications de la part de l'analyseur syntaxique pour lever les ambiguïtés sur les différents types possibles des tokens qu'il regarde. (Curieusement, la portée lexicale est l'une des choses que l'analyseur lexical ne comprend *pas* car c'est l'autre sens de « lexique ».) L'optimisateur a également besoin de garder une trace de ce que l'analyseur syntaxique est en train de faire car certaines optimisations ne peuvent être faites avant que l'analyseur syntaxique n'ait atteint un certain point, tel que la fin d'une expression, d'une instruction, d'un bloc ou d'un sous-programme.

Vous pouvez trouver étrange que le compilateur Perl fasse toutes ces choses en une fois plutôt que les unes après les autres mais il s'agit certainement du même processus confus que lorsque vous essayez de comprendre un langage naturel à la volée alors que vous êtes en train de l'écouter ou de le lire. Vous n'attendez pas la fin d'un chapitre pour comprendre ce que la première phrase voulait dire. Vous pourriez pensez aux correspondances suivantes :

Langage informatique	**Langage naturel**
Caractère	Lettre
Token	Morphème
Terme	Mot
Expression	Proposition
Instruction	Phrase
Bloc	Paragraphe
Fichier	Chapitre
Programme	Histoire

En supposant que l'analyse syntaxique se passe bien, le compilateur estime que ce que vous lui avez passé en entrée est une histoire valide — heu, pardon, un programme valide. Si vous utilisez l'option `-c` lorsque vous lancez votre programme, il affiche un message « syntax OK » et se termine. Sinon, le compilateur passe les fruits de ses efforts aux autres agents. Ces « fruits » se présentent sous forme d'un arbre syntaxique. Chaque fruit de l'arbre — ou chaque nœud, comme on les appelle — représente l'un des codes d'opération (*opcodes*) internes de Perl et les branches de l'arbre représentent le schéma d'évolution de cet arbre. Éventuellement, les nœuds seront enchaînés de manière linéaire, les uns après les autres, pour indiquer l'ordre dans lequel le système d'exécution devra les passer en revue.

Chaque code d'opération est la plus petite unité d'instruction exécutable que Perl peut appréhender. Vous pourriez rencontrer une expression telle que `$a = -($b + $c)` comme étant une seule instruction mais Perl l'appréhende comme six codes d'opération séparés. Présenté sous un format simplifié, l'arbre syntaxique pour cette expression ressemblerait à la *figure 18-2*. Les numéros représentent l'ordre de parcours que le système d'exécution de Perl suivra éventuellement.

Perl n'est pas un compilateur travaillant en une seule passe comme certains pourraient le penser. (Les compilateurs travaillant en une seule passe sont très doués pour rendre

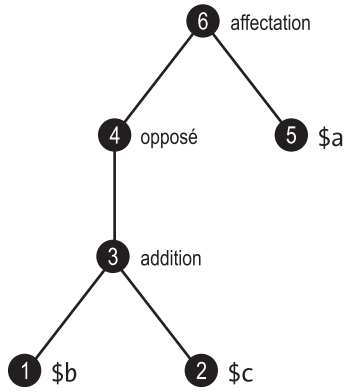


Figure 18-2. Ordre de parcours des codes d'opération de $a = -($b +$c)

les choses faciles à l'ordinateur et difficiles au programmeur.) Il s'agit réellement d'un compilateur multipasse, faisant des optimisations et consistant en au moins trois passes logiques différentes, qui dans la pratique s'entremêlent. Les passes 1 et 2 sont lancées en alternance alors que le compilateur fait des allers-retours à toute vitesse le long de l'arbre syntaxique durant sa construction ; la passe 3 se produit dès qu'un sous-programme ou qu'un fichier est entièrement analysé. Voici ces différentes passes :

Passe 1 : Analyse ascendante (Bottom-up parsing)

Durant cette passe, l'arbre syntaxique est construit par l'analyseur syntaxique *yacc*(1) en utilisant les tokens qu'on lui donne à partir de l'analyseur lexical sous-jacent (qui pourrait être considéré comme une autre phase logique à titre individuel). « Ascendante » signifie seulement que l'analyseur syntaxique connaît les feuilles de l'arbre avant ses branches et sa racine. Il calcule vraiment les choses de bas en haut dans la *figure 18-2*, puisque nous avons dessiné la racine en haut, dans la pure tradition idiosyncratique des informaticiens. (Et des linguistes.)

Alors que chaque nœud contenant un code d'opération est construit, une vérification de bon sens par code d'opération est effectuée pour contrôler si la sémantique est valide, comme le bon nombre et le type correct des arguments utilisés pour appeler les fonctions internes. Alors que chaque sous-section de l'arbre prend forme, l'optimisateur estime les transformations qu'il peut appliquer au sous-arbre entier se trouvant maintenant sous son emprise. Par exemple, une fois qu'il sait qu'une liste de valeurs est passée à une fonction prenant un certain nombre d'arguments, il peut se défaire du code d'opération qui enregistre le nombre d'arguments pour les fonctions en acceptant un nombre variable. Une optimisation plus importante, connue sous le nom de précalcul des expressions constantes (*constant folding*), est décrite plus loin dans ce paragraphe.

Cette passe construit également l'ordre de parcours des nœuds utilisé plus tard pour l'exécution, ce qui est une astuce vraiment géniale car le départ du parcours n'est presque jamais le nœud supérieur. Le compilateur réalise une boucle temporaire des codes d'opération, avec le nœud supérieur pointant sur le premier code d'opération à parcourir. Lorsque le nœud supérieur est incorporé dans quelque chose de plus gros, cette boucle de codes d'opération est brisée, uniquement pour

engendrer une boucle plus grosse avec le nouveau nœud supérieur. La boucle est éventuellement brisée pour de bon lorsque le nœud de départ est poussé dans une autre structure comme un descripteur de sous-programme. L'appelant du sous-programme peut toujours retrouver ce premier code d'opération bien qu'il soit dans les profondeurs de l'arbre, comme c'est le cas dans la *figure 18-2*. L'interpréteur n'a aucun besoin de revenir en bas de l'arbre pour arriver à comprendre par où commencer.

Passe 2 : Optimisateur descendant (Top-down optimizer)

Une personne lisant un extrait d'un code Perl (ou d'un code français, pour ce qui nous préoccupe ici) ne peut déterminer le contexte sans examiner les éléments lexicaux l'entourant. Vous ne pouvez parfois pas décider ce qui se passe réellement jusqu'à ce que vous ayez plus d'informations. Que cela ne vous afflige pas pour autant car vous n'êtes pas seul : le compilateur non plus. Dans cette passe, le compilateur redescend le sous-arbre qu'il vient de construire pour appliquer des optimisations locales, dont la plus remarquable est la *propagation de contexte*. Le compilateur marque les nœuds avec les contextes appropriés (vide, scalaire, liste, référence ou lvalue) imposés par le nœud courant. Les codes d'opération superflus sont annulés, mais non supprimés, car il est maintenant trop tard pour reconstruire l'ordre d'exécution. Nous compterons sur la troisième passe pour les enlever de l'ordre d'exécution provisoire déterminé par la première passe.

Passe 3 : Optimisateur à lucarne (Peephole optimizer)

Certaines unités de code ont leur propre espace de stockage dans lequel elles conservent leurs variables de portée lexicale. (De tels espace sont appelés mémoires de travail ou *scratchpads* dans le jargon de Perl.) Ces unités comprennent les `eval` *`CHAÎNE`*, les sous-programmes et les fichiers entiers. D'une manière importante du point de vue de l'optimisateur, elles ont chacune leur propre point d'entrée, ce qui signifie qu'alors que nous connaissons l'ordre d'exécution à partir d'ici, nous ne pouvons pas connaître ce qui s'est passé auparavant car la construction a pu être appelée de n'importe où. Ainsi, lorsque l'une de ces unités a fini d'être analysée, Perl lance un optimisateur regardant par une lucarne, sur ce code. Au contraire des deux passes précédentes, qui parcouraient la structure de branche de l'arbre syntaxique, cette passe-ci parcours le code dans un ordre d'exécution linéaire, puisqu'elle constitue à la base la dernière opportunité de le faire avant que nous isolions la liste de codes d'opération de l'analyseur syntaxique. La plupart des optimisations ont déjà été effectuées dans les deux premières passes mais certaines ne le pouvaient pas.

Des optimisations variées de dernière minute se produisent ici, y compris l'assemblage de l'ordre d'exécution final en court-circuitant les codes d'opération nuls et le fait de détecter lorsque diverses juxtapositions de codes d'opération peuvent être réduites à quelque chose de plus simple. L'identification d'une série de concaténations de chaînes est une optimisation importante puisque vous aimeriez vraiment éviter de copier la même chaîne encore et encore à chaque fois que vous ajoutez un petit quelque chose à la fin. Cette passe ne fait pas que de l'optimisation ; il réalise également une grande quantité de travail « réel » : l'interception de mots simples (*barewords*), la génération d'avertissements sur des constructions douteuses, la recherche de code peu susceptible d'être atteint, la résolution de clefs de pseudo-

hachages et la recherche de sous-programmes appelés avant que leurs prototypes n'aient été compilés.

Passe 4 : Génération de code (Code generation)

Cette passe est optionnelle ; elle n'est pas utilisée dans le déroulement normal des événements. Mais si l'un des trois générateurs de code — `B::Bytecode`, `B::C` ou `B::CC` — est invoqué, on accède une dernière fois à l'arbre syntaxique. Les générateurs de code émettent soit les bytecodes Perl sérialisés utilisés plus tard pour reconstruire l'arbre syntaxique, soit du code littéral C représentant l'état de l'arbre syntaxique à la compilation.

La génération de code C est disponible en deux variétés différentes. `B::C` reconstruit simplement l'arbre syntaxique et le lance en utilisant la boucle habituelle `runops()` que Perl lui-même utilise durant l'exécution. Tandis que `B:CC` produit un équivalent C linéarisé et optimisé du chemin emprunté par le code d'exécution (qui ressemble à une table de branchements (*jump table*) géante) et l'exécute.

Pendant la compilation, Perl optimise votre code de manières très, très, diverses. Il réarrange le code pour le rendre plus efficace à l'exécution. Il supprime le code qui ne pourra jamais être atteint durant l'exécution, comme un bloc `if (0)` ou les blocs `elsif` et `else` avec un bloc `if (1)`. Si vous utilisez des variables lexicalement typées, déclarées avec `my NomClasse $var` ou `our NomClasse $var` et que le paquetage NomClasse avait été initialisé avec le pragma `use fields`, les accès aux champs constants depuis le pseudo-hachage sous-jacent sont vérifiés pour y détecter des fautes de frappe et convertis en accès à un tableau. Si vous donnez à l'opérateur `sort` une routine de comparaison suffisamment simple, comme `{$a <=> $b}` ou `{$b cmp $a}`, celle-ci est remplacée par un appel à du code C compilé.

L'optimisation la plus impressionnante de Perl est probablement la manière dont il résout les expressions constantes dès que possible. Considérez par exemple l'arbre syntaxique présenté à la *figure 18-2*. Si les nœuds et 2 avaient tous deux été des littéraux ou des fonctions constantes, les nœuds 1 à 4 auraient été remplacés par le résultat de ce calcul, ce qui aurait donné quelque chose comme la *figure 18-3*.

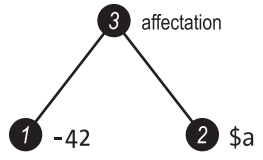


Figure 18-3. Précalcul de constantes

On appelle ceci le précalcul d'expressions constantes (*constant folding*). Ce précalcul ne se limite pas à des cas simples comme la conversion à la compilation de `2**10` en `1024`. Il résout également les appels de fonctions — à la fois les fonctions internes et les sous-programmes déclarés par l'utilisateurs remplissant les critères de la section *Inclure par référence les fonctions constantes* au chapitre 6, *Sous-programmes*. En réminiscence des compilateurs FORTRAN, célèbres pour leur connaissance de leurs propres fonctions intrinsèques, Perl sait également laquelle de ses fonctions internes appeler durant la compilation. C'est pourquoi lorsque vous essayez de prendre le `log` de `0.0` ou la racine

carrée (`sqrt`) d'une constante négative, vous encourez une erreur de compilation et non d'exécution, et l'interpréteur n'est pas lancé du tout.[5]

Même les expressions arbitrairement complexes sont résolues très tôt, en déclenchant parfois la suppression de blocs entiers comme celui-ci :

```
if (2 * sin(1)/cos(1) < 3 && une_fonc() { n_importe_quoi() }
```

Aucun code n'est généré pour ce qui ne pourra jamais être évalué. Comme la première partie est toujours fausse, ni `une_fonc`, ni `n_importe_quoi` ne seront jamais appelés. (Ne vous attendez donc pas à faire un `goto` sur des étiquettes à l'intérieur de ce bloc car il n'existera même pas à l'exécution.) Si `une_fonc` était une fonction constante susceptible d'être incluse par référence (*inlinable*), sa valeur aurait juste été insérée comme si c'était une constante littérale. Vous auriez alors encouru un avertissement « *Useless use of a constant in void context* » (Utilisation inutile d'une constante dans un contexte vide). Ceci pourrait vous surprendre si vous n'aviez pas réalisé que c'était une constante. Toutefois, si `n_importe_quoi` était la dernière instruction évaluée dans une fonction appelée dans un contexte non vide (tel que l'optimisateur le détermine), vous n'auriez jamais vu l'avertissement.

Vous pouvez voir le résultat final de l'arbre syntaxique construit après toutes les étapes d'optimisation avec *perl -Dx*. (L'option **`-D`** exige une version spéciale de Perl, permettant le débogage) Voir également le paragraphe sur `B::Deparse` décrit ci-dessous.

En fin de compte, le compilateur Perl travaille dur (mais pas *trop* dur) pour optimiser le code afin que, lorsqu'arrive le moment de l'exécution, celle-ci soit accélérée. C'est le moment où votre programme est lancé, alors occupons-nous en maintenant.

Exécution du code

À première vue, les programmes Sparc ne tournent que sur des machines Sparc, les programmes Intel ne tournent que sur des machines Intel et les programmes Perl ne tournent que sur des machines Perl. Une machine Perl possède des qualités qu'un programme Perl trouverait idéales dans un ordinateur : une mémoire qui s'alloue et se désalloue automatiquement, des types de données fondamentaux qui sont des chaînes, des tableaux et des hachages dynamiques et n'ont pas de limite de taille et des systèmes qui se comportent à peu près toujours de la même façon. Le travail de l'interpréteur Perl est de faire que n'importe quel ordinateur sur lequel il tourne, apparaisse comme l'une de ces machines Perl idéales.

Cette machine fictive présente l'illusion d'un ordinateur spécialement conçu pour ne rien faire tourner d'autre que des programmes Perl. Chaque code d'opération (*opcode*) produit par le compilateur est une commande fondamentale dans cet ensemble d'instructions émulées. Au lieu d'un compteur de programme matériel, l'interpréteur ne garde que la trace du code d'opération courant à exécuter. Au lieu d'un pointeur de pile matériel, l'interpréteur possède sa propre pile virtuelle. Cette pile est très importante

5. En fait, nous simplifions ici à l'extrême. L'interpréteur est bien lancé car c'est de cette manière que le précalcul de constantes est implémenté. Mais il est immédiatement lancé à la compilation, de la même façon que les blocs `BEGIN` sont exécutés.

car la machine virtuelle Perl (que nous refusons d'appeler une PVM[6]) est une machine reposant sur des opérations de pile. Les codes d'opération de Perl sont appelé en interne des *codes PP* (abréviation de codes « *push-pop* », « empiler-dépiler ») car ils manipulent la pile virtuelle de l'interpréteur pour trouver toutes les opérandes, traiter les valeurs temporaires et stocker tous les résultats.

Si vous avez jamais programmé en Forth ou en PostScript ou utilisé une calculatrice scientifique HP avec une saisie en RPN (« *Reverse Polish Notation* », notation polonaise inversée), vous savez comment fonctionne une machine basée sur des opérations de pile. Même si ce n'est pas le cas, le concept est simple : pour ajouter 3 et 4, vous faites les choses dans l'ordre `3 4 +` au lieu de l'ordre conventionnel `3 + 4`. Ce qui signifie en termes d'opérations de pile que vous empilez `3`, puis `4` et `+` dépile ensuite les deux arguments, les additionne et remet `7` dans la pile, où il restera jusqu'à ce que vous en fassiez quelque chose.

Comparé au compilateur Perl, l'interpréteur Perl est un programme rigoureux, presqu'ennuyeux. Tout ce qu'il fait est de parcourir pas à pas les codes d'opération compilés, un à un, et de les acheminer vers l'environnement d'exécution de Perl, c'est-à-dire, vers la machine virtuelle Perl. C'est juste un tas de code C, n'est-ce pas ?

En fait, ce n'est pas du tout ennuyeux. Une machine virtuelle Perl garde la trace d'une grande quantité de contextes dynamiques à votre place pour que vous n'ayez pas à le faire. Perl entretient un bon nombre de piles, que vous n'avez pas à connaître, mais que nous énumérerons ici, seulement pour vous épater :

pile d'opérandes (operand stack)

Il s'agit de la pile dont nous avons déjà parlé.

pile de sauvegardes (save stack)

Là où sont sauvegardées les valeurs déclarées locales en attendant d'être restaurées. Plusieurs routines internes déclarent également des valeurs comme étant locales sans que vous le sachiez.

pile de portées (scope stack)

Le contexte dynamique poids plume, qui contrôle le moment où la pile de sauvegardes doit être dépilée.

pile de contextes (context stack)

Le contexte dynamique poids lourd ; qui a appelé qui pour en arriver là où vous êtes. La fonction `caller` parcourt cette pile. Les fonctions de contrôle de boucle examinent cette pile pour trouver quelle boucle contrôler. Lorsque vous épluchez la pile de contexte, la pile de portées est épluchée en conséquence, ce qui restaure toutes vos variables locales à partir de la pile de sauvegardes, même si vous aviez quitté le contexte précédent par le biais de procédés infâmes comme de lever une exception et sortir avec un *longjmp*(3).

pile de changements d'environnement (jumpenv stack)

La pile des contextes de *longjmp*(3) qui nous permettent de lever des exceptions ou de quitter le programme de manière expéditive.

6. *N.d.T.* : PVM est l'acronyme habituel de *Parallel Virtual Machine*, machine parallèle virtuelle.

pile de retours (return stack)

Là d'où nous sommes partis lorsque nous sommes entrés dans ce sous-programme.

pile de repère (mark stack)

Là où commence, dans la pile d'opérandes, la liste d'arguments courante pour les fonctions avec un nombre variable d'arguments (*variadic*).

piles de mémoires de travail lexicales récursives (recursive lexical pad stacks)

Là où sont stockés les variables lexicales et les autres « registres de mémoires de travail » (*scratch register*) lorsque les sous-programmes sont appelés récursivement.

Bien entendu, il y a la pile C dans laquelle sont stockées toutes les variables C. Perl essaie en fait d'éviter de se reposer sur la pile du C pour le stockage de valeurs sauvegardées, puisque *longjmp*(3) court-circuite la restauration appropriée de telles valeurs.

Tout ceci pour dire que la vision habituelle que l'on a d'un interpréteur, un programme qui en interprète un autre, est vraiment inadéquate pour décrire ce qui se passe ici. Oui, il existe du code C implémentant des codes d'opération mais lorsque nous parlons d'« interpréteur », nous voulons dire quelque chose de plus que cela, de la même manière que lorsque nous parlons de « musicien », nous voulons dire quelque chose de plus qu'un ensemble d'instructions d'ADN changeant les notes en sons. Les musiciens sont des organismes réels, vivants et ont une « condition ». Il en va de même pour les interpréteurs.

En particulier, tout ce contexte dynamique et lexical, ainsi que les tables de symboles globales, plus l'arbre syntaxique, plus un thread d'exécution, est ce que nous appelons un interpréteur. En tant que contexte d'exécution, un interpréteur commence son existence avant même que le compilateur ne démarre et il peut se lancer dans une forme rudimentaire pendant même que le compilateur est en train de construire le contexte de l'interpréteur. En fait, c'est précisément ce qui arrive lorsque le compilateur appelle l'interpréteur pour exécuter des blocs `BEGIN` et d'autres choses dans le même genre. Et l'interpréteur peut se retourner et utiliser le compilateur pour se construire davantage. Chaque fois que vous définissez un autre sous-programme ou que vous chargez un autre module, la machine virtuelle Perl particulière que nous appelons un interpréteur se redéfinit elle-même. Vous ne pouvez pas vraiment dire si c'est le compilateur ou l'interpréteur qui a le contrôle car ils coopèrent pour contrôler le processus d'amorçage (*bootstrap process*) que nous appelons communément « lancer un script Perl ». C'est comme lorsqu'on amorce le cerveau d'un enfant. Est-ce l'ADN qui le fait ou les neurones ? Un peu des deux, selon nous, avec quelques interventions de programmeurs externes.

Il est possible de lancer de multiples interpréteurs dans le même processus ; ils peuvent ou non partager des arbres syntaxiques, selon qu'ils ont démarré en clônant un interpréteur existant ou en construisant un nouvel interpréteur à partir de zéro. Il est également possible de lancer de multiples threads dans un seul interpréteur, auquel cas ils ne partagent pas seulement des arbres syntaxiques mais également les symboles globaux — voir le chapitre 17, *Threads*.

Mais la plupart des programmes Perl n'utilisent qu'un seul interpréteur Perl pour exécuter leur code compilé. Et alors que vous pouvez lancer de multiples interpréteurs Perl

indépendants dans un seul processus, l'API actuelle pour ceci n'est accessible qu'en C.[7] Chaque interpréteur Perl individuel joue le rôle d'un processus complètement séparé mais ne coûte pas autant que la création d'un nouveau processus complet. C'est ainsi que l'extension `mod_perl` d'Apache obtient de telles performances : lorsque vous lancez un script CGI sous `mod_perl`, ce script a déjà été compilé en codes d'opération Perl, évitant ainsi le besoin de le recompiler — mais, ce qui est plus important encore, en évitant le besoin de démarrer un nouveau processus, ce qui constitue le véritable goulot d'étranglement. Apache initialise un nouvel interpréteur Perl dans un processus existant et passe à cet interpréteur le code précédemment compilé pour qu'il l'exécute. Bien entendu, c'est un peu plus compliqué que cela — c'est toujours un peu plus compliqué. Pour plus d'informations sur `mod_perl`, voir *Writing Apache Modules with Perl and C* (O'Reilly, 1999).

Plusieurs autres applications, comme *nvi*, *vim* et *innd*, peuvent intégrer des interpréteurs Perl ; nous ne pouvons pas espérer les lister toutes ici. Il existe un bon nombre de produits commerciaux qui ne font même pas la publicité pour le moteur Perl qu'ils ont intégré. Ils l'utilisent seulement en interne car il accomplit leur travail avec classe.

Sorties du compilateur

Ainsi, si Apache peut compiler un programme Perl maintenant et l'exécuter plus tard, pourquoi pas vous ? Apache et d'autres programmes contenant des interpréteurs Perl intégrés le font facilement — ils ne stockent jamais l'arbre syntaxique dans un fichier externe. Si cette approche vous convient et que vous cela ne vous embête pas d'utiliser l'API C pour en bénéficier, vous pouvez faire la même chose. Voir le paragraphe « Intégrer du Perl » au chapitre 21, *Mécanismes internes et accès externes* pour apprendre comment accéder à Perl depuis une infrastructure C l'entourant.

Si vous ne voulez pas prendre cette route ou si vous avez d'autres besoins, vous disposez alors de plusieurs options. Au lieu d'alimenter immédiatement l'interpréteur Perl avec les codes d'opération issus du compilateur Perl, vous pouvez invoquer l'une des nombreuses autres sorties (*backends*). Ces sorties peuvent sérialiser et stocker les codes d'opération compilés dans un fichier externe ou même les convertir en deux variétés différentes de code C.

Merci de bien prendre conscience que les générateurs de code sont tous des utilitaires extrêmement expérimentaux qui ne devraient pas espérer fonctionner dans un environnement de production. En fait, ils ne devraient même pas espérer fonctionner dans un environnement de non-production, sauf peut-être une fois tous les 36 du mois. Maintenant que nous avons assez rabaissé vos illusions pour que toute réussite les surpasse nécessairement, nous pouvons vous parler en toute sécurité du fonctionnement des sorties du compilateur.

Certains des modules de sortie (*backend*) sont des générateurs de code, comme `B::Bytecode`, `B::C` et `B::CC`. Les autres sont réellement des analyseurs de code et des outils de

7. Avec une exception jusqu'ici : la version 5.6.0 de Perl peut cloner des interpréteurs dans l'implémentation de l'émulation de `fork` sur Microsoft Windows. Il pourrait bien y avoir une API Perl pour les « ithreads » , comme on les appelle, au moment où vous lirez ceci.

débogage, comme `B::Deparse`, `B::Lint` et `B::Xref`. Au-delà de ces sorties, la distribution standard inclut plusieurs autres modules de bas niveau d'un intérêt potentiel pour ceux désirant devenir auteurs d'outils Perl de développement de code. D'autres modules de sortie peuvent être trouvés sur CPAN, comprenant (au moment ou ces lignes sont écrites) `B::Fathom`, `B::Graph`, `B::JVM::Jasmin` et `B::Size`.

Lorsque vous utilisez le compilateur Perl pour autre chose que l'alimentation de l'interpréteur, le module `O` (c'est-à-dire en utilisant le fichier `O.pm`) se tient entre le compilateur et les modules de sorties variés. Vous n'appelez pas les sorties directement : vous appelez plutôt la partie intermédiaire, qui à son tour appelle la sortie désignée. Ainsi, si vous aviez un module appelé `B::Backend`, vous l'invoqueriez dans un script donné de cette manière :

```
% perl -MO=Backend NOM_SCRIPT
```

Certaines sorties prennent des options, spécifiées comme ceci :

```
% perl -MO=Backend OPTS NOM_SCRIPT
```

Certaines sorties ont déjà leur propre frontal pour invoquer leurs parties intermédiaires pour vous afin que vous n'ayez pas à vous souvenir de leur M.O. En particulier, *perlcc*(1) invoque ce générateur de code, qui peut être encombrant à lancer.

Générateurs de code

Les trois sorties actuelles qui convertissent des codes d'opération Perl en un autre format sont toutes emphatiquement expérimentales. (Oui, nous l'avons déjà dit mais nous ne voulons pas que vous l'oubliiez.) Même lorsqu'il arrive qu'ils produisent en sortie quelque chose qui se lance correctement, les programmes résultants peuvent prendre plus d'espace disque, plus de mémoire et plus de temps CPU qu'ils ne le feraient d'ordinaire. Ceci est un sujet encore en recherche et développement. Les choses vont s'améliorer.

Le générateur de bytecode

Le module `B::Bytecode` écrit les codes d'opération de l'arbre syntaxique dans un encodage indépendant de la plate-forme. Vous pouvez prendre un script Perl compilé en bytecode et copier cela sur une autre machine où Perl est installé.

La commande standard, bien qu'actuellement expérimentale, *perlcc*(1) sait comment convertir du code source Perl en un programme Perl compilé en bytecode. Tout ce que vous devez faire est :

```
% perlcc -B script_src > script_pby
```

Et vous devriez maintenant être capable d'« exécuter » directement le *script_pby* résultant. Le début de ce fichier ressemble à quelque chose comme ceci :

```
#!/usr/bin/perl
use ByteLoader 0.04;
PLBCi386-linux^@0.04^@^D^@^@^@^D^@^@^@0x1234^@^G
^C^@^@^@^@^F^A^C^A^@^@^@^G^@^C^B^@^@^@^G^@^C^C^@^@^@^G^@^C^D^
^@^@^G^@^C^E^@^@^@^G^@^C^F^@^@^@^G^@^C^G^@^@^@^G^@^C^H^@^@^@^
...
```

Vous devriez trouver un petit en-tête de script suivit par des données purement binaires. Cela peut ressembler à de la pure magie mais son mystra, heu. . . mystère n'est pas très puissant. Le module `ByteLoader` utilise une technique appelée *filtre de source* pour altérer le code source avant que Perl n'ait une chance de le voir. Un filtre de source est un genre de préprocesseur s'appliquant à tout ce qui se trouve en dessous de lui dans le fichier courant. Au lieu d'être limité à des transformations simplistes comme les préprocesseurs de macros comme le sont *cpp*(1) et *m4*(1), il n'y a ici aucune limitation. Les filtres de source ont été utilisés pour enrichir la syntaxe de Perl, pour compresser ou crypter du code source et même pour écrire des programmes Perl en latin. E perlibus unicode; cogito, ergo substr; carp dbm, et al. Heu, avertissement du scriptor, désolé, du scribe latin.

Le module `ByteLoader` est un filtre de source qui sait comment désassembler les codes d'opération sérialisés pour reconstruire l'arbre syntaxique original. Le code Perl reconstitué est intégré dans l'arbre syntaxique actuel sans utiliser le compilateur. Lorsque l'interpréteur rencontre ces codes d'opération, il n'a qu'à les exécuter comme s'ils étaient ici depuis toujours, n'attendant que ça.

Les générateurs de code C

Les autres générateurs de codes, `B::C` et `B::CC`, produisent tous deux du code C au lieu de codes d'opération Perl sérialisés. Le code qu'ils génèrent est loin d'être lisible et si vous essayez de le lire, vous risquez de devenir aveugle. Ce n'est pas quelque chose que vous pouvez utiliser pour insérer quelques morceaux de code Perl traduit en C dans un programme C plus vaste. Pour faire cela, voir le chapitre 21.

Le module `B::C` ne fait qu'écrire les structures de données C nécessaires pour recréer l'environnement d'exécution Perl complet. Vous obtenez un interpréteur Perl dédié avec toutes les structures de données internes préinitialisées. En un certain sens, le code généré ressemble à celui que produit `B::Bytecode`. Les deux sont une traduction littérale des arbres de codes d'opération que construit le compilateur mais là où B::Bytecode les laisse sous forme symbolique pour être recréés plus tard et branchés dans un interpréteur Perl déjà lancé, `B::C` laisse ces codes d'opération en C. Lorsque vous compilez ce code C avec votre compilateur C et que vous faites l'édition de liens avec la bibliothèque Perl, le programme résultant n'aura pas besoin d'un interpréteur Perl installé sur le système cible. (Il pourra cependant avoir besoin de bibliothèques partagées, si vous n'avez pas fait une édition de liens statique pour tous les objets.) Cependant, ce programme n'est pas vraiment différent de l'interpréteur Perl normal qui lance votre script. Il est seulement pécompilé dans une image exécutable de manière isolée.

Le module `B::CC` essaie toutefois d'en faire plus. Le début du fichier source C qu'il génère ressemble un peu à ce que produit `B::C`[8] mais, éventuellement, toute ressemblance s'arrête là. Dans le code de `B::C`, vous avez une énorme table de codes d'opération en C qui est manipulée exactement de la même manière que l'interpréteur l'aurait fait tout seul, alors que dans le code C généré par `B::CC` est présenté dans l'ordre correspondant au flux d'exécution de votre programme. Il possède même une fonction C correspon-

8. Mais alors, comme c'est le cas pour tout, une fois que vous êtes devenu aveugle. Est-ce que nous ne vous avions pas prévenu de ne pas jeter de coup d'œil ?

dant à chaque fonction de votre programme. Une certaine quantité d'optimisations basées sur le type des variables est accomplie ; quelques benchmarks peuvent tourner deux fois plus vite que dans l'interpréteur standards. Ceci est le plus ambitieux des générateurs de code actuels, celui qui représente le plus grand espoir pour l'avenir. Et ce n'est pas une coïncidence si c'est également le moins stable des trois.

Les étudiants en informatique recherchant des projets de thèse n'ont pas besoin de chercher ailleurs. Il y a ici une montagne de diamants attendant d'être affinés.

Outils de développement de code

Le module `O` possède de nombreux modi operandi intéressants autres que l'alimentation des générateurs de codes désespérément expérimentaux. En fournissant un accès relativement indolore à la sortie du compilateur Perl, ce module facilite l'écriture d'autres outils ayant besoin de tout savoir à propos d'un programme Perl.

Le module `B::Lint` est ainsi baptisé d'après *lint*(1), le vérificateur de programmes C. Il inspecte les programmes à la recherche de constructions douteuses qui égarent souvent les débutants mais qui, normalement, ne déclenchent pas d'avertissements. Appelez le module directement :

```
% perl -MO=Lint,all mon_prog
```

Seules quelques vérifications sont actuellement définies, comme l'emploi d'un tableau dans un contexte scalaire implicite, le fait de se fier à des variables par défaut et l'accès à des identificateurs d'un autre paquetage, commençant par _ (normalement privés). Voir *B::Lint*(3) pour plus de détails.

Le module `B::Xref` génère des listes avec les références croisées des déclarations et des utilisations de toutes les variables (à la fois globales et de portée lexicale), de tous les sous-programmes et des tous les formats dans un programme, répartis par fichier et par sous-programme. Appelez le module de cette manière :

```
% perl -MO=Xref mon_prog > mon_prog.pxref
```

Voici, par exemple, un extrait de compte-rendu :

```
Subroutine analyse_argv
  Package (lexical)
    $allume           i113, 114
    $opt              i113, 114
    %config_getopt    i107, 113
    @args_config      i112, 114, 116, 116
  Package Getopt::Long
    $ignorecase       101
    &GetOptions       &124
  Package main
    $Options          123, 124, 141, 150, 165, 169
    %$Options         141, 150, 165, 169
    &verifie_lecture  &167
    @ARGV             121, 157, 157, 162, 166, 166
```

Ceci montre que la routine `analyse_argv` avait quatre variables lexicales propres ; elle accédait également à des identificateurs globaux des paquetages `main` et `Getopt::Long`.

Les nombres représentent les lignes où l'élément en question a été utilisé : un `i` au début, indique que cet élément est utilisé pour la première fois à ce numéro de ligne et un `&` au début signifie qu'un sous-programme a été appelé ici. Les déréférences sont listées séparément, c'est pourquoi figurent à la fois `$Options` et `%$Options`.

`B::Deparse` est un module offrant un affichage agréable (*pretty printer*) pouvant démystifier du code Perl et vous aidez à comprendre quelles transformations l'optimisateur a effectué sur votre code. Par exemple, ceci montre les valeurs utilisées par défaut par Perl pour diverses constructions :

```
% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = <ARGV>)) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}
```

L'option **`-p`** ajoute des parenthèses pour que vous puissiez voir l'idée que Perl se fait de la préséance :

```
% perl -MO=Deparse,-p -e 'print $a ** 3 + sqrt(2) / 10 ** -2 ** $c'
print((($a ** 3) + (1.4142135623731 / (10 ** (-(2 ** $c))))));
```

Vous pouvez utiliser **`-q`** pour voir en quelle opérations élémentaires les chaînes interpolées sont compilées :

```
% perl -MO=Deparse,-q -e '"Un $nom et quelques @ARGV\n"'
'Un ' . $nom . ' et quelques ' . join($", @ARGV) . "\n";
```

Et ceci montre comment Perl compile vraiment une boucle `for` avec trois arguments en une boucle `while` :

```
% perl -MO=Deparse,-x3 -e 'for ($i=0;$i<10;$i++) { $x++ }'
$i = 0;
while ($i < 10) {
    ++$x;
}
continue {
    ++$i
}
```

Vous pouvez même appeler `B::Deparse` sur un fichier de bytecode Perl produit par *perlcc -B* et `B::Deparse` le décompile alors pour vous. Cependant, lire des codes d'opération Perl sérialisés est peut-être un jeu d'enfant mais ce n'est pas le cas lorsqu'il s'agit d'un cryptage élevé.

Compilateur d'avant-garde, interpréteur rétro

Il y a un temps pour penser à tout ; parfois ce temps est derrière nous et parfois il est devant. Parfois il se situe entre les deux. Perl ne prétend pas savoir quand penser, il offre donc au programmeur un certain nombre d'options pour qu'il lui dise quand il faut penser. D'autres fois, il sait qu'une certaine sorte de pensée est nécessaire mais il n'a aucune idée de ce qu'il doit penser, il a donc besoin d'un moyen de le demander à votre programme. Votre programme répond à ce genre de questions en définissant des sous-programmes avec des noms appropriés à ce que Perl essaie de trouver.

Non seulement le compilateur peut appeler l'interpréteur lorsqu'il veut avancer dans sa pensée mais l'interpréteur peut également appeler le compilateur en retour lorsqu'il veut réviser l'histoire. Votre programme peut utiliser plusieurs opérateurs pour rappeler le compilateur. Comme le compilateur, l'interpréteur peut également appeler des sous-programmes nommés lorsqu'il veut se renseigner sur certaines choses. À cause de tous ces échanges entre le compilateur, l'interpréteur et votre programme, vous devez être conscient des choses qui se passent et quand elles se passent. Nous parlerons tout d'abord du moment où les sous-programmes nommés sont déclenchés.

Au chapitre 10, *Paquetages*, nous avons vu comment un sous-programme `AUTOLOAD` d'un paquetage était déclenché lorsqu'une fonction indéfinie dans ce paquetage était appelée. Au chapitre 12, *Objets*, nous avons fait connaissance avec la méthode `DESTROY` qui est invoquée lorsque la mémoire occupée par un objet est sur le point d'être récupérée par Perl. Enfin au chapitre 14, *Variables liées*, nous avons abordé plusieurs fonctions appelées implicitement lorsqu'on accédait à une variable liée.

Ces sous-programmes suivent tous la convention selon laquelle, si un sous-programme est déclenché automatiquement par le compilateur ou l'interpréteur, nous écrivons son nom en majuscules. Associés avec les différentes étapes de la vie de votre programme, il existe quatre autres sous-programmes de ce type, appelés `BEGIN`, `CHECK`, `INIT` et `END`. Le mot-clé `sub` est optionnel devant leurs déclarations. Peut-être devraient-ils être plutôt appelés « blocs » car d'une certaine manière ils ressemblent plus à des blocs nommés qu'à de véritables sous-programmes.

Par exemple, au contraire des sous-programmes ordinaires, il n'y a aucun mal à déclarer ces blocs plusieurs fois, puisque Perl garde une trace du moment où les appeler, vous n'avez donc pas à les appeler par leur nom. (Ils se différencient également des sous-programmes ordinaires en ce que `shift` et `pop` se comportent comme si vous étiez dans le programme principal et manipulent donc `@ARGV` par défaut et non `@_`.)

Ces quatre types de blocs s'exécutent dans cet ordre :

`BEGIN`

: S'exécute *ASAP* (*as soon as parsed*[9]) à chaque fois qu'on les rencontre pendant la compilation et avant de compiler le reste du fichier.

`CHECK`

: S'exécute lorsque la compilation est achevée mais avant que le programme ne démarre. (`CHECK` peut vouloir dire « *checkpoint* » (point de contrôle) ou « *double-check* » (double vérification) ou même simplement « *stop* ».)

`INIT`

: S'exécute au début de l'exécution, juste avant que le flux principal de votre programme ne commence.

`END`

: S'exécute à la fin de l'exécution, juste après que le programme a fini.

9. *N.d.T.* : dès que *parsés*, analysés, jeu de mot avec la signification habituelle d'ASAP : *as soon as possible*, dès que possible.

Si vous déclarez plus d'un de ces blocs avec le même nom, même dans des modules séparés, les BEGIN s'exécutent tous avant les CHECK, qui eux-mêmes s'exécutent tous avant les INIT, qui à leur tour s'exécutent tous avant les END — qui finalement sont les derniers à s'exécuter avant de mourir, après que votre programme a fini. De multiples BEGIN et INIT s'exécutent dans l'ordre dans lequel ils ont été déclarés (FIFO, *first in, first out* : premier arrivé, premier servi) et les CHECK et END s'exécutent dans l'ordre inverse dans lequel ils ont été déclarés (LIFO, *last in, first out* : dernier arrivé, premier servi).

Ceci est probablement plus clair avec une démonstration :

```
#!/usr/bin/perl -l
print     "démarre l'exécution principale ici\n";
die       "l'exécution principale meurt ici\n";
die       "XXX: jamais atteint\n";
END       { print "1er END: fini d'exécuter" }
CHECK     { print "1er CHECK: fini de compiler" }
INIT      { print "1er INIT: commence à exécuter" }
END       { print "2ème END: fini d'exécuter" }
BEGIN     { print "1er BEGIN: toujours en train de compiler" }
INIT      { print "2ème INIT: commence à exécuter" }
BEGIN     { print "2ème BEGIN: toujours en train de compiler" }
CHECK     { print "2ème CHECK: fini de compiler" }
END       { print "3ème END: fini d'exécuter" }
```

Lorsqu'il s'exécute, ce programme de démonstration affiche ceci :

```
1er BEGIN: toujours en train de compiler
2ème BEGIN: toujours en train de compiler
2ème CHECK: fini de compiler
1er CHECK: fini de compiler
1er INIT: commence à exécuter
2ème INIT: commence à exécuter
démarre l'exécution principale ici
l'exécution principale meurt ici
3ème END: fini d'exécuter
2ème END: fini d'exécuter
1er END: fini d'exécuter
```

Puisqu'un bloc BEGIN s'exécute immédiatement, il peut entrer dans les déclarations, les définitions et les importations de sous-programmes avant même que le reste du fichier ne soit compilé. Ceci peut altérer la manière dont le compilateur analyse le reste du fichier courant, tout particulièrement si vous importez des définitions de sous-programmes. Du moins, déclarer un sous-programme lui permet d'être utilisé comme un opérateur de liste, rendant les parenthèses optionnelles. Si le sous-programme importé est déclaré avec un prototype, les appels à ce sous-programme peuvent être analysés comme les fonctions internes et peuvent même supplanter les fonctions internes du même nom afin de leur donner une sémantique différente. La déclaration use n'est qu'un bloc BEGIN qui s'y croit.

Les blocs END, par contraste, s'exécutent aussi *tard* que possible : lorsque votre programme quitte l'interpréteur Perl, même si c'est le résultat d'un die non intercepté ou d'une autre exception fatale. Il existe deux situations dans lesquelles un bloc END (ou une méthode DESTROY) est court-circuité. Il ne s'exécute pas si, au lieu de se terminer, le proces-

sus courant se métamorphose d'un programme à un autre *via* `exec`. Un processus balayé de la surface de la terre par un signal non intercepté court-circuite également ses routines `END`. (Voir le pragma `use sigtrap` décrit au chapitre 31, *Modules de pragmas*, pour un moyen facile de convertir les signaux interceptables en exceptions. Pour des informations générales sur les signaux, voir la section *Signaux* au chapitre 16, *Communication inter-processus*.) Pour éviter le traitement de tous les `END`, vous pouvez appeler `POSIX::_exit` ou faire un `kill -9, $$` ou simplement `exec` de n'importe quel programme inoffensif, comme */bin/true* sur les systèmes Unix.

À l'intérieur d'un bloc `END`, `$?` contient le statut avec lequel le programme va sortir en faisant `exit`. Vous pouvez modifier `$?` depuis l'intérieur du bloc `END` pour changer la valeur de sortie du programme. Faites attention au changement accidentel de `$?` en lançant un autre programme avec `system` ou des apostrophes inverses.

Si vous avez plusieurs blocs `END` à l'intérieur d'un fichier, ils s'exécutent dans l'ordre *inverse* de leur définition. C'est-à-dire que le dernier bloc `END` défini est le premier à s'exécuter lorsque votre programme a fini. Cette inversion permet aux blocs `BEGIN` et `END` correspondants d'être imbriqués comme vous le souhaiteriez, si vous les aviez associés par paires. Par exemple, si le programme principal et un module qu'il charge ont leur propre paire de sous-programmes `BEGIN` et `END`, comme ceci :

```
BEGIN { print "le programme principal a commencé" }
END { print "le programme principal a fini" }
use Module;
```

et dans ce module, vous avez ces déclarations :

```
BEGIN { print "le module a commencé" }
END { print "le module a fini" }
```

alors le programme principal sait que son `BEGIN` arrivera toujours en premier et son `END` toujours en dernier. (Oui, `BEGIN` est vraiment un bloc à la compilation mais des arguments identiques s'appliquent à des paires de blocs `INIT` et `END` à l'exécution.) Ce principe est vrai récursivement pour tout fichier qui en inclut un autre lorsque les deux ont des déclarations comme celles-ci. Cette propriété d'imbrication implique que ces blocs fonctionnent bien en tant que constructeurs et destructeurs. Chaque module peut avoir ses propres fonctions d'initialisation et de démolition que Perl appellera automatiquement. De cette manière, le programmeur n'a pas à se se rappeler si une bibliothèque particulière est utilisée, ni quelle initialisation spéciale ou quel code de nettoyage doit être invoqué, ni quand l'invoquer. Les déclarations du module s'occupent de ces événements.

Si vous pensez à un `eval` *`CHAÎNE`* comme à un *r*appel (c'est-à-dire, un appel *en arrière*) du compilateur par l'interpréteur, alors vous pourriez penser à un `BEGIN` comme à un appel *en avant* de l'interpréteur par le compilateur. Les deux suspendent temporairement l'activité et basculent le mode d'opération. Lorsque nous disons qu'un bloc `BEGIN` s'exécute aussi tôt que possible, nous voulons dire qu'il s'exécute dès qu'il est complètement défini, avant même que le reste du fichier qui le contient ne soit analysé. Les blocs `BEGIN` sont par conséquent exécutés à la compilation, jamais à l'exécution. Une fois qu'un bloc `BEGIN` s'est exécuté, il devient immédiatement indéfini et le code qu'il a utilisé est renvoyé à l'espace mémoire de Perl. Vous ne pourriez pas appeler un bloc `BEGIN` comme un sous-programme même si vous essayiez car à l'heure où vous le feriez, il serait déjà parti.

De même que les blocs `BEGIN`, les blocs `INIT` s'exécutent juste avant que l'exécution de Perl ne commence, dans un ordre FIFO (*first in, first out* : premier arrivé, premier servi). Par exemple, les générateurs de codes documentés dans *perlcc* utilisent des blocs `INIT` pour initialiser et résoudre les pointeurs vers les XSUB. Les blocs `INIT` sont vraiment comme les blocs `BEGIN`, sauf qu'ils permettent au programmeur de distinguer les constructions qui se déroulent à la compilation de celles qui se produisent à l'exécution. Lorsque vous lancez un script directement, ceci n'est pas extrêmement important car le compilateur est de toute façon invoqué à chaque fois ; mais lorsque la compilation est séparée de l'exécution, la distinction peut devenir cruciale. Le compilateur peut n'être invoqué qu'une seule fois et l'exécutable résultant peut l'être plusieurs fois.

De même que les blocs `END`, les blocs `CHECK` s'exécutent dès que la phase de compilation de Perl se finit mais avant que la phase d'exécution ne commence, dans un ordre LIFO (*last in, first out* : dernier arrivé, premier servi). Les blocs `CHECK` sont utiles pour « libérer » le compilateur tout comme les blocs `END` sont utiles pour libérer votre programme. En particulier, les sorties (*backends*) utilisent toutes des blocs `CHECK` comme le point à partir duquel invoquer leurs générateurs de code respectifs. Tout ce qu'elles ont besoin de faire est de mettre un bloc `CHECK` dans leur propre module et il s'exécutera au moment approprié, vous n'avez donc pas à installer de `CHECK` dans votre programme. Pour cette raison, vous n'écrirez que très rarement un bloc `CHECK` vous-même, à moins que vous n'écriviez de tels modules.

Récapitulant tout ceci, le *tableau 18-1* énumère les diverses constructions en détaillant le moment où elles compilent le code représenté par « . . . » et celui où elles l'exécutent.

Tableau 18-1.

Bloc ou expression	Compile pendant la phase de	Capture les erreurs de compilation	Exécute pendant la phase de	Capture les erreurs d'exécution	Politique de déclenchement d'appel
`use ...`	C	Non	C	Non	Maintenant
`no ...`	C	Non	C	Non	Maintenant
`BEGIN {...}`	C	Non	C	Non	Maintenant
`CHECK {...}`	C	Non	C	Non	Tard
`INIT {...}`	C	Non	E	Non	Tôt
`END {...}`	C	Non	E	Non	Tard
`eval {...}`	C	Non	E	Oui	Inclus (*inline*)
`eval "..."`	E	Oui	E	Oui	Inclus (*inline*)
`truc(...)`	C	Non	E	Non	Inclus (*inline*)
`sub truc {...}`	C	Non	E	Non	Appel n'importe quand
`eval "sub {...}"`	E	Oui	E	Non	Appel plus tard
`s/motif/.../e`	C	Non	E	Non	Inclus (*inline*)
`s/motif/"..."/ee`	E	Oui	E	Oui	Inclus (*inline*)

Maintenant que vous connaissez la partition, nous espérons que vous serez capable de composer et de jouer vos morceaux de Perl avec plus d'assurance.

19

L'interface de la ligne de commande

Ce chapitre montre comment faire pointer Perl dans la bonne direction avant d'ouvrir le feu avec. Il existe diverses façon de faire viser Perl mais les deux moyens primaires sont par le biais d'options (*switches*) dans la ligne de commande et par le biais de variables d'environnement. Les options sont le moyen le plus immédiat et le plus précis d'ajuster une commande particulière. Les variables d'environnement sont plus souvent employées pour mettre en place la politique globale.

Traitement des commandes

Il est heureux que Perl se soit développé dans le monde Unix, car cela signifie que sa syntaxe d'appel fonctionne assez bien sous les interpréteurs de commandes des autres systèmes d'exploitation. La plupart des interpréteurs de commandes savent comment traiter une liste de mots en arguments, et ne voient pas d'inconvénient à ce qu'un argument commence par un signe moins. Bien entendu, il existe cependant quelques cas où vous vous faites piéger lorsque vous passez d'un système à l'autre. Vous ne pouvez pas utiliser les apostrophes sous MS-DOS comme vous le faites sous Unix, par exemple. Et sur les systèmes comme VMS, il est nécessaire d'employer un emballage logiciel (*wrapper*) sautant à travers divers cerceaux pour émuler l'indirection des entrées/sorties d'Unix. Nous posons notre joker quant à l'interprétation des jokers (*wildcards*). Mais une fois ces problèmes résolus, Perl traite ses options et ses arguments à peu près de la même manière sur n'importe quel système d'exploitation.

Même lorsque vous ne disposez pas d'un interpréteur de commandes *per se*, il est assez facile d'exécuter un programme Perl depuis un autre programme, écrit dans n'importe quel langage. Le programme appelant peut non seulement passer des arguments de la manière traditionnelle, mais il peut également passer des informations *via* des variables d'environnement et, si votre système d'exploitation l'implémente, des descripteurs de fichiers hérités (voir *Passage de handles de fichiers* au chapitre 16, *Communication inter-processus*). Même des mécanismes de passage d'argument plus exotiques peuvent facilement être encapsulés dans un module, amenés ensuite dans votre programme Perl *via* une simple directive `use`.

Perl analyse les options de la ligne de commande de la manière standard.[1] C'est-à-dire qu'il s'attend à ce que les options (les mots commençant par un moins) viennent en premier sur la ligne de commande. Le nom du script vient généralement ensuite, suivi par les arguments additionnels à passer au script. Certains de ces arguments supplémentaires peuvent eux-mêmes ressembler à des options, auquel cas ils doivent être traités par le script car Perl abandonne l'analyse des options dès qu'il voit une non-option, ou l'option spéciale « `--` » qui vaut dire « Je suis la dernière option ».

Perl vous procure une certaine souplesse en ce qui concerne ce que vous fournissez au programme. Pour les petites tâches rapides et brouillonnes, vous pouvez programmer Perl entièrement depuis la ligne de commande. Pour les travaux plus importants, plus permanents, vous pouvez fournir un script Perl dans un fichier séparé. Perl recherche à compiler et à lancer un script que vous pouvez spécifier avec l'une de ces trois manières :

1. Spécifié ligne par ligne *via* des options **`-e`** sur la ligne de commande. Par exemple :

```
% perl -e "print 'Salut, tout le monde.'"
Salut, tout le monde.
```

2. Contenu dans le fichier spécifié par le premier nom de fichier sur la ligne de commande. Les systèmes supportant la notation `#!` sur la première ligne d'un script exécutable invoquent les interpréteurs de cette façon à votre insu.
3. Passé implicitement *via* l'entrée standard. Cette méthode ne fonctionne que s'il n'y a pas de noms de fichier en argument ; pour passer des arguments à un script en entrée standard, vous devez utiliser la méthode 2, spécifier explicitement un « - » pour le nom du script. Par exemple :

```
% echo "print qq(Salut, @ARGV.)" | perl - tout le monde
Salut, tout le monde.
```

Avec les méthodes 2 et 3, Perl commence par analyser le fichier d'entrée depuis le début — à moins d'avoir spécifié l'option **`-x`**, ce qui lui fait rechercher la première ligne commençant par `#!` contenant le mot « `perl` », d'où il démarre. Cela sert à lancer un script inclus dans un message plus grand. En ce cas, vous pourriez indiquer la fin du script par le token `__END__`.

Que vous utilisiez ou non **`-x`**, la ligne `#!` est toujours examinée pour y trouver des options au moment de l'analyse syntaxique de la ligne. Ainsi, si vous vous trouvez sur une plate-forme qui n'autorise qu'un argument sur la ligne `#!` ou pis, qui ne reconnaît même pas la ligne `#!` comme étant spéciale, vous pouvez toujours conserver un comportement cohérent des options quelle que soit la manière dont Perl est invoqué, même si **`-x`** a été utilisé pour trouver le début du script.

Attention : comme les versions plus anciennes d'Unix interrompent sans rien dire l'interprétation par le noyau de la ligne `#!` après 32 caractères, certaines options peuvent traiter votre programme correctement ou non ; vous pourriez même obtenir un « - » sans sa lettre, si vous n'êtes pas prudent. Assurez-vous que tous vos scripts tombent avant ou après cette limite de 32 caractères. La plupart des options ne se soucient pas d'être traitées plusieurs fois mais un « - » au lieu d'une option complète pousserait Perl à essayer de lire son code source depuis l'entrée standard plutôt que depuis votre script. Et

1. En supposant que vous êtes d'accord avec le fait qu'Unix soit standard et maniéré.

une option **-I** tronquée pourrait également provoquer des résultats étranges. Toutefois, certaines options tiennent compte du fait qu'elles soient traitées deux fois, comme les combinaisons de **-l** et de **-0**. Mettez toutes les options après la limite de 32 caractères (le cas échéant) ou remplacez l'emploi de **-0***CHIFFRES* par `BEGIN{$/ = "\0`*CHIFFRES*`";}`. Évidemment, si vous n'êtes pas sur un système UNIX, vous êtes sûr de ne pas avoir ce genre de problème.

L'analyse des options sur la ligne `#!` démarre après que « `perl` » est mentionné pour la première fois. Les séquences « `-*` » et « `-` » sont spécifiquement ignorées au bénéfice des utilisateurs d'*emacs* et, si vous vous en sentez l'envie, vous pouvez écrire :

```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec perl $0 -S ${1+"$@"}'
    if 0;
```

et Perl ne verra que l'option **-p**. La jolie séquence « `-*- perl -*-` » indique à *emacs* de démarrer en mode Perl ; elle n'est pas nécessaire si vous n'utilisez pas *emacs*. Le sale **-S** est expliqué plus loin dans la description de cette option.

Une astuce identique met en jeu le programme *env*(1), si vous en disposez :

```
#!/usr/bin/env perl
```

Les exemples précédents utilisent un chemin relatif jusqu'à l'interpréteur Perl, obtenant ainsi la version trouvée en premier dans le chemin de l'utilisateur. Si vous voulez une version particulière de Perl, écrivez, *perl5.6.1* et placez ceci directement dans le chemin de la ligne `#!`, soit avec le programme *env*, soit avec le sale **-S**, soit avec un traitement ordinaire de `#!`.

Si la ligne `#!` ne contient *pas* le mot « `perl` », le programme cité après le `#!` est exécuté à la place de l'interpréteur Perl. Par exemple, supposons que vous ayez ici un script ordinaire pour le Bourne shell qui dirait :

```
#!/bin/sh
echo "Je suis un script shell"
```

Si vous donnez ce fichier à Perl, celui-ci lancera */bin/sh* pour vous. C'est un peu bizarre, mais cela facilite la vie à ceux dont les machines ne reconnaissent pas `#!`, parce qu'ils peuvent — en positionnant leur variable d'environnement `SHELL` — indiquer à un programme (comme un gestionnaire de courrier) que leur shell est */usr/bin/perl* et Perl enverra le programme à l'interpréteur qui leur convient, même si leur noyau est trop stupide pour le faire tout seul.

Mais revenons aux scripts Perl qui sont vraiment des scripts Perl. Après avoir localisé votre script, Perl compile le programme complet sous une forme interne (voir le chapitre 18, *Compilation*). S'il y a une erreur de compilation, l'exécution du script n'est même pas lancée. (Au contraire du script shell typique ou du fichier de commande, qui pourrait s'exécuter partiellement avant de trouver une erreur de syntaxe.) Si le script est syntaxiquement correct, il est exécuté. S'il arrive à la fin sans avoir rencontré un opérateur `exit` ou `die`, un `exit(0)` implicite est fourni par Perl pour indiquer le succès de l'opération à celui qui vous appelait. (Au contraire du programme C typique, où vous êtes susceptible d'obtenir un statut de sortie aléatoire si votre programme ne fait que se terminer de la manière normale.)

#! et isolement (quotation) sur les systèmes non-Unix

La technique `#!`, propre à Unix, peut être simulée sur d'autres systèmes :

Macintosh

Un programme Perl sur Macintosh aura la paire *Type/Creator* appropriée, de façon à ce qu'un double clic invoque l'application Perl.

MS-DOS

Créez un fichier batch pour lancer votre programme et codifiez-le dans `ALTERNATIVE_SHEBANG`. Voir le fichier *dosish.h* dans les sources de la distribution Perl pour plus d'informations à ce sujet.

OS/2

Ajoutez cette ligne :

```
extproc perl -S -vos_options
```

en tant que première ligne des fichiers **.cmd* (**`-S`** contourne un bogue dans la manière dont *cmd.exe* gère « `extproc` »).

VMS

Mettez ces lignes :

```
$ perl -mesopt 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7'
'p8' !
$ exit++ + ++$status != 0 and $exit = $status = undef;
```

au début de votre programme, où *`-mesopt`* représente toutes les options de ligne de commande que vous voulez passer à Perl. Vous pouvez désormais invoquer le programme directement en tapant `perl programme`, en tant que procédure DCL en disant `@programme` ou implicitement *via* `DCL$PATH` en utilisant juste le nom du programme. Cette incantation est un peu difficile à mémoriser mais Perl l'affichera pour vous si vous tapez `perl "-V:startperl"`. Si vous n'arrivez pas à vous souvenir de ça — hé bien, c'est la raison pour laquelle vous avez acheté ce livre.

Win??

Lorsque vous utilisez la distribution ActiveState de Perl, sous certaines variantes de la suite de systèmes d'exploitation Windows de Microsoft (c'est-à-dire, Win95, Win98, Win00,[2] WinNT, mais pas Win3.1), la procédure d'installation de Perl modifie la base de registre de Windows pour associer l'extension *.pl* avec l'interpréteur Perl.

Si vous installez un autre portage de Perl, y compris celui du répertoire Win32 de la distribution de Perl, vous devrez alors modifier vous-même la base de registre de Windows.

Remarquez que l'utilisation d'une extension *.pl* implique que vous ne pourrez plus faire la différence entre un programme exécutable Perl et un fichier de bibliothèque Perl (« *perl library* »). Vous pourriez plutôt utiliser *.plx* pour un programme Perl afin d'éviter ceci. Ce n'est plus très ennuyeux aujourd'hui que la plupart des modules Perl se trouvent dans des fichiers *.pm*.

2. Heu, veuillez excuser ces problèmes techniques...

Les interpréteurs de commandes les systèmes non-Unix ont des idées extraordinairement différentes des shells Unix concernant l'isolement (*quoting*). Il vous faudra apprendre quels sont les caractères spéciaux de votre interpréteur de commandes (on trouve couramment *, \ et ") et comment protéger les espaces et ces caractères spéciaux pour lancer des scripts sur une seule ligne (*one-liners*) *via* l'option **-e**. Vous pourriez également avoir à changer un seul % en %% ou le protéger avec une séquence d'échappement, s'il s'agit d'un caractère spécial pour votre shell.

Sur certains systèmes, vous devrez peut-être changer les apostrophes en guillemets. Mais ne faites pas cela sur les systèmes Unix ou Plan9 ou n'importe quel système où tourne un shell à la Unix, comme les systèmes du MKS Toolkit ou du paquetage Cygwin produit par les gens de chez Cygnus, appartenant maintenant à Redhat. Le nouvel émulateur d'Unix, Interix, produit par Microsoft, commence également à devenir, hum, hum, intérixant.

Par exemple, sur Unix et Mac OS X, utilisez :

```
% perl -e 'print "Salut tout le monde\n"'
```

Sur Macintosh (avant MAC OS X), utilisez :

```
print "Salut tout le monde\n"
```

puis lancez « Mon_script » ou Shift-Command-R.

Sur VMS, utilisez :

```
$ perl -e "print ""Salut tout le monde\n"""
```

ou encore, avec qq// :

```
$ perl -e "print qq(Salut tout le monde\n)"
```

Et sur MS-DOS, etc., utilisez :

```
A:> perl -e "print \"Salut tout le monde\n\""
```

ou utilisez qq// pour choisir vos propre caractères d'isolement :

```
A:> perl -e "print qq(Salut tout le monde\n)"
```

Le problème est qu'aucun de ces exemples n'est fiable : cela dépend de l'interpréteur de commandes que vous utilisez. Si le shell était *4DOS*, ceci fonctionnerait probablement mieux :

```
perl -e "print <Ctrl-x>"Salut tout le monde\n<Ctrl-x>""
```

Le programme *CMD.EXE* dans Windows NT semble avoir récupéré beaucoup de fonctionnalités du shell standard Unix lorsque tout le monde avait le dos tourné mais essayez de trouver de la documentation sur ses règles concernant l'isolement (*quoting rules*) !

Sur Macintosh,[3] tout ceci dépend de l'environnement que vous utilisez. MPW, qui peut être utilisé comme un shell pour MacPerl, est très proche des shells Unix pour son support de différentes variantes d'isolement, si ce n'est qu'il utilise allègrement les caractères non ASCII de Macintosh comme caractères de contrôle.

3. Du moins, sur les versions antérieures à Mac OS X, qui, de manière assez heureuse, est un système dérivé de BSD.

Il n'y a pas de solution générale à tout ceci. C'est tout bonnement une belle pagaille. Si vous n'êtes pas sur un système Unix mais que vous voulez faire des choses avec la ligne de commande, vous avez plus vite fait d'acquérir un meilleur interpréteur de commandes que celui que votre constructeur vous a fourni, ce qui ne devrait pas poser trop de difficultés.

Ou écrivez le tout en Perl et oubliez les scripts sur une seule ligne.

Emplacement de Perl

Bien que cela puisse paraître évident, Perl n'est utile que lorsque les utilisateurs peuvent facilement le trouver. Autant que possible, il est bon d'avoir à la fois */usr/bin/perl* et */usr/local/bin/perl* comme liens symboliques vers le véritable binaire. Si cela n'est pas possible, les administrateurs système sont fortement encouragés à mettre Perl et les utilitaires associés dans un répertoire généralement présent dans le `PATH` standard d'un utilisateur ou dans un quelconque autre endroit qui soit évident et pratique.

Dans ce livre, nous utilisons la notation standard `#!/usr/bin/perl` sur la première ligne d'un programme, pour désigner le mécanisme fonctionnant sur votre système, quel qu'il soit. Si vous avez besoin de lancer une version particulière de Perl, utilisez un chemin spécifique :

```
#!/usr/local/bin/perl5.6.0
```

Si vous voulez juste utiliser une version *au moins* égale à un certain numéro de version mais que vous vous fichez des versions supérieures, placez une instruction telle que celle-ci vers le début de votre programme :

```
use v5.6.0;
```

(Remarque : les versions plus anciennes de Perl utilisaient des numéros comme 5.005 ou 5.004_05. De nos jours, nous les verrions plutôt comme 5.5.0 et 5.4.5 mais les versions de Perl antérieures à 5.6.0 ne comprendraient pas cette notation.)

Options

Une *option* de la ligne de commande ne comportant qu'un seul caractère sans son argument peut toujours être combinée (reliée) avec l'option suivante.

```
#!/usr/bin/perl -spi.bak # comme -s -p -i.bak
```

Les options sont également appelées des *flags* ou des drapeaux ou des *switchs* (au singulier *switch*). Peu importe le nom que vous leur donnez, voici celles que Perl reconnaît :

--

Termine le traitement des options, même si l'argument suivant commence par un moins. Il n'a aucun autre effet.

-o*NUM_OCT*
-o

Spécifie le séparateur d'enregistrements (`$/`) comme étant un nombre octal. Si *NUM_OCT* n'est pas présent, le séparateur est le caractère nul (c'est-à-dire le caractère 0, soit `"\0"` en Perl). D'autres options peuvent précéder ou suivre le nombre octal.

Par exemple, si vous disposez d'une version de *find*(1) pouvant afficher les noms de fichier terminés par le caractère nul, vous pouvez écrire ceci :

```
% find . -name '*.bak' -print0 | perl -n0e unlink
```

La valeur spéciale `00` fait que Perl lira les fichiers en mode paragraphe, ce qui est équivalent à positionner la variable `$/` à `""`. La valeur `0777` fait que Perl gobera des fichiers entiers d'un seul coup. Cela revient à indéfinir la variable `$/`. Nous employons `0777` puisqu'il n'existe pas de caractère ASCII de cette valeur. (Malheureusement, il *existe* un caractère Unicode de cette valeur, `\N{LATIN SMALL LETTER O WITH STROKE AND ACUTE}`, mais quelque chose nous dit que vous ne l'utiliserez pas comme séparateur pour vos enregistrements.)

`-a`

Active le mode *autosplit* mais seulement lorsque cette option est utilisée avec un **`-n`** ou un **`-p`**. Une commande `split` implicite est d'abord exécutée sur le tableau `@F` dans la boucle `while` implicite produite par le **`-n`** ou le **`-p`**. Ainsi :

```
% perl -ane 'print pop(@F), "\n";'
```

est équivalent à :

```
while (<>) {
    @F = split(' ');
    print pop(@F), "\n";
}
```

Un séparateur de champ différent peut être spécifié en passant une expression régulière pour `split` à l'option **`-F`**. Par exemple, ces deux appels sont équivalents :

```
% awk -F: '$7 && $7 !~ /^\/bin/' /etc/passwd
% perl -F: -lane 'print if $F[6] && $F[6] !~ m(^/bin)' /etc/passwd
```

`-c`

Fait vérifier par Perl la syntaxe du script avant de sortir sans exécuter ce qu'il vient de compiler. Techniquement, Perl en fera un peu plus que ça : il exécutera tous les blocs `BEGIN` ou `CHECK` et toutes les directives `use`, puisque ceux-ci sont censés se produire avant l'exécution du programme. Cependant, les blocs `END` ou `INIT` ne sont plus exécutés. L'ancien comportement, toutefois rarement utile, peut toujours être obtenu en mettant :

```
BEGIN { $^C = 0; exit; }
```

à la fin de votre script principal.

`-C`

Active l'utilisation par Perl des API de caractères larges (*wide-character*) sur le système cible, si elles sont implémentées (avec la version 5.6.0, cela ne fonctionne que sur les plates-formes Microsoft). La variable spéciale `${^WIDE_SYSTEM_CALLS}` reflète l'état de cette option.

`-d`

Lance le script sous le débogueur Perl. Voir le chapitre 20, *Le débogueur Perl*.

`-d:`*`MODULE`*

Lance le script sous le contrôle d'un module de débogage ou de traçage installé dans la bibliothèque Perl sous le nom de `Devel::`*`MODULE`*. Par exemple, **`-d:DProf`**

exécute le script avec le profileur `Devel::DProf`. Voir également le paragraphe sur le débogage au chapitre 20.

-D*LETTRES*
-D*NOMBRE*

Active les drapeaux de débogage. (Ceci ne fonctionne que si le débogage est compilé dans votre Perl, comme décrit ci-dessous.) Vous pouvez spécifier soit un *NOMBRE* valant la somme des bits désirés, soit une liste de *LETTRES*. Pour voir comment il exécute votre script, employez, par exemple, **-D14** ou **-Dslt**. Une autre valeur utile est **-D1024** ou **-Dx**, qui liste l'arbre syntaxique compilé. Et **-D512** ou **-Dr** affiche les expressions régulières compilées. La valeur numérique est disponible en interne grâce à la variable spéciale `$^D`. Le tableau 19-1 liste les valeurs de bits assignées :

Tableau 19-1. Options de -D

Bit	Lettre	Signification
1	p	Mise en tokens et analyse syntaxique.
2	s	Instantanés de la pile.
4	l	Traitement de la pile d'étiquettes.
8	t	Trace de l'exécution.
16	o	Résolution des méthodes et des surcharges.
32	c	Conversions chaînes/numérique.
64	p	Affichage des commandes de préprocesseur pour **-P**.
128	m	Allocation mémoire.
256	f	Traitements des formats.
512	r	Analyse et exécution d'expressions régulières.
1024	x	Vidage mémoire (*dump*) de l'arbre syntaxique.
2048	u	Vérification du marquage.
4096	L	Fuites de mémoire (nécessite de compiler Perl avec **-DLEAKTEST**).
8192	H	Vidage mémoire (*dump*) de hachage — usurpe `values()`.
16384	X	Allocation de mémoires de travail (*scratchpad*).
32768	D	Nettoyage.
65536	S	Synchronisation de threads.

Tous ces drapeaux exigent un exécutable Perl ayant été spécialement compilé pour le débogage. Toutefois, comme ce n'est pas le cas par défaut, vous serez incapable d'utiliser l'option **-D** sauf si votre administrateur système ou vous-même avez compilé cette version spéciale de Perl pour le débogage. Voir le fichier *INSTALL* dans le répertoire des sources de Perl pour plus de détails mais, pour résumer, vous devez passer **-DDEBUGGING** à votre compilateur C lorsque vous compilez Perl lui-même. Ce drapeau est automatiquement positionné si vous incluez l'option **-g** lorsque *Configure* vous pose des questions à propos des drapeaux de l'optimisateur et du débogueur.

Si vous essayez seulement d'obtenir une impression de chaque ligne de code Perl au fur et à mesure qu'il s'exécute (de la même manière que *sh -x* le fait pour les

scripts shell), vous ne ne pouvez pas utiliser l'option **-D** de Perl. À la place, faites ceci :

```
# Syntaxe du Bourne shell
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS programme

# Syntaxe de csh
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS
programme)
```

Voir le chapitre 20 pour les détails et d'autres possibilités.

-e *CODE_PERL*

Peut être utilisé pour entrer une ou plusieurs lignes de script. Si **-e** est utilisé, Perl ne cherchera pas le nom du programme dans la liste d'arguments. L'argument *CODE_PERL* est traité comme s'il se terminait par un saut de ligne, ainsi, plusieurs commandes **-e** peuvent être données pour former un programme de plusieurs lignes. (Assurez-vous de mettre des points-virgules comme vous le feriez dans un programme normal stocké dans un fichier.) Ce n'est pas parce que **-e** fournit un saut de ligne à chaque argument que vous devez utiliser plusieurs options **-e** ; si votre shell autorise l'isolement (*quoting*) sur plusieurs lignes, comme *sh*, *ksh* ou *bash*, vous pouvez alors passer un script sur plusieurs lignes comme un seul argument de **-e** :

```
$ perl -e 'print "Salut, ";
           print "@ARGV !\n";' la compagnie
Salut, la compagnie !
```

Avec *csh*, il vaut probablement mieux utiliser plusieurs options **-e** :

```
% perl -e 'print "Salut, ";' \
       -e 'print "@ARGV !\n";' la compagnie
Salut, la compagnie !
```

Les sauts de ligne, qu'ils soient implicites ou explicites, sont pris en compte dans la numérotation des lignes, ainsi, le second `print` se trouve à la ligne 2 du script de **-e**, dans les deux cas.

-F*MOTIF*

Spécifie le motif selon lequel s'effectue l'éclatement (*split*) lorsque le mode *autosplit* est activé *via* l'option **-a** (sinon, n'a aucun effet). Le motif peut être entouré par des slash (`//`), des guillemets (`""`) ou des apostrophes (`''`). Si ce n'est pas le cas, il sera automatiquement mis entre apostrophes. Souvenez-vous que pour passer des apostrophes ou des guillemets d'isolement *via* un shell, vous devrez isoler ces apostrophes et la manière dont vous pouvez le faire dépend du shell.

-h

Affiche un résumé des options de ligne de commande de Perl.

-i*EXTENSION*
-i

Spécifie que les fichiers traités par la construction `<>` doivent être édités sur place. Ceci est réalisé en renommant le fichier d'entrée, en ouvrant le fichier de sortie d'après le nom d'origine et en sélectionnant ce fichier de sortie comme fichier par

défaut pour les appels à `print`, `printf` et `write`.[4]

L'*`EXTENSION`* est utilisée pour modifier le nom de l'ancien fichier afin d'en faire une copie de sauvegarde. Si aucune *`EXTENSION`* n'est fournie, aucune sauvegarde n'est effectuée et le fichier actuel est écrasé. Si l'*`EXTENSION`* ne contient pas de *, la chaîne est alors ajoutée à la fin du nom de fichier actuel. Si l'*`EXTENSION`* contient un ou plusieurs caractères *, chaque * est alors remplacé par le nom du fichier actuellement en cours de traitement. En termes de Perl, vous pourriez penser à ceci comme :

```
($sauvegarde = $extension) =~ s/\*/$nom_fichier/g;
```

Ceci vous permet d'utiliser un préfixe pour le fichier de sauvegarde, au lieu — ou même en complément — d'un suffixe :

```
% perl -pi'orig_*' -e 's/truc/machin/' xyz # sauvegarde dans 'orig_xyz'
```

Vous pouvez même mettre les copies de sauvegarde des fichiers originaux dans un autre répertoire (à condition que le répertoire existe déjà) :

```
% perl -pi'ancien/*.orig' -e 's/truc/machin/' xyz # sauvegarde dans
'ancien/xyz.orig'
```

Les scripts sur une seule ligne sont équivalents dans chaque paire :

```
% perl -pi -e 's/truc/machin/' xyz          # écrase le fichier actuel
% perl -pi'*' -e 's/truc/machin/' xyz       # écrase le fichier actuel

% perl -pi'.orig' -e 's/truc/machin/' xyz   # sauvegarde dans 'xyz.orig'
% perl -pi'*.orig' -e 's/truc/machin/' xyz  # sauvegarde dans 'xyz.orig'
```

Depuis le shell, écrire :

```
% perl -p -i.orig -e "s/truc/machin/;"
```

revient au même que d'utiliser le programme :

```
#!/usr/bin/perl -pi.orig s/truc/machin/;
```

Ce qui est un raccourci pratique pour le programme éminemment plus long :

```
#!/usr/bin/perl
$extension = '.orig';
LIGNE: while (<>) {
    if ($ARGV ne $ancien_argv) {
        if ($extension !~ /\*/) {
            $sauvegarde = $ARGV . $extension;
        }
        else {
            ($sauvegarde = $extension) =~ s/\*/$ARGV/g;
        }
        unless (rename($ARGV, $sauvegarde)) {
            warn "impossible de renommer $ARGV en $sauvegarde : $!\n";
            close ARGV;
            next;
        }
        open(ARGV_SORTIE, ">$ARGV");
```

4. Techniquement, il ne s'agit pas vraiment d'un édition « sur place ». Il s'agit du même nom de fichier, mais d'un fichier physique différent.

```
            select(ARGV_SORTIE);
            $ancien_argv = $ARGV;
        }
        s/truc/machin/;
    }
    continue {
        print; # ceci écrit dans le fichier de nom original
    }
    select(STDOUT);
```

Ce code un peu long est virtuellement identique au simple code sur une seule ligne avec l'option **-i**, hormis le fait que la forme **-i** n'a pas besoin de comparer `$ARGV` à `$ancien_argv` pour savoir quand le nom de fichier a changé. Mais elle utilise bien `ARGV_SORTIE` pour le handle de fichier sélectionné et restaure, après la boucle, l'ancien `STDOUT` comme handle de fichier de sortie par défaut. Comme le code ci-dessus, Perl crée le fichier de sauvegarde sans tenir compte du fait que la sortie ait réellement changé ou non. Voir la description de la fonction `eof` sans parenthèses pour des exemples sur la manière de détecter la fin de chaque fichier d'entrée, au cas où vous voudriez ajouter des lignes à chaque fichier ou remettre à zéro la numérotation des lignes.

Si, pour un fichier donné, Perl est incapable de créer le fichier de sauvegarde comme l'*EXTENSION* le spécifie, il émettra un avertissement à cet effet et continuera à traiter les autres fichiers.

Vous ne pouvez pas utiliser **-i** pour créer des répertoires ou pour enlever des extensions à des fichiers. Vous ne pouvez pas non plus l'utiliser avec un `~` indiquant le répertoire maison (*home directory*) — ce qui est d'autant mieux car certaines personnes aiment utiliser ce caractère pour leurs fichiers de sauvegarde :

```
% perl -pi~ -e 's/truc/machin/' fichier1 fichier2 fichier3...
```

Enfin, l'option `-i` n'arrête pas l'exécution de Perl si aucun nom de fichier n'est donné sur la ligne de commande. Lorsque cela arrive, aucune sauvegarde n'est effectuée puisqu'on ne peut pas déterminer quel est le fichier original et le traitement s'applique de `STDIN` vers `STDOUT` comme on pouvait s'y attendre.

-I*REPERTOIRE*

Les répertoires spécifiés par **-I** sont ajoutés au début de `@INC`, qui contient le chemin de recherche des modules. **-I** dit aussi au préprocesseur où il faut chercher les fichiers d'inclusion. Le préprocesseur C est invoqué par **-P** ; il recherche par défaut dans */usr/include* et */usr/lib/perl*. À moins que vous ne soyez sur le point d'utiliser le préprocesseur C (ce que quasiment plus personne ne fait), vous feriez mieux d'utiliser la directive `use lib` dans votre script. Cependant, comme `use lib`, l'option **-I** ajoute implicitement les répertoires spécifiques à la plate-forme. Voir `use lib` au chapitre 31, *Modules de pragmas* pour plus de détails.

-l*NUM_OCT*
-l

Active le traitement de fin de ligne automatique. Son effet est double : premièrement, il fait automatiquement un `chomp` sur la terminaison de ligne quand il est employé avec **-n** ou **-p** et deuxièmement, il positionne `$\` à la valeur de *NUM_OCT* afin que toute instruction d'affichage ajoute un terminateur de ligne de valeur

ASCII *NUM_OCT*. Si *NUM_OCT* est omis, **-l** positionne `$\` à la valeur actuelle de `$/`, généralement un saut de ligne. Par exemple, pour couper les lignes à quatre-vingt colonnes, écrivez ceci :

```
% perl -lpe 'substr($_, 80) = ""'
```

Remarquez que l'affectation `$\ = $/` est faite lorsque l'option est traitée, le séparateur d'enregistrements d'entrée peut donc être différent de celui de sortie si l'option **-l** est suivie d'une option **-0** :

```
% gnufind / -print0 | perl -ln0e 'print "trouvé $_" if -p'
```

Cela positionne `$\` à la valeur saut de ligne, puis `$/` au caractère nul. (Remarquez que `0` aurait été interprété comme faisant partie de l'option **-l** s'il avait suivi immédiatement le **-l**. C'est pourquoi nous avons inséré l'option **-n** entre les deux.)

-m et **-M**

Ces options chargent un *MODULE* comme si vous aviez exécuté un `use`, à moins que vous n'ayez spécifié *-MODULE* au lieu de *MODULE*, auquel cas `no` est invoqué. Par exemple, **-Mstrict** est identique à `use strict`, alors que **-M-strict** est identique à `no strict`.

-m*MODULE*

Exécute `use` *MODULE*`()` avant d'exécuter votre script.

-M*MODULE*
-M'*MODULE***...'**

Exécute `use` *MODULE* avant d'exécuter votre script. La commande est formée par simple interpolation du reste de l'argument suivant le **-M**, vous pouvez donc isoler l'argument par des apostrophes pour ajouter du code supplémentaire après le nom du module, par exemple, **-M'module qw(truc machin)'**.

-M*MODULE*=*arg1*,*arg2***...**

Un peu de sucre de syntaxe vous permet d'écrire également **-Mmodule=truc,machin** comme raccourci pour **-M'module qw(truc machin)'**. Cette possibilité dispense des apostrophes lorsqu'on importe des symboles. Le code réellement généré par **-Mmodule=truc,machin** est :

```
use module split(/,/, q{truc,machin})
```

Remarquez que la forme avec = élimine la distinction entre **-m** et **-M**, mais il vaut mieux utiliser la forme en majuscules pour éviter toute confusion.

Vous ne pouvez utiliser les options **-M** et **-m** que depuis une véritable invocation de Perl depuis la ligne de commande, non comme des options trouvées sur la ligne `#!`. (Hé, si vous alliez le mettre dans un fichier, pourquoi ne pas simplement écrire à la place le `use` ou le `no` équivalent?)

-n

Pousse Perl à se comporter comme si la boucle suivante encadrait le script, ce qui le fait itérer sur les noms de fichiers en argument à la manière de *sed -n* ou de *awk* :

```
LINE:
while (<>) {
    ...           # le script vient ici
}
```

Vous pouvez utiliser `LINE` (*`LIGNE`* en anglais) comme une étiquette de boucle depuis l'intérieur de votre script, même si vous ne pouvez pas voir la véritable étiquette dans votre fichier.

Remarquez que les lignes ne sont pas affichées par défaut. Voir **`-p`** pour ce faire. Voici un moyen efficace de détruire tous les fichiers datant de plus d'une semaine :

```
find . -mtime +7 -print | perl -nle 'unlink;'
```

Cette opération est plus rapide que l'option *-exec* de *find*(1) car vous n'avez pas à lancer un processus à chaque nom de fichier trouvé. Par une coïncidence stupéfiante, les blocs `BEGIN` et `END` peuvent être utilisés pour prendre le contrôle avant ou après la boucle implicite, exactement comme dans *awk*.

`-p`

Pousse Perl à se comporter comme si la boucle suivante encadrait le script, ce qui le fait itérer sur les noms de fichier en argument à la manière de *sed* :

```
LINE:
while (<>) {
    ...           # le script vient ici
}
continue {
    print or die "-p destination: $!\n";
}
```

Vous pouvez utiliser `LINE` (*`LIGNE`* en anglais) comme étiquette de boucle depuis l'intérieur de votre script, même si vous ne pouvez pas voir la véritable étiquette dans votre fichier.

Si un fichier désigné par un argument ne peut pas être ouvert pour une raison ou pour une autre, Perl vous en avertit et passe au fichier suivant. Remarquez que les lignes sont affichées automatiquement. Toujours par une coïncidence stupéfiante, les blocs `BEGIN` et `END` peuvent être utilisés pour prendre le contrôle avant ou après la boucle implicite, exactement comme dans *awk*.

`-P`

Pousse votre script à passer par le préprocesseur C avant la compilation par Perl. (Comme les commentaires et les directives de *cpp*(1) commencent tous par le caractère `#`, il vaut mieux éviter de commencer les commentaires par des mots-clés du préprocesseur C comme « `if` », « `else` » ou « `define` ».) Que vous utilisiez l'option **`-P`** ou non, Perl accorde toujours de l'attention aux directives `#line` pour contrôler le numéro de ligne et le nom de fichier, n'importe quel préprocesseur peut donc informer Perl sur de telles choses. Voir la section *Générer du Perl dans d'autres langages* au chapitre 24, *Techniques couramment employées*.

`-s`

Permet une analyse rudimentaire de la ligne de commande afin de rechercher les options suivant le nom du script, mais précédant tout nom de fichier ou une terminaison d'options **`--`**. Toute option trouvée est éliminée de `@ARGV` et une variable du même nom que l'option est positionnée dans Perl. Aucun accolement d'options n'est permis car les options de plusieurs caractères sont autorisées.

Le script suivant affiche « `vrai` » si, et seulement si, le script est invoqué avec une option `-truc`.

```
#!/usr/bin/perl -s
if ($truc) { print "vrai\n"; }
```

Si l'option est de la forme **-xxx=yyy**, la variable $xxx est positionnée à ce qui suit ce signe égal dans l'argument (« yyy » en l'occurrence). Le script suivant affiche « `vrai` » si et seulement si le script est invoqué avec une option **-truc=machin**.

```
#!/usr/bin/perl -s
if ($truc eq 'machin') { print "vrai\n"; }
```

-S

Pousse Perl à utiliser la variable d'environnement `PATH` pour rechercher le script (à moins que le nom du script ne contienne des séparateurs de répertoires).

Généralement, cette option est utilisée pour émuler le démarrage par `#!` sur les plates-formes qui ne l'implémentent pas. Sur de nombreuses plates-formes disposant d'un shell compatible avec le Bourne Shell ou le C shell, vous pouvez utiliser ceci :

```
#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
        if $lance_sous_un_shell;
```

Le système ignore la première ligne et donne le script à */bin/sh*, qui essaye d'exécuter le script Perl comme un shell script. Le shell exécute la seconde ligne comme une commande shell normale et démarre donc l'interpréteur Perl. Sur certains systèmes, `$0` ne contient pas toujours le nom de chemin complet et le **-S** indique alors à Perl de rechercher le script si nécessaire. Après que Perl l'a trouvé, il analyse les lignes et les ignore car la variable `$lance_sous_un_shell` n'est jamais vraie. `${1+"$@"}` vaudrait mieux que `$*` mais cette expression ne fonctionne pas si le script est interprété par *csh*. Afin de lancer *sh* au lieu de *csh*, certains systèmes doivent remplacer la ligne `#!` par une ligne ne contenant qu'un deux-points, qui sera poliment ignoré par Perl. D'autres systèmes ne peuvent contrôler cela et ont besoin d'un montage totalement déviant pouvant fonctionner sous *csh*, *sh* ou *perl*, comme ce qui suit :

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    & eval 'exec /usr/bin/perl -S $0 $argv:q'
                    if 0;
```

Oui. C'est laid, tout autant que les systèmes qui fonctionnent[5] ainsi.

Sur certaines plates-formes, l'option **-S** pousse également Perl à ajouter des suffixes au nom de fichier pendant qu'il le recherche. Par exemple, sur les plates-formes Win32, les suffixes *.bat* et *.cmd* sont ajoutés si la recherche du fichier original échoue et que le nom ne se termine pas déjà par l'un de ces deux suffixes. Si votre version de Perl a été construite avec le débogage activé, vous pouvez utiliser l'option **-Dp** de Perl pour suivre la progression de la recherche.

Si le nom de fichier fourni contient des séparateurs de répertoires (même simplement en tant que nom de chemin relatif, et non un chemin absolu) et si le fichier n'est pas trouvé, les plates-formes ajoutant implicitement des extensions de fichiers (non Unix) le feront et chercheront le fichier avec l'ajout de ces extensions, l'une après l'autre.

5. Nous utilisons le terme en connaissance de cause.

Sur les plates-formes comme DOS, si le script ne contient pas de séparateur de répertoires, il sera tout d'abord recherché dans le répertoire actuel avant de l'être dans le `PATH`. Sur les plates-formes Unix, le script sera recherché strictement dans `PATH`, à cause des problèmes de sécurité engendrés par l'exécution accidentelle de quelque chose dans le répertoire de travail courant sans en faire explicitement la demande.

`-T`

Force l'activation des vérifications « de marquage » afin que vous puissiez les tester. Ces vérifications ne sont généralement lancées qu'en exécutant en setuid ou setgid. Il vaut mieux les activer explicitement pour les programmes qui tournent sous le nom de quelqu'un d'autre, comme les programmes CGI. Voir le chapitre 23, *Sécurité*.

Remarquez que, pour des raisons de sécurité, Perl doit voir cette option assez tôt ; cela signifie habituellement qu'elle doit se trouver au début de la ligne de commande ou de la `ligne #!`. Si elle n'est pas trouvée assez tôt, Perl s'en plaint.

`-u`

Provoque un *core dump* de Perl après avoir compilé le script. Vous pouvez alors, en théorie, prendre ce *core dump* et le transformer en fichier exécutable grâce au programme *undump* (qui n'est pas fourni). Le démarrage est ainsi accéléré au détriment de l'espace disque (que vous pouvez minimiser en faisant *strip* sur l'exécutable). Si vous voulez exécuter une partie du script avant le vidage, utilisez plutôt l'opérateur `dump` de Perl. Remarque : la disponibilité de *undump* dépend des plates-formes ; il peut ne pas exister pour certains portages de Perl. Il a été remplacé par le nouveau générateur de code Perl-vers-C, qui est bien plus portable (mais toujours expérimental).

`-U`

Permet à Perl d'effectuer des opérations dangereuses. Actuellement, les seules opérations « dangereuses » sont la suppression de répertoires en tant que super-utilisateur et le lancement de programmes en setuid alors que les vérifications fatales de marquage sont transformées en avertissements. Remarquez que les avertissements doivent être activés pour engendrer les avertissements de vérifications de marquage.

`-v`

Affiche la version et le niveau de correctif (*patch level*) de votre exécutable Perl, ainsi que quelques informations supplémentaires.

`-V`

Affiche un résumé des principales valeurs de configuration de Perl et la valeur actuelle de `@INC`.

`-V:`*`NOM`*

Affiche sur `STDOUT` la valeur de la variable de configuration indiquée. Le *`NOM`* peut contenir des caractères d'expressions régulières, comme « **`.`** » pour correspondre à n'importe quel caractère ou « **`.*`** » pour correspondre à toute séquence de caractères optionnelle.

```
% perl -V:man.dir
man1dir='/usr/local/man/man1'
man3dir='/usr/local/man/man3'

% perl -V:'.*threads'
d_oldpthreads='undef'
use5005threads='define'
useithreads='undef'
usethreads='define'
```

Si vous demandez une variable de configuration qui n'existe pas, sa valeur sera rapportée comme « UNKNOWN » (*INCONNUE*). Les informations de configuration sont accessibles depuis l'intérieur d'un programme en utilisant le module `Config`, bien que les motifs ne soit pas implémentés pour les indices des hachages :

```
% perl -MConfig -le ' print $Config{man1dir}'
/usr/local/man/man1
```

Voir le module `Config` au chapitre 32, *Modules standard*.

-w

Affiche des messages d'avertissement concernant les variables qui ne sont mentionnées qu'une fois et sur les valeurs scalaires qui sont utilisées avant d'être positionnées. Prévient également des redéfinitions de sous-programme et des références à des handles de fichier indéfinis ou de ceux, ouverts en lecture seule, sur lesquels vous essayez d'écrire. Avertit également si vous utilisez des valeurs en tant que nombres alors qu'elles ne semblent pas numériques ou si vous utilisez un tableau comme s'il s'agissait d'un scalaire ou si vos sous-programmes rentrent dans plus de 100 récursions, ainsi que d'innombrables choses du même genre. Voir toutes les entrées marquées « (W) » au chapitre 33, *Messages de diagnostic*.

Cette option ne fait que positionner la variable globale `$^W`. Elle n'a pas d'effet sur les avertissements lexicaux — voir pour cela les options **-W** et **-X**. Vous pouvez activer ou désactiver des avertissements spécifiques *via* le pragma `use warnings`, décrit au chapitre 31.

-W

Active de manière inconditionnelle et permanente tous les avertissements tout au long du programme, même si les avertissements étaient désactivés localement en utilisant `no warnings` ou `$^W = 0`. Ceci comprend tous les fichiers chargés par `use`, `require` ou `do`. Pensez-y comme l'équivalent Perl de la commande *lint*(1).

-x*REPERTOIRE*

-x

Indique à Perl d'extraire un script qui est inclus dans un message. Les lignes résiduelles du début sont éliminées jusqu'à la première commençant par `#!` et contenant la chaîne « `perl` ». Toutes les options significatives de cette ligne après le mot « `perl` » seront appliquées. Si un nom de répertoire est spécifié, Perl basculera vers ce répertoire avant de lancer le script. L'option **-x** permet seulement d'éliminer les lignes superflues du début, pas les lignes superflues de la fin. Le script doit se terminer par `__END__` ou par `__DATA__` s'il y a des lignes superflues à ignorer après la fin du script. (Le script peut traiter au besoin tout ou partie des résidus restants *via*

le handle de fichier DATA. Il pourrait même en théorie se déplacer avec seek au début du fichier et traiter les résidus du début.)

-X

Désactive de manière inconditionnelle et permanente tous les avertissements, exactement à l'opposé de ce que fait l'option **-W**.

Variables d'environnement

En plus des diverses options modifiant explicitement le comportement de Perl, vous pouvez positionner diverses variables d'environnement pour influencer divers comportements sous-jacents. La manière de positionner ces variables d'environnement est dépendante du système, mais une astuce que vous devriez connaître si vous employez *sh*, *ksh* ou *bash*, est que vous pouvez positionner temporairement une variable d'environnement pour une seule commande, comme s'il s'agissait d'une drôle de sorte d'option. Elle doit être positionnée devant la commande :

```
$ PATH='/bin:/usr/bin' perl ma_programmette
```

Vous pouvez faire quelque chose de similaire avec un sous-shell dans *csh* ou *tcsh* :

```
% (setenv PATH='/bin:/usr/bin'; perl ma_programmette)
```

Sinon, vous auriez généralement positionné les variables d'environnement dans un fichier dont le nom ressemblerait à *.cshrc* ou *.profile* dans votre répertoire maison. Sous *csh* et *tcsh*, vous auriez écrit :

```
% setenv PATH='/bin:/usr/bin'
```

Et sous *sh*, *ksh* et *bash*, vous auriez écrit :

```
$ PATH='/bin:/usr/bin'; export PATH
```

D'autres systèmes auront d'autres moyens pour les positionner d'une manière semi-permanente. Voici les variables d'environnement auxquelles Perl accorde de l'attention :

HOME

Utilisée si chdir est appelé sans argument.

LC_ALL, LC_CTYPE, LC_COLLATE, LC_NUMERIC, PERL_BADLANG

Variables d'environnement contrôlant comment Perl gère les données spécifiques aux langages naturels particuliers. Voir la documentation en ligne de *perllocale*.

LOGDIR

Utilisée si chdir est appelé sans argument et si HOME n'est pas positionnée.

PATH

Utilisée lors de l'exécution de sous-processus et dans la recherche du programme si l'option **-S** est utilisée.

PERL5LIB

Une liste de répertoires, séparés par des deux-points, dans lesquels rechercher les fichiers de bibliothèque de Perl avant de les chercher dans la bibliothèque standard et dans le répertoire courant. Tous les répertoires spécifiques à l'architecture aux

endroits spécifiés sont automatiquement inclus s'ils existent. Si la variable d'environnement PERL5LIB n'est pas définie, on consulte PERLLIB pour assurer une compatibilité antérieure avec les versions plus anciennes.

Lors d'une exécution avec vérification du marquage (soit parce que le programme s'exécutait en setuid ou setgid, soit parce que l'option **-T** était utilisée), aucune des deux variables de recherche de bibliothèque n'est utilisée. De tels programmes doivent employer le pragma `use lib` dans cet objectif.

PERL5OPT

Options de ligne de commande par défaut. Les options dans cette variable sont considérées comme faisant partie de toute ligne de commande Perl. Seules les options **-[DIMUdmw]** sont autorisées. Lors d'une exécution avec vérifications de marquage (car le programme s'exécutait en setuid ou setgid ou si l'option **-T** était utilisée), cette variable est ignorée. Si PERL5OPT commence par **-T**, le marquage sera activé et toutes les options suivantes seront ignorées.

PERL5DB

La commande utilisée pour charger le code du débogueur. La valeur par défaut est :

```
BEGIN { require 'perl5db.pl' }
```

Voir le chapitre 20 pour d'autres utilisations de cette variable.

PERL5SHELL

(seulement sur les portages Microsoft) Peut être positionnée à une possibilité de shell que Perl doit utiliser en interne pour exécuter les commandes *via* des apostrophes inverses ou *via* `system`. La valeur par défaut est `cmd.exe /x/c` sur WinNT et `command.com /c` sur Win95. La valeur est considérée comme étant délimitée par des espaces. Préfixez tout caractère devant être isolé (comme un espace ou une antislash) par un antislash.

Remarquez que Perl n'utilise pas COMSPEC pour cela car COMSPEC a un haut degré de variabilité entre les utilisateurs, ce qui amène à des soucis de portabilité. De plus, Perl peut utiliser un shell qui ne convienne pas à un usage interactif et le fait de positionner COMSPEC à un tel shell peut interférer avec le fonctionnement correct d'autres programmes (qui regardent habituellement COMSPEC pour trouver un shell permettant l'utilisation interactive).

PERLLIB

Une liste de répertoires, séparés par des deux-points, dans lesquels rechercher les fichiers de bibliothèque de Perl avant de les chercher dans la bibliothèque standard et dans le répertoire courant. Si PERL5LIB est définie, PERLLIB n'est pas utilisée.

PERL_DEBUG_MSTATS

Appropriée seulement si Perl est compilé avec la fonction `malloc` incluse dans la distribution de Perl (c'est-à-dire si `perl -V:d_mymalloc` vaut « `define` »). Si elle est positionnée, cela provoque un affichage de statistiques sur la mémoire après l'exécution. Si sa valeur est un entier supérieur à un, les statistiques sur la mémoire sont également affichées après la compilation.

PERL_DESTRUCT_LEVEL

Appropriée seulement si votre exécutable perl a été construit en activant le débo-

gage, ceci contrôle le comportement de la destruction globale des objets et autres références.

À part celles-ci, Perl lui-même n'utilise pas d'autres variables d'environnement, sauf pour les rendre disponibles au programme s'exécutant et aux processus fils que lance ce programme. Certains modules, standard ou autres, peuvent se soucier d'autres variables d'environnement. Par exemple, le pragma `use re` utilise `PERL_RE_TC` et `PERL_RE_COLORS`, le module `Cwd` utilise `PWD` et le module `CGI` utilise les nombreuses variables d'environnement positionnées par votre démon HTTP (c'est-à-dire, votre serveur Web) pour passer des informations au script CGI.

Les programmes exécutés en setuid feraient bien d'exécuter les lignes suivantes avant de faire quoi que ce soit d'autre, simplement pour que les gens restent honnêtes :

```
$ENV{PATH}  = '/bin:/usr/bin'; # ou ce dont vous avez besoin
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

Voir le chapitre 23 pour plus de détails.

20

Le débogueur Perl

Avant tout, avez-vous essayé le pragma `use warnings`?

Si vous invoquez Perl avec l'option **`-d`**, votre programme tournera sous le débogueur Perl. Celui-ci fonctionne comme un environnement Perl interactif, offrant une invite (*prompting*) pour des commandes de débogage qui vous permettent d'examiner le code source, de positionner des points d'arrêt, d'afficher la pile de vos appels de fonctions, de changer la valeur des variables et ainsi de suite. Toute commande non reconnue par le débogueur est directement exécutée (en utilisant `eval`) en tant que code Perl dans le paquetage du code actuellement en train d'être débogué. (Le débogueur utilise le paquetage `DB` pour ses propres informations d'état, afin d'éviter de piétiner les vôtres.) C'est si formidablement pratique que les gens lancent souvent le débogueur tout seul pour tester des constructions Perl interactivement pour voir ce qu'elles font. Dans ce cas, le programme que vous dites à Perl de déboguer n'a pas d'importance, nous en choisirons donc un sans grande signification :

```
% perl -de 42
```

En Perl, le débogueur n'est pas un programme séparé de celui qui est débogué, comme il l'est d'habitude dans un environnement de programmation typique. L'option **`-d`** indique plutôt au compilateur d'insérer des informations sur le source dans les arbres syntaxiques qu'il est sur le point de passer à l'interpréteur. Cela signifie que votre code doit d'abord compiler correctement pour que le débogueur puisse s'en occuper. Puis si c'est le cas, l'interpréteur précharge un fichier de bibliothèque Perl spécial contenant le débogueur lui-même.

```
% perl -d /chemin/du/programme
```

Le programme s'arrêtera immédiatement avant la première instruction à l'exécution (mais voir la section *Utilisation du débogueur* concernant les instructions à la compilation) et vous demande d'entrer une commande de débogueur. Chaque fois que le débogueur s'arrête et vous montre une ligne de code, il affiche la ligne qu'il est *sur le point* d'exécuter, non celle qu'il vient d'exécuter.

Lorsque le débogueur rencontre une ligne, il vérifie d'abord s'il y a un point d'arrêt, affiche la ligne (si le débogueur est en mode trace), accomplit toutes les actions (créées

avec la commande a décrite plus loin dans *Commandes du débogueur*) et finalement affiche une invite (*prompt*) à l'utilisateur s'il existe un point d'arrêt ou si le débogueur est en mode pas-à-pas. Sinon, il évalue la ligne normalement et continue avec la ligne suivante.

Utilisation du débogueur

L'invite du débogueur ressemble à :

```
DB<8>
```

ou même :

```
DB< <17> >
```

où le numéro indique combien de commandes vous avez effectuées. Un mécanisme d'historique ressemblant à celui de *csh* permet d'accéder à des commandes antérieures par leur numéro. Par exemple, `!17` répéterait la commande numéro 17. Le nombre de signes inférieur/supérieur indique la profondeur du débogueur. Vous en obtenez plus d'un, par exemple, si vous vous trouvez déjà à un point d'arrêt et que vous affichez ensuite le résultat d'un appel de fonction qui comporte elle-même un point d'arrêt.

Si vous voulez entrer une commande multiligne, comme une définition de sous-programme comportant plusieurs instructions, vous pouvez protéger le saut de ligne qui termine normalement la commande du débogueur par un antislash. En voici un exemple :

```
DB<1> for (1..3) {       \
    cont:     print "ok\n"; \
    cont: }
    ok
    ok
    ok
```

Admettons que vous vouliez démarrer le débogueur sur un petit programme de votre cru (appelons-le *puce_du_chameau*[1]) et arrêtons-le dès qu'il arrive sur une fonction appelée `infeste`. Voici comment vous feriez cela :

```
% perl -d puce_du_chameau

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(puce_du_chameau:2): parasite('bacterie', 4);
DB<1>
```

1. *N.d.T.* : En anglais, « bogue » se dit *bug*, qui signifie également « punaise », « insecte », « coléoptère » ou « microbe », ce qui explique les différentes allusions à des parasites dans les exemples de ce chapitre. Nous aurions pu adapter ces exemples à l'enveloppe de la châtaigne (sens premier de « bogue » en français) mais avouez qu'un bogue informatique est aussi nuisible que certains insectes !

Le débogueur arrête votre programme juste avant la première instruction à l'exécution (mais voir ci-dessous les instructions à la compilation) et vous demande d'entrer une commande. Encore une fois, dès que le débogueur s'arrête pour vous montrer une ligne de code, il affiche celle qui est *sur le point* d'être exécutée, non celle qui vient de l'être. La ligne affichée peut ne pas ressembler exactement à celle présente dans votre fichier source, particulièrement si vous l'avez lancé par l'intermédiaire d'un préprocesseur.

Vous aimeriez maintenant vous arrêter dès votre programme arrive à la fonction `infeste`, vous y établissez donc un point d'arrêt comme ceci :

```
DB<1> b infeste
DB<2> c
```

Le débogueur continue maintenant jusqu'à ce qu'il tombe sur cette fonction, où il affiche ceci :

```
main::infeste(puce_du_chameau:8):       my $microbes = int rand(3);
```

Pour examiner une « fenêtre » de code source autour du point d'arrêt, utilisez la commande `w` :

```
DB<2> w
5     }
6
7       sub infeste {
8==>b       my microbes = int rand(3);
9:          our $Maitre;
10:         contaminer($Maitre);
11:         warn "bain nécessaire"
12             if $Maitre && $Maitre->isa("Humain");
13
14:         print "$microbes attrapés\n";

DB<2>
```

Comme vous pouvez le voir grâce au marqueur `==>`, votre ligne courante est la numéro 8 et, grâce au `b` qui s'y trouve, vous savez qu'elle comporte un point d'arrêt. Si vous aviez positionné une action, il y aurait également eu un `a`. Les numéros avec un deux-points peuvent comporter des points d'arrêt, les autres ne le peuvent pas.

Pour voir qui a appelé qui, demandez la trace arrière de la pile (*stack backtrace*) en utilisant la commande **`T`** :

```
DB<2> T
$ = main::infeste called from file 'Deplacement.pm' line 4
@ = Deplacement::pattes(1, 2, 3, 4) called from file 'puce_du_chameau' line 5
. = main::parasite('bacterie', 4) called from file 'puce_du_chameau' ligne 2
```

Le premier caractère (`$`, `@` ou `.`) indique si la fonction a été appelée respectivement dans un contexte scalaire, de liste ou vide. Il y a trois lignes car vous étiez à une profondeur de trois fonctions lorsque vous avez lancé la trace arrière de la pile. Voici ce que signifie chaque ligne :

- La première ligne dit que vous vous trouviez dans la fonction `main::infeste` lorsque vous avez lancé la trace de la pile. Elle vous indique que la fonction a été appelée dans un contexte scalaire depuis la ligne numéro 4 du fichier *Deplacement.pm*.

Elle montre également qu'elle a été appelée sans aucun argument, ce qui veut dire qu'elle a été appelée en tant que `&infeste` au lieu de la manière normale `infeste()`.

- La deuxième ligne montre que la fonction `Deplacement::pattes` a été appelée dans un contexte de liste depuis la ligne numéro 5 du fichier *puce_du_chameau*, avec ces quatre arguments.
- La troisième ligne montre que `main::parasite` a été appelée dans un contexte vide depuis la ligne numéro 2 de *puce_du_chameau*.

Si vous avez des instructions exécutables lors de la phase de compilation, comme du code venant de blocs `BEGIN` ou `CHECK` ou des instructions `use`, elles ne seront *pas* stoppées de manière ordinaire par le débogueur, alors que les `require` et les blocs `INIT` le sont, puisqu'ils interviennent après la transition vers la phase d'exécution (voir le chapitre 18, *Compiler*). Les instructions de la phase de compilation peuvent être tracées avec l'option `AutoTrace` positionnée dans `PERLDB_OPTS`.

Vous pouvez exercer un léger contrôle sur le débogueur Perl depuis l'intérieur de votre programme Perl lui-même. Vous pourriez faire ceci en positionnant, par exemple, un point d'arrêt automatique à un certain sous-programme lorsqu'un programme particulier s'exécute sous le débogueur. Cependant, depuis votre propre code Perl, vous pouvez transférer de nouveau le contrôle au débogueur en utilisant l'instruction suivante, qui est inoffensive si le débogueur n'est pas lancé :

```
$DB::single = 1;
```

Si vous positionnez `$DB::single` à 2, cela équivaut à la commande `n`, tandis qu'une valeur de 1 émule la commande `s`. La variable `$DB::trace` devrait être mise à 1 pour simuler le `t`.

Une autre façon de déboguer un module consiste à positionner un point d'arrêt au chargement :

```
DB<7> b load c:/perl/lib/Carp.pm
Will stop on load of 'c:/perl/lib/Carp.pm'.
```

puis de redémarrer le débogueur en utilisant la commande `R`. Pour un contrôle plus fin, vous pouvez utiliser `b compile nom_sous_programme` pour s'arrêter aussitôt qu'un sous-programme particulier est compilé.

Commandes du débogueur

Lorsque vous tapez des commandes dans le débogueur, vous n'avez pas besoin de les terminer par un point-virgule. Utilisez un antislash pour continuer les lignes (mais uniquement dans le débogueur).

Puisque le débogueur utilise `eval` pour exécuter des commandes, les initialisations avec `my`, `our` et `local` disparaîtrons une fois la commande terminée. Si une commande du débogueur coïncide avec une quelconque fonction de votre propre programme, vous n'avez qu'à préfixer l'appel de fonction avec quelque chose qui ne ressemble pas à une commande du débogueur, comme un `;` ou un `+` au début.

Si la sortie d'une commande interne du débogueur défile au-delà des limites de votre écran, vous n'avez qu'à préfixer la commande avec un symbole de pipe afin qu'elle passe

par votre *pager* :

```
DB<1> |h
```

Le débogueur comprend une grande quantité de commandes et nous les avons divisées (quelque peu arbitrairement) dans les catégories suivantes : pas à pas et lancement, points d'arrêt, traçage, affichage, localisation de code, exécution automatique de commandes et, bien entendu, divers.

La commande la plus importante est peut-être `h`, qui fournit de l'aide. Si vous tapez `h h` à l'invite du débogueur, vous obtiendrez une page d'aide compacte conçue pour tenir sur un seul écran. Si vous tapez `h` *`COMMANDE`*, vous obtiendrez de l'aide sur cette *`COMMANDE`* du débogueur.

Pas à pas et lancement

Le débogueur fonctionne en parcourant votre programme pas-à-pas (*stepping*), ligne par ligne. Les commandes suivantes vous permettent de contrôler ce que vous court-circuitez et où vous vous arrêtez.

`s`
`s` *`EXPR`*

La commande `s` du débogueur avance dans le programme d'un seul pas. C'est-à-dire que le débogueur exécutera la prochaine ligne de votre programme jusqu'à atteindre une autre instruction, en descendant dans les appels de sous-programmes si nécessaire. Si la prochaine ligne à exécuter implique un appel de fonction, le débogueur s'arrête alors à la première ligne à l'intérieur de cette fonction. Si une *`EXPR`* est fournie et qu'elle comprend des appels de fonctions, celles-ci seront également parcourues pas-à-pas.

`n`
`n` *`EXPR`*

La commande `n` exécute des appels de sous-programme, sans les parcourir pas à pas, jusqu'au début de la prochaine instruction du même niveau (ou d'un niveau plus haut). Si une *`EXPR`* est fournie et qu'elle comprend des appels de fonctions, celles-ci seront exécutées en s'arrêtant avant chaque instruction.

`<ENTRÉE>`

Si vous ne faites qu'appuyer sur la touche `<ENTRÉE>` à l'invite du débogueur, la commande `s` ou `n` précédente est répétée.

`.`

La commande `.` renvoie le pointeur interne du débogueur vers la dernière ligne exécutée et affiche cette ligne.

`r`

Cette commande continue jusqu'au retour du sous-programme actuellement exécuté. Elle affiche la valeur renvoyée si l'option `PrintRet` est positionnée, ce qui est le cas par défaut.

Points d'arrêt

b
b *LIGNE*
b *CONDITION*
b *LIGNE CONDITION*
b *NOM_SOUS_PROGRAMME*
b *NOM_SOUS_PROGRAMME CONDITION*
b postpone *NOM_SOUS_PROGRAMME*
b postpone *NOM_SOUS_PROGRAMME CONDITION*
b compile *NOM_SOUS_PROGRAMME*
b load *NOM_FICHIER*

La commande b du débogueur positionne un point d'arrêt avant la *LIGNE*, indiquant au débogueur d'arrêter le programme à cet endroit pour que vous puissiez fureter autour. Si *LIGNE* est omise, la commande positionne un point d'arrêt sur la ligne qui est sur le point d'être exécutée. Si *CONDITION* est spécifiée, elle est évaluée à chaque fois que l'instruction est atteinte : un point d'arrêt n'est déclenché que si la *CONDITION* est vraie. Les points d'arrêt ne peuvent être positionnés que sur les lignes commençant par une instruction exécutable. Remarquez que les conditions n'emploient pas if :

```
b 237 $x > 30
b 237 ++$compteur237 < 11
b 33 /motif/i
```

La forme b *NOM_SOUS_PROGRAMME* positionne un point d'arrêt (qui peut être conditionnel) avant la première ligne du sous-programme cité. *NOM_SOUS_PROGRAMME* peut être une variable contenant une référence de code ; si c'est le cas, *CONDITION* n'est pas implémentée.

Il existe plusieurs manières de positionner un point d'arrêt sur du code qui n'a même pas encore été compilé. La forme b postpone positionne un point d'arrêt (qui peut être conditionnel) à la première ligne de *NOM_SOUS_PROGRAMME* après qu'il a été compilé.

La forme b compile positionne un point d'arrêt sur la première instruction devant être exécutée après que *NOM_SOUS_PROGRAMME* a été compilé. Remarquez qu'au contraire de la forme avec postpone, cette instruction se trouve à l'extérieur du sous-programme en question car ce dernier n'a pas encore été appelé, il a seulement été compilé.

La forme b load positionne un point d'arrêt sur la première ligne exécuté du fichier. Le *NOM_FICHIER* devrait être un nom de chemin complet tel qu'on l'aurait trouvé dans les valeurs de %INC.

d
d *LIGNE*

Cette commande supprime le point d'arrêt de la *LIGNE* ; si celle-ci est omise, la commande supprime le point d'arrêt sur la ligne qui est sur le point d'être exécutée.

D

Cette commande supprime tous les points d'arrêt.

`L`

Cette commande liste tous les points d'arrêt et toutes les actions.

`c`
`C` *`LIGNE`*

Cette commande continue l'exécution, en insérant éventuellement un point d'arrêt, actif une seule fois, à la *`LIGNE`* spécifiée.

Traçage

`T`

Cette commande produit une trace arrière de la pile (*backtrace stack*).

`t`
`t` *`EXPR`*

Cette commande bascule l'activation ou la désactivation du mode trace, qui imprime chaque ligne de votre programme au moment où elle est évaluée. Voir également l'option `AutoTrace`, présentée plus loin dans ce chapitre. Si une *`EXPR`* est fournie, le débogueur tracera son exécution. Voir également plus loin la section *Exécution autonome*.

`W`
`W` *`EXPR`*

Cette commande ajoute *`EXPR`* en tant qu'expression globale de surveillance. (Une expression de surveillance est une expression qui provoquera un point d'arrêt lorsque sa valeur changera.) Si aucune *`EXPR`* n'est fournie, toutes les expressions de surveillance sont supprimées.

Affichage

Le débogueur de Perl comprend plusieurs commandes pour examiner des structures de données pendant que votre programme est immobilisé à un point d'arrêt.

`p`
`p` *`EXPR`*

Cette commande est identique à `print DB::OUT` *`EXPR`* dans le paquetage courant. En particulier, comme il s'agit de la propre fonction `print` de Perl, les structures de données imbriquées et les objets ne sont pas affichés — utiliser la commande `x` pour ce faire. Le handle `DB::OUT` imprime sur votre terminal (ou peut-être une fenêtre d'éditeur) quel que soit l'endroit où la sortie standard a pu être redirigée.

`x`
`x` *`EXPR`*

La commande `x` évalue son expression dans un contexte de liste et affiche le résultat, agréablement présenté (*pretty-printed*). C'est-à-dire que les structures de données imbriquées sont imprimées récursivement et avec les caractères invisibles convenablement encodés.

```
V
V PAQUETAGE
V PAQUETAGE  VARS
```

Cette commande affiche toutes les variables (ou quelques-unes, lorsque vous spécifiez *VARS*) dans le *PAQUETAGE* spécifié (`main` par défaut) en utilisant une présentation agréable (*pretty printer*). Les hachages affichent leurs clés et leurs valeurs, les caractères de contrôles sont présentés lisiblement, les structures de données imbriquées sont imprimées de manière lisible et ainsi de suite. Ceci est identique à un appel de la commande `x` sur chaque variable possible, sauf que `x` fonctionne également avec les variables lexicales. De plus, vous pouvez tapez ici les identificateurs *sans* spécificateur de type, tel que `$` ou `@`, comme ceci :

```
V Animal::Chameau SPOT FIDO
```

Au lieu d'un nom de variable dans *VARS*, vous pouvez utiliser `~`*MOTIF* ou `!`*MOTIF* pour imprimer les variables existantes dont le nom correspond ou ne correspond pas au *MOTIF* spécifié.

```
X
X VARS
```

Cette commande est identique à `V` *PAQUETAGE_COURANT*, où *PAQUETAGE_COURANT* est le paquetage où la ligne courante a été compilée.

```
H
H -NUMÉRO
```

Cette commande affiche la dernière commande *NUMÉRO*. Seules les commandes de plusieurs caractères sont stockées dans l'historique. (Sinon, la plupart d'entre elles seraient des `s` ou des `n`.) Si *NUMÉRO* est omis, toutes les commandes sont listées.

Localisation de code

À l'intérieur du débogueur, vous pouvez extraire et afficher des parties de votre programme avec ces commandes.

```
l
l LIGNE
l NOM_SOUS_PROGRAMME
l MIN+INCR
l MIN-MAX
```

La commande `l` liste quelques-unes des prochaines lignes de votre programme ou la *LIGNE* spécifiée, si elle est fournie ou quelques lignes au début du sous-programme ou de la référence de code *NOM_SOUS_PROGRAMME*.

La forme `l` *MIN+INCR* liste *INCR*+1 lignes, en commençant à *MIN*. La forme `l` *MIN-MAX* liste les lignes *MIN* jusqu'à *MAX*.

```
-
```

Cette commande liste quelques lignes au début de votre programme.

`w`
`w` *`LIGNE`*

Liste une fenêtre (quelques lignes) autour de la *`LIGNE`* de code source donnée ou de la ligne courante, si aucune *`LIGNE`* n'est fournie.

`f` *`NOM_FICHIER`*

Cette commande vous permet de visualiser un programme ou une instruction `eval` différent. Si le *`NOM_FICHIER`* n'est pas un nom de chemin absolu tel qu'on l'aurait trouvé dans les valeurs de `%INC`, il est interprété en tant qu'expression régulière pour trouver le nom de chemin que vous voulez.

`/MOTIF/`

Cette commande recherche le *`MOTIF`* en avant dans votre programme ; le `/` final est optionnel. Le *`MOTIF`* entier est également optionnel et s'il est omis, la recherche précédente est répétée.

`?MOTIF?`

Cette commande recherche le *`MOTIF`* en arrière dans votre programme ; le `/` final est optionnel. La recherche précédente est répétée si le *`MOTIF`* est omis.

`S`
`S` *`MOTIF`*
`S` *`!MOTIF`*

La commande `S` liste les sous-programmes dont le nom correspond (ou avec `!`, ne correspond pas) au *`MOTIF`*. Si aucun *`MOTIF`* n'est fourni, tous les sous-programmes sont listés.

Actions et exécution de commandes

Depuis l'intérieur du débogueur, vous pouvez spécifier des actions à accomplir à certains moments. Vous pouvez également lancer des programmes externes.

`a`
`a` *`COMMANDE`*
`a` *`LIGNE`*
`a` *`LIGNE COMMANDE`*

Cette commande positionne une action à accomplir avant d'exécuter la *`LIGNE`* ou la ligne courante si *`LIGNE`* est omise. Par exemple, ceci affiche `$truc` chaque fois que la ligne 53 est atteinte :

```
a 53 print "DB a trouvé $truc\n"
```

Si aucune *`COMMANDE`* n'est spécifiée, l'action à la *`LIGNE`* spécifiée est supprimée. Sans *`LIGNE`*, ni *`ACTION`*, l'action sur la ligne courante est supprimée.

`A`

La commande `A` du débogueur supprime toutes les actions.

`<`
`< ?`
`< EXPR`
`<< EXPR`

La forme `< EXPR` spécifie une expression Perl à évaluer avant toute invite du débogueur. Vous pouvez ajouter une autre expression avec la forme `<< EXPR`, les lister avec `< ?` et toutes les supprimer avec un simple `<`.

`>`
`> ?`
`> EXPR`
`>> EXPR`

Les commandes `>` se comportent exactement comme leur cousines `<` mais sont exécutées après l'invite du débogueur plutôt qu'avant.

`{`
`{ ?`
`{ COMMANDE`
`{{ COMMANDE`

Les commandes `{` du débogueur se comportent exactement comme `<` mais spécifient une commande du débogueur à exécuter plutôt qu'une expression Perl. Un avertissement est émis s'il apparaît que vous avez accidentellement entré un bloc de code à la place. Si c'est vraiment ce que vous voulez faire, écrivez le avec `;{ ... }` ou même `do { ... }`.

`!`
`! NUMÉRO`
`! -NUMÉRO`
`!MOTIF`

Un `!` isolé répète la commande précédente. Le *NUMÉRO* spécifie quelle commande exécuter depuis l'historique ; par exemple, `! 3` exécute la troisième commande tapée dans le débogueur. Si un signe moins précède le *NUMÉRO*, on compte les commandes à rebours : `! -2` exécute l'avant-dernière commande. Si un *MOTIF* (sans slash) est fourni à la place d'un *NUMÉRO*, la dernière commande commençant par ce *MOTIF* est exécutée. Voir également l'option du débogueur `recallCommand`.

`!! CMD`

Cette commande du débogueur lance la commande externe *CMD* dans un sous-processus, qui lira depuis `DB::IN` et écrira vers `DB::OUT`. Voir également l'option du débogueur `shellBang`. Cette commande emploie le shell cité dans `$ENV{SHELL}`, qui peut parfois interférer avec une interprétation correcte des informations concernant les statuts, les signaux et les *core dump*. Si vous voulez une valeur de sortie cohérente, positionnez `$ENV{SHELL}` à `/bin/sh`.

`|`
`| CMD_DB`
`|| CMD_PERL`

La commande `|CMD_DB` lance la commande *CMD_DB* du débogueur, en *pipant* `DB::OUT` vers `$ENV{PAGER}`. On emploie souvent ceci avec les commandes qui pro-

duiraient sinon une sortie trop longue, comme :

```
DB<1> |V main
```

Remarquez que ceci est valable pour les commandes du débogueur et non pour celles que vous auriez tapées depuis votre shell. Si vous vouliez envoyer la commande externe *who* vers votre pager, vous pourriez faire quelque chose comme ceci :

```
DB<1> !!who | more
```

La commande `||`*CMD_PERL* est semblable à `|`*CMD_DB* mais `DB::OUT` est également temporairement sélectionné (avec `select`), afin que toutes les commandes appelant `print`, `printf` ou `write` sans handle de fichier soient également envoyées vers le pipe. Par exemple, si vous aviez une fonction générant des tonnes d'affichages en appelant `print`, vous utiliseriez cette commande à la place de la précédente pour envoyer la sortie vers le *pager* :

```
DB<1> sub qui { print "Utilisateurs : ", `who` }
DB<2> ||qui()
```

Commandes diverses

`q` *et* `^D`

Ces commandes quittent le débogueur. Il s'agit de la manière recommandée de sortir du débogueur, bien que taper deux fois `exit` fonctionne parfois. Positionnez l'option `inhibit_exit` à `0` si vous voulez être capable de passer directement à la fin du programme et de tout de même rester dans le débogueur. Vous pouvez également avoir besoin de positionner `$DB::finished` à `0` si vous voulez suivre la destruction globale.

`R`

Redémarre le débogueur en lançant une nouvelle session avec `exec`. Le débogueur essaie de maintenir votre historique entre les sessions mais quelques réglages internes et quelques options de la ligne de commande peuvent être perdus. Les réglages suivant sont actuellement préservés : l'historique, les points d'arrêt, les actions, les options du débogueur et les options **`-w`**, **`-I`** et **`-e`** de la ligne de commande de Perl.

`=`
`=` *ALIAS*
`=` *ALIAS VALEUR*

Cette commande affiche la valeur actuelle d'*ALIAS* si aucune *VALEUR* n'est fournie. Avec une *VALEUR*, elle définit une nouvelle commande du débogueur avec le nom *ALIAS*. Si *ALIAS* et *VALEUR* sont tous deux omis, tous les alias actuels sont listés. Par exemple :

```
= quit q
```

Un *ALIAS* devrait être un simple identificateur et devait également se convertir en un simple identificateur. Vous pouvez obtenir des mécanismes d'alias plus sophistiqués en ajoutant directement vos propres entrées à `%DB::aliases`. Voir *Personnalisation du débogueur* plus loin dans ce chapitre.

`man`
`man` *`PAGE_DE_MANUEL`*

Cette commande appelle la visionneuse de documentation par défaut de votre système pour la page donnée ou la visionneuse elle-même si *`PAGE_DE_MANUEL`* est omise. Si cette visionneuse est *man*, les informations courantes de `%Config` sont utilisées pour l'invoquer. Le préfixe « `perl` » sera automatiquement fourni à votre place si nécessaire ; ceci vous permet de taper `man debug` et `man op` depuis le débogueur.

Sur les systèmes ne disposant pas d'ordinaire de l'utilitaire *man*, le débogueur invoque *perldoc* ; si vous voulez changer ce comportement, positionnez `$DB::doccmd` avec la visionneuse que vous désirez. Ceci peut être initialisé dans un fichier *rc* ou par l'intermédiaire d'une affectation directe.

`O`
`O` *`OPTION`* `...`
`O` *`OPTION`*`?` `...`
`O` *`OPTION`*`=`*`VALEUR`* `...`

La commande `O` vous permet de manipuler les options du débogueur, qui sont énumérées au paragraphe « Options du débogueur » plus loin dans ce chapitre. La form `O` *`OPTION`* positionne chacune des options listées à `1`. Si un point d'interrogation suit une *`OPTION`*, sa valeur actuelle est affichée.

La forme `O` *`OPTION`*`=`*`VALEUR`* positionne la valeur ; si *VALEUR* contient des espaces, ils doivent être isolés (*quoted*). Par exemple, vous pouvez positionner `O pager="less -MQeicsNfr"` pour utiliser *less* avec ces drapeaux spécifiques. Vous pouvez utiliser des apostrophes ou des guillemets, mais si c'est le cas, vous devez isoler des instances imbriquées avec un type de caractère de protection identique à celui avec lequel vous avez commencé. Vous devez également protéger tout antislash précédant immédiatement l'apostrophe ou le guillemet mais n'étant pas destiné à protéger l'apostrophe ou le guillemet lui-même. En d'autres termes, vous n'avez qu'à suivre les règles de l'isolement par apostrophes sans tenir compte du type de caractère d'isolement réellement utilisé. Le débogueur répond en vous montrant la valeur de l'option qui vient d'être positionnée, en utilisant toujours la notation avec des apostrophes dans l'affichage :

```
DB<1> O OPTION='ceci n\'est pas mal'
                OPTION = 'ceci n\'est pas mal'

DB<2> O OPTION="\"N'est-ce pas ?\", dit-elle"
                OPTION = '"N\'est-ce pas ?", dit-elle'
```

Pour des raisons historiques, le `=`*`VALEUR`* est optionnel, mais vaut `1` par défaut, seulement là où cela peut être fait en toute sécurité — c'est-à-dire surtout pour les options booléennes. Il vaut mieux affecter une *`VALEUR`* spécifique en utilisant `=`. L'*`OPTION`* peut être abrégée mais à moins que vous ne soyez volontairement occulte, il vaudrait certainement mieux vous abstenir. Plusieurs options peuvent être positionnées ensemble. Voir la section *Options du débogueur* pour en avoir la liste.

Personnalisation du débogueur

Le débogueur contient probablement assez de points d'entrée pour sa configuration, pour que vous n'ayez jamais à le modifier vous-même. Vous pouvez changer le comportement du débogueur depuis l'intérieur de ce dernier en utilisant sa commande `O`, depuis la ligne de commande *via* la variable d'environnement `PERLDB_OPTS` et en lançant l'une des commandes prédéfinies stockées dans des fichiers *rc*.

Implémentation de l'éditeur pour le débogage

Le mécanisme d'historique de la ligne de commande du débogueur n'offre pas de facilités d'édition pour la ligne de commande comme le font de nombreux shells : vous ne pouvez pas accéder aux lignes précédentes avec `^p`, ni vous déplacer au début de la ligne avec `^a`, bien que vous puissiez exécuter les lignes précédentes en utilisant la syntaxe avec un point d'exclamation, familière aux utilisateurs du shell. Toutefois, si vous installez les modules `Term::Readkey` et `Term::ReadLine` de CPAN, vous obtiendrez des fonctionnalités d'édition complètes, similaires à ce qu'offre GNU *readline*(3).

Si *emacs* est installé sur votre système, il peut interagir avec le débogueur Perl pour fournir un environnement intégré de développement logiciel, avec des réminiscences de ses interactions avec les débogueurs C. Perl est distribué avec un fichier de démarrage pour pousser *emacs* à agir comme un éditeur sensible à la syntaxe qui comprenne la syntaxe (du moins, une partie) de Perl. Regardez dans le répertoire *emacs* du code source de la distribution Perl. Les utilisateurs de *vi* devraient également regarder *vim* (et *gvim*, la version fenêtre et souris) pour obtenir une coloration des mots-clés de Perl.

Une configuration similaire, faite par l'un des auteurs (Tom), pour interagir avec le *vi* de n'importe quel fabriquant et le système de fenêtrage X11, est également disponible. Ceci fonctionne comme l'implémentation intégrée multifenêtre fournie par *emacs*, où le débogueur dirige l'éditeur. Toutefois, à l'heure où ces lignes sont écrites, son emplacement éventuel dans la distribution de Perl reste incertain. Mais nous avons pensé que vous deviez connaître cette possibilité.

Personnalisation avec des fichiers d'initialisation

Vous pouvez effectuer certaines personnalisations en configurant un fichier *.perldb* ou *perldb.ini* (selon votre système d'exploitation), contenant du code d'initialisation. Ce fichier d'initialisation contient du code Perl, et non des commandes du débogueur, et il est traité avant de regarder la variable d'environnement `PERLDB_OPTS`. Par exemple, vous pourriez construire des alias en ajoutant des entrées dans le hachage `%DB::alias`, de cette manière :

```
$alias{longueur} = 's/^longueur(.*)/p length($1)/';
$alias{stop}     = 's/^stop (at|in)/b/';
$alias{ps}       = 's/^ps\b/p scalar /';
$alias{quit}     = 's/^quit(\s*)/exit/';
$alias{aide}     = 's/^aide\s*$/|h/';
```

Vous pouvez changer les options depuis votre fichier d'initialisation en utilisant les appels de fonctions dans l'API interne du débogueur :

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

Si votre fichier d'initialisation définit la routine `afterinit`, cette fonction est appelée après la fin de l'initialisation du débogueur. Le fichier d'initialisation peut se trouver dans le répertoire courant ou dans le répertoire maison. Comme ce fichier contient du code Perl arbitraire, il doit appartenir au super-utilisateur ou à l'utilisateur courant et n'être modifiable que par son propriétaire, pour des raisons de sécurité.

Si vous voulez modifier le débogueur, copiez *perl5db.pl* depuis la bibliothèque Perl vers un autre nom et bidouillez-le selon votre inspiration. Vous voudrez alors peut-être positionner votre variable d'environnement `PERL5DB` pour écrire quelque chose comme ceci :

```
BEGIN { require "mon_perl5db.plx" }
```

En dernier ressort, vous pourriez également utiliser `PERL5DB` pour personnaliser le débogueur en positionnant directement les variables internes ou en appelant directement les fonctions internes du débogueur. Toutefois, soyez bien conscient que toutes les variables et les fonctions qui ne sont documentées ni ici, ni dans les pages de manuels en ligne *perldebug*, *perldebguts* ou *DB*, sont considérées comme étant réservées à un usage interne et sont susceptibles d'être modifiées sans préavis.

Options du débogueur

Le débogueur possède de nombreuses options que vous pouvez positionner avec la commande `O`, soit interactivement, soit depuis l'environnement, soit depuis un fichier d'initialisation.

`recallCommand, ShellBang`

Les caractères utilisés pour rappeler une commande ou pour engendrer un shell. Par défaut, les deux options sont positionnées à `!`.

`pager`

Le programme utilisé pour afficher les commandes envoyées sur un *pager* (celles commençant par un caractère `|`). Par défaut, `$ENV{PAGER}` sera utilisé. Comme le débogueur utilise les caractéristiques de votre terminal courant pour les caractères gras et le soulignement, si le *pager* choisi ne transmet pas sans changements les séquences d'échappement, l'affichage de certaines commandes du débogueur ne sera plus lisible après leur passage dans le *pager*.

`tkRunning`

Exécute sous le module `Tk` lors de l'invite (avec `ReadLine`).

`signalLevel, warnLevel, dieLevel`

Positionne le niveau de verbosité. Par défaut, le débogueur laisse tranquille vos exceptions et vos avertissements car leur altération peut empêcher le bon fonctionnement des programmes.

Pour désactiver ce mode sécurisé par défaut, placez ces valeurs à une valeur supérieure à 0. À un niveau de 1, vous obtenez une trace arrière pour tous les types d'avertissements (c'est souvent gênant) ou toutes les exceptions (c'est souvent précieux). Malheureusement, le débogueur ne peut pas discerner les exceptions fatales

et non fatales. Si `dieLevel` vaut 1, alors vos exceptions non fatales sont aussi tracées et altérées sans cérémonie si elles proviennent de chaînes évaluées (avec `eval`) ou d'un `eval` quelconque à l'intérieur des modules que vous essayez de charger. Si `dieLevel` est à 2, le débogueur ne se soucie pas de leur provenance : il usurpe vos gestionnaires d'exceptions et affiche une trace, puis modifie toutes les exceptions avec ses propres embellissements. Ceci peut être utile dans une optique de traçage mais tend à désespérément semer la confusion dans tout programme prenant au sérieux sa gestion des exceptions.

Le débogueur essayera d'afficher un message lorsque des signaux `INT`, `BUS` ou `SEGV` arriveront sans être traités. Mais si vous vous trouvez dans un appel système lent (comme un `wait`, un `accept` ou un `read` depuis le clavier ou depuis une socket) et si vous n'avez pas positionné votre propre gestionnaire pour `$SIG{INT}`, vous ne serez alors pas capable de faire un Ctrl-C pour revenir dans le débogueur car le propre gestionnaire du débogueur pour `$SIG{INT}` ne comprend pas qu'il doit lever une exception et sortir des appels système lents par un *longjmp*(3).

`AutoTrace`

Positionne le mode trace (similaire à la commande `t` mais peut être mise dans `PERLDB_OPTS`).

`LineInfo`

Affecte le fichier ou le pipe dans lequel écrire les informations sur les numéros de ligne. S'il s'agit d'un pipe (mettons, `|visual_perl_db`), alors un message court est utilisé. C'est le mécanisme mis en oeuvre pour interagir avec un éditeur esclave ou un débogueur visuel, comme les points d'entrées spéciaux de *vi* ou d'*emacs* ou le débogueur graphique *ddd*.

`inhibit_exit`

Si cette option vaut 0, elle permet le passage direct à la fin du script.

`PrintRet`

Affiche la valeur de retour après la commande `r` si cette option est activée (ce qui est le cas par défaut).

`ornaments`

Affecte l'apparence à l'écran de la ligne de commande (voir les documentations en ligne de `Term::ReadLine`). Il n'y a actuellement aucun moyen de désactiver les ornements, ce qui peut rendre certains affichages illisibles sur certains écrans ou avec certains *pagers*. Ceci est considéré comme un bogue.

`frame`

Affecte l'affichage des messages à l'entrée et à la sortie des sous-programmes. Si `frame & 2` est faux, les messages ne sont affichés qu'à l'entrée. (L'affichage de messages à la sortie pourrait s'avérer utile s'ils sont entrecroisés avec d'autres messages).

Si `frame & 4` est vrai, les arguments des fonctions sont affichés en plus d'informations sur le contexte et sur l'appelant. Si `frame & 8` est vrai, les `FETCH` surchargés, chaînifiés (`stringify`) et liés (avec `tie`) sont activés sur les arguments affichés. Si `frame & 16` est vrai, la valeur de retour du sous-programme est affichée.

La longueur à partir de laquelle la liste d'arguments est tronquée est régie par l'option suivante.

`maxTraceLen`

La longueur à laquelle la liste d'arguments est tronquée lorsque le bit 4 de l'option `frame` est positionné.

Les options suivantes affectent ce qui se produit avec les commandes `V`, `X`, et `x` :

`arrayDepth, hashDepth`

Affiche seulement les *n* premiers éléments. Si *n* est omis, tous les éléments seront affichés.

`compactDump, veryCompact`

Change le style de sortie des tableaux et des hachages. Si `compactDump` est activé, les tableaux courts peuvent être affichés sur une seule ligne.

`globPrint`

Afficher le contenu des typeglobs.

`DumpDBFiles`

Affiche les tableaux contenant les fichiers débogués.

`DumpPackages`

Affiche les tables de symboles des paquetages.

`DumpReused`

Affiche le contenu des adresses « réutilisées ».

`quote, HighBit, undefPrint`

Change le style d'affichage des chaînes. La valeur par défaut de `quote` est `auto` ; vous pouvez activer le format avec des guillemets ou avec des apostrophes en la positionnant à `"` ou `'`, respectivement. Par défaut, les caractères dont le bit de poids fort est positionné sont affichés tels quels.

`UsageOnly`

Au lieu de montrer le contenu des variables de paquetage, vous obtenez lorsque cette option est activée, un affichage rudimentaire de l'usage de la mémoire par paquetage, basé sur la taille totale des chaînes trouvées dans les variables du paquetage. Puisque la table de symboles du paquetage est utilisée, les variables lexicales sont ignorées.

Exécution autonome

Pendant le démarrage, les options sont initialisées à partir de `$ENV{PERLDB_OPTS}`. Vous pouvez y placer les options d'initialisation `TTY`, `noTTY`, `ReadLine` et `NonStop`.

Si votre fichier d'initialisation contient :

```
parse_options("NonStop=1 LineInfo=tperl.out AutoTrace");
```

alors votre script s'exécutera sans intervention humaine, plaçant les informations de trace dans le fichier *tperl.out* (Si vous l'interrompez, vous avez intérêt à réinitialiser `LineInfo` à */dev/tty* si vous voulez voir quelque chose.)

Les options suivantes peuvent être spécifiées uniquement au démarrage. Pour les positionner dans votre fichier d'initialisation, appelez `parse_options("`*`OPT=VAL`*`")`.

`TTY`

Le terminal à utiliser pour les entrées/sortie de débogage.

`noTTY`

Si cette option est positionnée, le débogueur entre en mode `NonStop` et ne se connectera pas à un terminal. En cas d'interruption (ou si le contrôle passe au débogueur *via* un réglage explicite de `$DB::signal` ou de `$DB::single` depuis le programme Perl), il se connecte au terminal spécifié par l'option `TTY` au démarrage ou à un terminal trouvé à l'exécution en utilisant le module `Term::Rendezvous` de votre choix.

Ce module devrait implémenter une méthode appelée `new`, renvoyant un objet contenant deux méthodes : `IN` et `OUT`. Celles-ci devraient renvoyer deux handles de fichiers à utiliser que le débogueur utiliserait comme son entrée et sa sortie, respectivement. La méthode `new` devrait inspecter un argument contenant la valeur de `$ENV{PERLDB_NOTTY}` au démarrage ou de `"/tmp/perldbtty$$"` autrement. On n'inspecte pas l'appartenance appropriée ni l'accès grand ouvert en écriture de ce fichier, des risques de sécurité sont donc théoriquement possibles.

`ReadLine`

Si cette option est fausse, l'implémentation de `ReadLine` dans le débogueur est désactivée de façon à pouvoir déboguer les applications utilisant elles-mêmes le module `ReadLine`.

`NonStop`

Si cette option est mise, le débogueur entre en mode non interactif jusqu'à ce qu'il soit interrompu ou que votre programme positionne `$DB::signal` ou `$DB::single`.

Les options peuvent parfois être abrégées de manière unique par leur initiale mais nous vous recommandons de toujours les épeler en entier, pour la lisibilité et la compatibilité future.

Voici un exemple de l'utilisation de la variable d'environnement `$ENV{PERLDB_OPTS}` pour positionner les options automatiquement.[2] Il lance votre programme de façon non interactive, affichant les infos à chaque entrée dans un sous-programme et pour chaque ligne exécutée. La sortie de la trace du débogueur est placée dans le fichier *tperl.out*. Ceci permet à votre programme de continuer à utiliser son entrée et sa sortie standards, sans que les informations de trace se mettent sur son chemin.

```
$ PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out" perl -d
mon_prog
```

Si vous interrompez le programme, vous aurez besoin de réinitialiser rapidement `O LineInfo=/dev/tty` ou à autre chose de sensé sur votre plate-forme. Sinon, vous ne verrez plus l'invite du débogueur.

2. Nous utilisons la syntaxe du shell *sh* pour montrer la configuration des variables d'environnement. Les utilisateurs d'autres shells feront les adaptations en conséquence.

Implémentation du débogueur

Perl fournit des points d'entrée spéciaux pour le débogage, à la fois à la compilation et à l'exécution pour créer des environnements de débogage, tels que le débogueur standard. Ces points d'entrées ne doivent pas être confondus avec les options `perl -D`, qui ne sont utilisables que si votre version de Perl a été compilée avec l'implémentation **`-DDEBUGGING`**.

Par exemple, à chaque fois que vous appelez la fonction interne de Perl `caller`, depuis le paquetage `DB`, les arguments avec lesquels l'enregistrement d'activation correspondant avait été appelé, sont copiés dans le tableau `@DB::args`. Lorsque vous invoquez Perl avec l'option **`-d`**, les fonctionnalités supplémentaires suivantes sont activées :

- Perl insert le contenu de `$ENV{PERL5DB}` (ou, si cette variable est absente, `BEGIN {require 'perl5db.pl'}`) avant la première ligne de votre programme.
- Le tableau `@{"_<$nom_fichier"}` contient les lignes de `$nom_fichier` pour tous les fichiers compilés par Perl. Il en va de même pour les chaînes évaluées (avec `eval`) qui contiennent des sous-programmes ou sont actuellement en cours d'exécution. Le `$nom_fichier` pour les chaînes évaluées ressemble à `(eval 34)`. Les affectations de code dans les expressions régulières ressemblent à `(re_eval 19)`.
- Le hachage `%{"_<$nom_fichier"}` contient les points d'arrêt et les actions avec le numéro de la ligne comme clef. Vous pouvez positionner des entrées individuelles, par opposition au hachage complet. Perl ne se soucie ici que de la vérité booléenne, bien que les valeurs utilisées par *perl5db.pl* se présentent sous la forme `"$condition_d_arret\0$action"`. Les valeurs dans ce hachage sont magiques dans un contexte numérique : elles valent zéro si l'on ne peut pas placer de point d'arrêt sur la ligne.

  Le même hachage renferme les chaînes évaluées qui contiennent des sous-programmes ou sont actuellement en train d'être exécutées. Le `$nom_fichier` pour les chaînes évaluées ressemble à `(eval 34)` ou `(re_eval 19)`.
- Le scalaire `${"_<$nom_fichier}` contient `"_<$nom_fichier"`. Ceci est également le cas pour les chaînes évaluées qui contiennent des sous-programmes ou sont actuellement en cours d'exécution. Le `$nom_fichier` pour les chaînes évaluées ressemble à `(eval 34)` ou `(re_eval 19)`.
- Après que chaque fichier chargé par `require` est compilé mais avant qu'il soit exécuté, `DB::postponed(*{"_<$nom_fichier"})` est appelé si le sous-programme `DB::postponed` existe. Ici, le `$nom_fichier` est le nom développé du fichier chargé par `require`, tel qu'on le trouve dans les valeurs de `%INC`.
- Après que chaque sous-programme *`nom_sous_programme`* est compilé, l'existence de `$DB::postponed{`*`nom_sous_programme`*`}` est vérifiée. Si cette clef existe, `DB::postponed(`*`nom_sous_programme`*`)` est appelé si le sous-programme `DB::postponed` existe également.
- Un hachage `%DB::sub` est maintenu, dont les clefs sont les noms des sous-programmes et dont les valeurs se présentent sous la forme *`nom_fichier`*`:`*`ligne_debut`*`-`*`ligne_fin`*. *`nom_fichier`* se présente sous la forme `(eval 34)` pour les sous-programmes définis à l'intérieur de chaînes évaluées ou `(re_eval 19)` pour ceux définis à l'intérieur d'affectations de code dans les expressions régulières.

- Lorsque l'exécution de votre programme atteint un certain point qui pourrait contenir un point d'arrêt, le sous-programme `DB::DB()` est appelé si l'une des variables `$DB::trace`, `$DB::single` ou `$DB::signal` est vraie. Ces variables ne peuvent être affectées par `local`. Cette fonctionnalité est désactivée lorsque l'exécution se déroule à l'intérieur de `DB::DB()`, y compris les fonctions appelées depuis là, sauf si `$^D & (1< <30)` est vrai.
- Lorsque l'exécution du programme atteint un appel de sous-programme, un appel à `&DB::sub(`*`args`*`)` est effectué à la place, avec `$DB::sub` contenant le nom du sous-programme appelé. Ceci ne se produit pas si le sous-programme avait été compilé dans le paquetage `DB`.

Remarquez que si `&DB::sub` a besoin de données externes pour fonctionner, aucun appel de sous-programme n'est possible jusqu'à ce que ceci soit accompli. Pour le débogueur standard, la variable `$DB::deep` (à combien de niveaux de profondeur de récursion pouvez-vous aller avant d'arrêter obligatoirement) donne un exemple d'une telle dépendance.

Écriture de son propre débogueur

Un débogueur minimal en état de fonctionner consiste en une ligne :

```
sub DB::DB {}
```

qui, puisqu'il ne fait rien de particulier, peut facilement être défini *via* la variable d'environnement `PERL5DB` :

```
% PERL5DB="sub DB::DB {}" perl -d votre_programme
```

Un autre débogueur minuscule, légèrement plus utile, pourrait être créé comme ceci :

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

Ce petit débogueur imprimerait le numéro de ligne séquentiel à chaque instruction rencontrée et attendrait que vous entriez un retour à la ligne avant de continuer.

Le débogueur suivant, bien qu'il paraisse petit, est vraiment tout à fait fonctionnel :

```
{
    package DB;
    sub DB {}
    sub sub {print ++$i, " $sub\n"; &$sub}
}
```

Il affiche le numéro séquentiel de l'appel de sous-programme et le nom du sous-programme appelé. Remarquez que `&DB::sub` doit être compilé dans le paquetage `DB`, tel que nous l'avons fait ici.

Si vous basez votre nouveau débogueur sur le débogueur courant, il y a quelques points d'entrée qui peuvent vous aider à le personnaliser. Au démarrage, le débogueur lit votre fichier d'initialisation dans le répertoire courant ou dans votre répertoire maison. Après que le fichier est lu, le débogueur lit la variable d'environnement `PERLDB_OPTS` et parcourt celle-ci comme le reliquat d'une ligne `O ...`, telle que vous l'auriez entrée à l'invite du débogueur.

Le débogueur maintient également des variables internes magiques, comme `@DB::dbline` et `%DB::dbline`, qui sont des alias pour `@{"::_<`*`fichier_courant`*`"}` et

`%{"::_<`*`fichier_courant`*`"}`. Ici, *`fichier_courant`* est le fichier actuellement sélectionné, soit choisi explicitement avec la commande `f` du débogueur, soit implicitement de par le flux d'exécution.

Certaines fonctions peuvent faciliter la personnalisation. `DB::parse_options(`*`CHAÎNE`*`)` analyse une ligne comme l'option `O`. `DB::dump_trace(`*`SAUT`*`[,`*`COMPTEUR`*`])` saute le nombre spécifié d'enregistrements d'activation dans la pile (*stack frames*) et renvoie une liste contenant des informations à propos des enregistrements d'activation appelants (tous, si *`COMPTEUR`* est omis). Chaque entrée est une référence vers un hachage avec les clefs « `context` » (`.`, `$` ou `@`), « `sub` » (nom du sous-programme ou informations sur un `eval`), « `args` » (`undef` ou une référence vers un tableau), « `file` » (nom du fichier) et « `line` » (numéro de ligne). `DB::print_trace(`*`HF`*`,`*`SAUT`*`[,`*`COMPTEUR`*`[,`*`COURT`*`]])` imprime des informations formatées, concernant les enregistrements d'activation appelants de la pile, dans le handle de fichier donné. Les deux dernières fonctions peuvent s'avérer pratiques en tant qu'arguments des commandes `<` et `<<` du débogueur.

Vous n'avez pas besoin d'apprendre tout cela — la plupart d'entre nous ne l'ont pas fait. En fait, lorsque nous avons besoin de déboguer un programme, nous ne faisons généralement qu'insérer quelques instructions `print` ici et là avant de relancer le programme.

Dans des jours meilleurs, nous nous rappellerons qu'il faut activer les avertissements en premier. Cela fait souvent disparaître le problème, en évitant de nous arracher les cheveux (pour ce qu'il en reste). Mais lorsque cela ne fonctionne pas, il est bon de savoir, en vous attendant patiemment derrière cette bonne vieille option **`-d`**, qu'il existe un adorable petit débogueur qui peut raccommoder n'importe quoi *sauf* trouver les bogues à votre place.

Mais si vous devez vous rappeler une seule chose à propos de la personnalisation du débogueur, c'est peut-être celle-ci : ne restreignez pas la notion de bogue aux choses qui rendent Perl malheureux. Il s'agit également d'un bogue si votre programme *vous* rend malheureux. Plus tôt, nous vous avons montré quelques débogueurs personnalisés très simples. Dans le prochain paragraphe, nous allons vous montrer un exemple d'une différente sorte de débogueur personnalisé, une sorte qui peut (ou peut ne pas) vous aider à déboguer le bogue connu sous le nom de « Est-ce que ce truc va s'arrêter un jour ? ».

Le profileur Perl

Voulez-vous rendre votre programme plus rapide ? La réponse est oui, bien entendu. Mais vous devriez d'abord vous arrêter est vous demander, « Ai-je vraiment besoin de passer du temps à rendre ce programme plus rapide ? » L'optimisation récréative peut être amusante,[3] mais normalement, vous avez mieux à faire de votre temps. Parfois, vous n'avez besoin que de planifier votre travail et de lancer le programme pendant que vous allez boire un café. (Ou vous prenez ceci comme excuse pour en boire un.) Mais si votre programme a absolument besoin de tourner plus rapidement, vous devriez commencer par le profiler. Un profileur peut vous dire quelles parties de votre programme prennent le plus de temps à s'exécuter, afin que vous ne perdiez pas de temps à optimiser un sous-programme dont les effets sont insignifiants par rapport au temps d'exécution total.

3. Enfin, d'après Nathan Torkington, qui a contribué à la version originale de ce paragraphe.

Perl est livré avec un profileur, le module `Devel::DProf`. Vous pouvez l'utiliser pour profiler le programme Perl de *mon_code.plx* en tapant :

```
perl -d:DProf mon_code.plx
```

Même si nous l'avons appelé un profileur — puisque c'est ce qu'il fait — le mécanisme employé par `DProf` est exactement le même que celui dont nous avons parlé auparavant dans ce chapitre. `DProf` n'est qu'un débogueur qui enregistre le temps entre le moment où Perl entre dans chaque sous-programme et celui où il en sort.

Lorsque votre script profilé se terminera, `Dprof` déversera les informations mesurant le temps dans un fichier nommé *tmon.out*. Le programme *dprofpp*, livré avec Perl, sait comment analyser *tmon.out* et produit un rapport. Vous pouvez également utiliser *dprofpp* comme frontal pour le processus complet avec l'option **`-p`** (voir la description plus loin).

Étant donné ce programme :

```
exterieur();

sub exterieur {
    for (my $i=0; $i < 100; $i++) { interieur() }
}

sub interieur {
    my $total = 0;
    for (my $i=0; $i < 1000; $i++) { $total += $i }
}

interieur();
```

la sortie de *dprofpp* est :

```
Total Elapsed Time = 0.158572 Seconds
  User+System Time = 0.158572 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 88.2   0.140  0.140    101   0.0014 0.0014 main::interieur
 0.00   0.000  0.139      1   0.0000 0.1393 main::exterieur
```

Remarquez que la somme des pourcentages n'est pas égale à 100. En fait, dans ce cas, elle en est assez loin, ce qui devrait être un signe comme quoi vous avez besoin de lancer le programme plus longtemps. En règle générale, plus vous pouvez collecter de données de profilage, meilleur sera votre échantillon statistique. Si nous augmentons la boucle extérieure pour tourner 1000 fois au lieu de 100, nous obtiendrons des résultats plus justes :

```
Total Elapsed Time = 1.403988 Seconds
  User+System Time = 1.373988 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 98.2   1.350  1.348   1001   0.0013 0.0013 main::interieur
 0.58   0.008  1.357      1   0.0085 1.3570 main::exterieur
```

La première ligne rapporte combien de temps le programme a mis à s'exécuter, du début à la fin. La seconde ligne affiche la somme de deux nombres différents : le temps pris pour exécuter votre code (« *user* », utilisateur) et le temps pris par le système d'ex-

ploitation à exécuter les appels systèmes faits par votre code (« *system* », système). (Nous devrons admettre un peu de mauvaise précision dans ces nombres — l'horloge de l'ordinateur n'a certainement pas des tics d'un millionième de seconde. Elle peut avoir des tics d'un centième de seconde si vous avez de la chance.)

Les temps « *user+system* » peuvent être changés avec les options de la ligne de commande de *dprofpp*. **-r** affiche le temps écoulé, **-s** le temps système uniquement et **-u** le temps utilisateur seulement.

Le reste du rapport est une analyse du temps passé dans chaque sous-programme. La ligne « *Exclusive Times* » (Temps exclusifs) indique que lorsque le sous-programme `exterieur` a appelé le sous-programme `interieur`, le temps passé dans `interieur` n'a pas compté dans le calcul du temps de `exterieur`. Pour changer ceci et faire que le temps de `interieur` soit comptabilisé dans celui d'`exterieur`, donnez l'option **-I** à *dprofpp*.

Pour chaque sous-programme, les informations suivantes sont rapportées : `%Time`, le pourcentage de temps passé dans l'appel à ce sous-programme ; `ExclSec`, le temps en secondes passé dans ce sous-programmes, en n'incluant pas les sous-programmes appelés depuis celui-ci ; `CumulS`, le temps en secondes passé dans ce sous-programme et dans ceux appelés depuis celui-ci ; `#Calls`, le nombre d'appels du sous-programme ; `sec/call`, le temps moyen en secondes de chaque appel au sous-programme, en n'incluant pas les sous-programmes appelés depuis celui-ci ; `Csec/c`, le temps moyen en secondes de chaque appel au sous-programme et de ceux appelés depuis celui-ci.

De toutes ces valeurs, la plus utile est `%Time`, qui vous dira où passe votre temps. Dans notre cas, le sous-programme `interieur` prend le plus de temps, nous devrions donc essayer de l'optimiser ou de trouver un algorithme qui l'appellera moins souvent. `:-)`

Les options de *dprofpp* fournissent un accès à d'autres informations ou modifient la manière dont les temps sont calculés. Vous pouvez également pousser *dprofpp* à lancer en premier lier le script à votre place, afin que vous n'ayez pas à vous souvenir de l'option **-d:DProf** :

-p *SCRIPT*

Dit à *dprofpp* qu'il devrait profiler le *SCRIPT* donné et ensuite interpréter ses données de profilage. Voir également l'option **-Q**.

-Q

Utilisée avec **-p** pour dire à *dprofpp* de sortir après avoir profilé le script, sans interpréter les données.

-a

Trie la sortie dans l'ordre alphabétique des noms de sous-programme plutôt que dans l'ordre décroissant des pourcentages de temps.

-R

Compte séparément les sous-programmes anonymes définis dans le même paquetage. Le comportement par défaut est de compter tous les sous-programmes anonymes en un seul, appelé `main::__ANON__`.

-I

Affiche les temps des sous-programmes en incluant les temps des sous-programmes fils.

-l

Trie les sous-programmes par nombre d'appels. Ceci peut faciliter la sélection de candidats à l'inclusion par référence (*inlining*).

-O *COMPTEUR*

Montre seulement le top *COMPTEUR* des sous-programmes. La valeur par défaut est 15.

-q

N'affiche pas les en-têtes des colonnes.

-T

Affiche l'arbre d'appel des sous-programmes sur la sortie standard. Les statistiques des sous-programmes ne sont pas affichées. Une fonction appelée plusieurs fois (consécutivement) au même niveau d'appel, ne sera affichée qu'une seule fois, avec un compteur de répétition.

-S

Produit un affichage structuré par la manière dont vos sous-programmes s'appellent l'un, l'autre :

```
main::interieur x 1       0.00s
main::exterieur x 1       1.35s = (0.01 + 1.35)s
  main::interieur x 1000  1.35s
```

Lisez ceci ainsi : le niveau supérieur de votre programme a appelé `interieur` une fois et pendant une durée de 0.00 s et le niveau supérieur a appelé `exterieur` une fois et il a tourné pendant 1.35 s inclusivement (0.01 s dans `exterieur` lui-même, 1.35 s dans les sous-programmes appelés depuis `exterieur`) en appelant `interieur` 1000 fois (ce qui a pris 1.35 s). Waouh, compris ?

Les branches du même niveau (par exemple, `interieur` appelé une fois et `exterieur` appelé une fois) sont triées dans l'ordre des temps inclusifs.

-U

Ne trie pas. Affiche dans l'ordre trouvé dans le fichier de profilage tel quel.

-v

Trie dans l'ordre des temps moyens passés dans les sous-programmes à chaque appel. Ceci peut faciliter l'identification de candidats pour une optimisation à la main en incluant par référence (*inlining*) le corps des sous-programmes.

-g *SOUS_PROGRAMME*

Ignore les sous-programmes sauf *SOUS_PROGRAMME* et tous ceux qui sont appelés depuis celui-ci.

D'autres options sont décrites dans *dprofpp*(1), sa page de manuel standard.

`DProf` n'est pas votre seul choix de profileur. CPAN contient également `Devel::SmallProf`, qui rapporte les temps passés dans chaque ligne de votre programme. Cela peut vous aider à vous faire une idée si vous utiliser une construction Perl particulière qui s'avère étonnamment coûteuse. La plupart des fonctions internes sont assez efficaces mais il est facile d'écrire accidentellement une expression régulière dont le temps systè-

me (*overhead*) croît exponentiellement avec la taille de l'entrée. Voir également le paragraphe *Efficacité* au chapitre 24, *Techniques couramment employées*, pour d'autres astuces utiles.

Maintenant, allez boire un café. Vous en aurez besoin pour le prochain chapitre.

21

Mécanismes internes et accès externes

Comme nous l'avons expliqué au chapitre 18, *Compilation*, *perl* (le programme) contient à la fois un compilateur et un interpréteur pour les programmes écrits en Perl (le langage). Le compilateur/interpréteur Perl est lui-même écrit en C. Dans ce chapitre, nous allons esquisser la manière dont les programmes en C fonctionnent si l'on se place dans la perspective de quelqu'un voulant soit étendre, soit insérer Perl. Lorsque vous *étendez* Perl, vous mettez un bout de code C (appelé l'*extension*) sous le contrôle de Perl et lorsque vous *insérez* Perl, vous mettez un interpréteur Perl[1] sous le contrôle d'un programme C plus large.

La couverture succinte que nous faisons ici ne remplace en aucun cas les documentations en ligne sur les tripes de Perl : *perlguts*, *perlxs*, *perlxstut*, *perlcall*, *perlapi* et *h2xs*, toutes livrées avec Perl. Encore une fois, à moins que vous n'étendiez ou insériez Perl, vous n'aurez jamais besoin de connaître tout ceci.

En supposant que vous avez besoin de savoir, ce que vous devez savoir en premier est un peu des entrailles de Perl. Vous aurez également besoin de connaître C pour la plupart de ce qui suit. Vous aurez besoin d'un compilateur C pour lancer les exemples. Si votre but final est de créer un module dont les autres gens se serviront, ils auront également besoin d'un compilateur C. Plusieurs de ces exemples ne tourneront que sur les systèmes ressemblant à Unix. Oh, et ces informations sont susceptibles de changer dans les futures versions de Perl.

En d'autres termes, nous pénétrons en *terra incognita*, au risque de rencontrer des monstres marins.

1. Alors que nous prenons soin de distinguer le compilateur de l'interpréteur lorsque cette distinction est importante, cela devient un peu ennuyeux de persister à dire « compilateur/interpréteur », nous raccourcissons donc cela souvent en « interpréteur » pour signifier la globalité du code et des données C fonctionnant comme une instance de *perl* (le programme) ; lorsque vous incluez du Perl, vous pouvez avoir plusieurs instances de l'interpréteur mais chacune se comporte en tant que son propre petit *perl*.

Comment fonctionne Perl

Lorsqu'on alimente le compilateur Perl avec un programme Perl, la première tâche qu'il effectue est l'*analyse lexicale* : décomposer le programme en ses éléments syntaxiques de base (souvent appelés *tokens*). Si le programme est :

```
print "Salut, tout le monde !\n";
```

l'analyseur lexical le décompose en trois tokens : `print`, `"Salut, tout le monde !\n"` et le point-virgule final. La séquence de tokens est alors *analysée syntaxiquement*, en fixant les relations entre les tokens. En Perl, la frontière entre l'analyse lexicale et syntaxique est encore plus floue que dans les autres langages. (C'est-à-dire, les autres langages informatiques. Si vous pensez à toutes les significations que `new Bestiau` peut prendre selon qu'il s'agit d'un paquetage Bestiau ou d'un sous-programme appelé `new`, vous comprendrez alors pourquoi. D'un autre côté, nous passons notre temps à lever les ambiguïtés de ce genre en français.)

Une fois qu'un programme a été analysé syntaxiquement et compris (vraisemblablement), il est compilé dans un arbre de *codes d'opérations* (*opcodes*), représentant les opérations de bas niveau et finalement, cet arbre d'opérations est exécuté — sauf si vous aviez invoqué Perl avec l'option **`-c`** (« contrôle de la syntaxe »), qui sort après avoir terminé la phase de compilation. C'est pendant la compilation, et non pendant l'exécution, que les blocs `BEGIN`, les blocs `CHECK` et les instructions `use` sont exécutés.

Types de données internes

Pendant que l'arbre de codes d'opérations, constituant un programme Perl, est exécuté, des valeurs Perl sont créées, manipulées et détruites. Les types de données avec lesquels vous êtes familier en Perl ont tous leur types de données correspondant dans le C que Perl a sous le capot et vous aurez besoin de connaître ces types lorsque vous passerez des données entre les deux langages.

Trois *typedefs* (définitions de type) C correspondent au trois types de données de base de Perl : `SV` (valeur scalaire), `AV` (valeur tableau) et `HV` (valeur hachage). De plus, `IV` est un type simple entier signé dont on est sûr qu'il est assez grand pour contenir soit un pointeur, soit un entier ; et `I32` et `I16` sont des types dont est sûr qu'ils sont assez grands pour contenir respectivement 32 et 16 bits. Pour stocker les versions non signées des ces trois derniers typedefs, il existe également `UV`, `U32` et `U16`. Tous ces typedefs peuvent être manipulés avec les fonctions C décrites dans la documentation de *perlguts*. Nous esquissons le comportement de certaines de ces fonctions ci-dessous :

- Il existe quatre types de valeurs pouvant être copiées dans un `SV` : une valeur entière (`IV`), un double (`NV`), une chaîne (`PV`) et un autre scalaire (`SV`). Il existe des douzaines de fonctions pour les `SV`, pour vous permettre de créer, de modifier, d'agrandir et de vérifier la vérité booléenne ou le fait que les scalaires Perl qu'ils représentent soient définis. Les références Perl sont implémentées en tant que `RV`, un type spécial de `SV`.
- Lorsqu'un `AV` est créé, il peut être créé vide ou peuplé de `SV`, ce qui est sensé puisqu'un tableau est une collection de scalaires.
- Le `HV` possède des fonctions C associées pour stocker, récupérer, supprimer et vérifier l'existence de paires clef/valeur dans le hachage que le `HV` représente.

- Il existe également un `GV` (valeur glob), pouvant contenir des références vers n'importe laquelle des valeurs associées à un identificateur de variable : une valeur scalaire, un valeur tableau, une valeur hachage, un sous-programme, un handle d'entrée/sortie ou un format.

Lorsque vous étendez Perl, vous aurez parfois besoin de connaître ces valeurs lorsque vous créerez les liaisons vers les fonctions C. Lorsque vous insérez Perl, vous aurez besoin de connaître ces valeurs lorsque vous échangerez des données avec l'interpréteur Perl inclus dans votre programme C.

Étendre Perl (utiliser du C depuis du Perl)

Si vous voulez utiliser du code source C (ou une bibliothèque C) depuis Perl, vous devez créer une bibliothèque qui puisse être chargée dynamiquement ou avec laquelle vous puissiez faire une édition de liens statique avec l'exécutable *perl*. (On préfère d'ordinaire le chargement dynamique pour minimiser le nombre des différents exécutables *perl* qui prennent place en étant différents.) Vous créez cette bibliothèque en créant un fichier *XS* (avec un suffixe *.xs*) contenant une série de sous-programmes d'enveloppe (*wrapper*). Les sous-programmes d'enveloppe ne sont toutefois pas des sous-programmes Perl ; ils sont écrits en langage XS et nous appelons un tel sous-programme, une *XSUB* (pour « *eXternal SUBroutine* », routine externe). Une XSUB peut envelopper une fonction C depuis une bibliothèque externe, une fonction C ailleurs dans le fichier XS ou du code C brut dans la XSUB elle-même. Vous pouvez alors utiliser l'utilitaire *xsubpp*, livré avec Perl pour prendre le fichier XS et le traduire en code C qui puisse être compilé dans une bibliothèque que Perl comprendra.

En supposant que votre système d'exploitation implémente l'édition de liens dynamique, le résultat final sera un module Perl se comportant comme tout autre module écrit à 100 % en Perl pur mais contenant sous le capot du code C compilé. Il fait ceci en enlevant des arguments depuis la pile d'arguments de Perl, en convertissant les valeurs Perl vers les formats attendus par une fonction C particulière (spécifiés à travers une déclaration de XSUB), en appelant la fonction C et en retransférant finalement les valeurs renvoyées par la fonction C vers Perl. Ces valeurs de retour peuvent être repassées à Perl soit en les mettant dans la pile de Perl, soit en modifiant les arguments fournis depuis Perl. (Si votre système n'implémente pas l'édition de liens dynamique, il existe un autre cerceau dans lequel vous devrez sauter et nous en parlerons dans le prochain paragraphe.)

La description précédente est en quelque sorte une vision simplifiée de ce qui se passe réellement. Puisque Perl autorise des conventions d'appels plus flexibles que C, les XSUB peuvent en faire bien plus en pratique, comme vérifier la validité des paramètres d'entrée, lever des exceptions, renvoyer `undef` ou une liste vide, appeler différentes fonctions C en se basant sur le nombre et le type des arguments ou fournir une interface orientée objet. Encore une fois, voir les pages de manuel de *perlxs* et *perlxstut*.

XS et XSUB

XS est un moyen pratique : il n'y a rien qui vous empêche d'écrire du code de glu directement en C et de le lier dans votre exécutable Perl. Toutefois, ceci serait pénible, parti-

culièrement si vous devez écrire la glu pour de multiples fonctions C ou si vous n'êtes pas familier avec la discipline de la pile de Perl et autres mystères. XS vous permet d'écrire une description concise de ce qui devrait être fait par la glu et le compilateur XS, *xsubpp*, s'occupe du reste.

Pour les gens qui ne trouvent pas XS assez pratique, le système SWIG génère automatiquement des XSUB simples. Voir *http://www.swig.org* pour plus d'informations.

Le langage XS vous permet de décrire la correspondance entre une fonction C et une fonction Perl. Il vous permet également décrire une fonction Perl qui soit une enveloppe autour de code C pur que vous écrivez vous-même. Lorsque XS est utiliser uniquement pour faire la correspondance entre C et Perl, la déclaration d'une XSUB est quasiment identique à la déclaration d'une fonction C. Dans de telles circonstances, un outils appelé *h2xs* (livré avec Perl) est capable de traduire un fichier d'en-tête C entier vers un fichier XS correspondant, fournissant la glu vers les fonctions C et les macros.

L'outil *xsubpp* crée les constructions nécessaires pour permettre à une XSUB de manipuler des valeurs Perl et la glu nécessaire pour permettre à Perl d'appeler la XSUB.

Un fichier XS commence par le code C que vous voulez inclure, ce qui n'est souvent rien de plus qu'un ensemble de directives `#include`. Après un mot-clé `MODULE`, le reste du fichier devrait être dans le « langage » XS, une combinaison de directives XS et de définitions de XSUB. Nous verrons bientôt l'exemple d'un fichier XS complet mais en attendant, voici la définition d'une XSUB simple, permettant à un programme Perl d'accéder à une fonction de bibliothèque C, appelée *sin*(3). La XSUB spécifie le type renvoyé (un nombre en virgule flottante à double précision), le nom de la fonction et une liste d'argumentd (avec un argument traduit `x`), ainsi que le type de l'argument (un autre double) :

```
double
sin(x)
    double x
```

Les XSUB plus compliquées contiendront souvent d'autres morceaux de code XS. Chaque section d'une XSUB commence par un mot-clé suivi d'un deux-points, tel que `INIT:` ou `CLEANUP:`. Cependant, les deux premières lignes d'une XSUB contiennent toujours les mêmes données : la description du type renvoyé et le nom de la fonction avec ses paramètres. Tout ce qui suit immédiatement ceci est considéré comme étant une section `INPUT:` sauf si on le marque explicitement avec un autre mot-clé. Les divers mots-clés sont tous expliqués dans la page de manuel *perlxs*, que vous devriez lire pour apprendre tout ce que vous pouvez faire avec les XSUB.

Si votre système n'est pas capable de charger dynamiquement des bibliothèques partagées, vous pouvez toujours utiliser des XSUB mais vous devez faire une édition de liens statique des XSUB avec le reste de Perl, créant ainsi un nouvel exécutable Perl (qui prend place en étant différent). Le mécanisme de construction d'une XSUB vérifiera le système et construira une bibliothèque partagée si possible, sinon une bibliothèque statique. Optionnellement, il peut construire un nouvel exécutable lié statiquement avec cette bibliothèque statique incluse. Mais vous pourriez vouloir attendre pour lier toutes vos nouvelles extensions dans un seul exécutable (qui prend place en étant le même, puisque c'est ce qu'il est.)

Si votre système est capable de lier dynamiquement les bibliothèques mais que vous

voulez toujours construire un exécutable lié statiquement, vous pouvez lancer `make perl` au lieu de `make` dans les exemples suivants. Vous devrez ensuite lancer `make test_static` au lieu de `make test` pour tester votre extension.

Le programme *xsubpp* a également besoin de savoir comment convertir des types de données Perl en types de données C. Il le devine souvent mais avec des types définis par l'utilisateur, vous pouvez avoir besoin de l'aider en spécifiant la conversion dans un fichier *typemap* (littéralement : carte des types). Les conversions par défaut sont stockées dans *CHEMIN-VERS-LA-BIBLIOTHÈQUE/ExtUtils/typemap*.

Le fichier typemap est découpé en trois sections. La première, étiquetée `TYPEMAP`, dit au compilateur quels sont les fragments de code parmi ceux des deux sections suivantes devraient être utilisées pour faire la correspondance entre les types C et les valeurs Perl. La deuxième section, `INPUT`, contient du code C spécifiant comment les valeurs Perl devraient être converties en types C. La troisième section, `OUTPUT`, contient du code C spécifiant comment traduire les types C en valeurs Perl.

Création d'extensions

Une extension convenable consiste en plusieurs fichiers : l'un contenant le code XS, plus des fichiers implémentés qui aident Perl à deviner quoi faire avec le code XS. Vous pouvez créer tous ces fichiers à la main mais il est plus facile d'utiliser l'outil *h2xs*, qui crée un squelette d'extension que vous pouvez ensuite compléter :

```
h2xs -A -n Montest
```

Ceci crée un répertoire nommé *Montest*, peut-être sous *ext/* si ce répertoire existe dans le répertoire courant. Six fichiers seront créés dans le répertoire *Montest* : *MANIFEST*, *Makefile.PL*, *Montest.pm*, *Montest.xs*, *test.pl* et *Changes*. Nous décrivons les quatre premiers ci-dessous.

MANIFEST

Le fichier *MANIFEST* contient le nom de tous les fichiers qui viennent d'être créés dans le répertoire *Montest*. Si vous ajoutez d'autres fichiers dans votre extension et que vous la distribuez au monde entier, ajouter le nom des fichiers ici. Ceci est testé par certains systèmes pour s'assurer que votre distribution est complète.

Makefile.PL

Il s'agit d'un programme Perl générant un *Makefile* (qui est ensuite passé à *make* ou à un équivalent). *Makefile.PL* est décrit plus en détail dans « Créer des modules CPAN » au chapitre 22, *CPAN* .

Montest.pm

Les utilisateurs feront un `use` sur ce module lorsqu'ils voudront charger votre extension. Vous êtes supposés remplir les blancs du module squelette créé pour vous par *h2xs*[2] :

```
package Montest;
```

2. *N.d.T.* : Le squelette généré par *h2xs* contient des commentaires en anglais. Toutefois, nous les avons traduits ici pour une meilleure compréhension.

```
use strict;
use warnings;

require Exporter;
require DynaLoader;

our @ISA = qw(Exporter DynaLoader);
# Éléments à exporter par défaut dans les espaces de noms des appelants.
# Remarque : N'exportez pas des noms par défaut sans une très bonne
# raison. Utilisez plutôt EXPORT_OK. N'exportez pas bêtement toutes vos
# fonctions/méthodes/constantes.
our @EXPORT = qw(

);
our $VERSION = '0.01';

bootstrap Montest $VERSION;

# Les méthodes préchargées viennent ici.

# Les méthodes autochargées viennent après __END__ et sont traitées par
# le programme autosplit.

1;
__END__
# En-dessous se trouve une souche de documentation pour votre module.
# Vous feriez mieux de l'éditer !
```

La plupart des modules d'extension feront un `require` des extensions `Exporter` et `DynaLoader`. Après avoir alimenté `@ISA` (pour l'héritage) et `@EXPORT` (pour rendre les fonctions disponibles pour le paquetage utilisant le module), le code d'initialisation dit à Perl d'*amorcer* (`bootstrap`) le code XS. Perl lie alors dynamiquement la bibliothèque partagée dans le processus *perl* à l'exécution.

Montest.xs

Le fichier *Montest.xs* contient les XSUB disant à Perl comment passer les données aux routines C compilées. Initialement, *Montest.xs* ressemblera à quelque chose comme ceci :

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

MODULE = Montest        PACKAGE = Montest
```

Éditons le fichier XS en ajoutant ceci à la fin :

```
void
salut()
    CODE:
        printf("Salut, tout le monde !\n");
```

Lorsque vous lancez `perl Makefile.PL`, le *Makefile* dont *make* a besoin est créé :

```
% perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Montest
```

Le lancement de *make* produit maintenant un affichage qui ressemble à ceci (certaines lignes trop longues ont été raccourcies pour plus de clarté et certaines lignes accessoires ont été supprimées) :

```
% make
umask 0 && cp Montest.pm ./blib/Montest.pm
perl xsubpp -typemap typemap Montest.xs >Montest.tc && mv Montest.tc
Montest.c
cc -c Montest.c
Running Mkbootstrap for Montest ()
chmod 644 Montest.bs
LD_RUN_PATH="" ld -o ./blib/PA-RISC1.1/auto/Montest/Montest.sl -b Montest.o
chmod 755 ./blib/PA-RISC1.1/auto/Montest/Montest.sl
cp Montest.bs ./blib/PA-RISC1.1/auto/Montest/Montest.bs
chmod 644 ./blib/PA-RISC1.1/auto/Montest/Montest.bs
Manifying ./blib/man3/Montest.3
```

Nous supposons que le programme *make* que Perl utilise pour construire le programme est appelé *make*. Au lieu de lancer *make* dans ces exemple, il se peut que vous deviez substituer un autre programme *make* que Perl a été configuré pour utiliser. Vous pouvez trouver ce programme avec :

```
% perl -V:make
```

Le lancement de *make* a créé un répertoire nommé *blib* (pour « *build library* », bibliothèque de construction) dans le répertoire de travail courant. Ce répertoire contiendra la bibliothèque partagée que nous construirons. Une fois que nous sommes sûrs que nous savons ce que nous faisons, nous pouvons l'installer depuis ce répertoire vers son emplacement définitif. Jusqu'alors, nous devrons ajouter explicitement le répertoire *blib* au tableau `@INC` de Perl en utilisant le module `ExtUtils::testlib`. Si nous créons maintenant un fichier appelé *salut*, ressemblant à ceci :

```
use ExtUtils::testlib;      # ajoute les répertoires blib/* à @INC
use Montest;
Montest::salut();
```

nous pouvons creuser un passage depuis Perl vers C :

```
% perl salut
Salut, le monde !
```

Une fois que notre extension est achevée et qu'elle a passé tous ses tests, nous pouvons l'installer avec `make install`.

Vous aurez besoin des permissions en écriture pour votre bibliothèque Perl. (Si ce n'est pas le cas, vous pouvez spécifier un autre répertoire comme indiqué dans *Installation des modules de CPAN dans la bibliothèque Perl* au chapitre 22.)

Entrées et sorties des XSUB

En poursuivant l'exemple précédent, nous ajouterons une seconde XSUB, prenant un seul argument numérique en entrée et renvoyant 1 si le nombre est pair et 0 s'il est impair :

```
int
est_pair(x)
        int x
    CODE:
        RETVAL = (x %2 == 0);
    OUTPUT:
        RETVAL
```

La liste des paramètres de sortie arrive tout à la fin de la fonction, juste après la directive `OUTPUT:`. L'utilisation de `RETVAL` indique à Perl que vous souhaitez renvoyer cette valeur comme valeur de retour de la XSUB. Si nous avions voulu que la fonction modifie son paramètre d'entrée, nous aurions utilisé `x` à la place de `RETVAL`.

Nous pouvons construire à nouveau notre nouvelle bibliothèque partagée avec les mêmes étapes qu'avant, en générant un *Makefile* à partir du *Makefile.PL* et en lançant *make*.

Pour tester que notre extension fonctionne, nous créerons une suite de tests dans *test.pl*. Ce fichier est initialisé par *h2xs* pour imiter le script de test de Perl lui-même. À l'intérieur de ce script, vous pouvez lancer des tests pour confirmer que l'extension se comporte comme il faut, en affichant `ok` lorsque c'est le cas et `not ok` dans le cas contraire. Changez l'instruction d'affichage dans le bloc `BEGIN` de *test.pl* en `print "1..4\n";` et ajoutez le code suivant à la fin de ce fichier :

```
print Montest::est_pair(0) == 1 ? "ok 2" : "not ok 2", "\n";
print Montest::est_pair(1) == 0 ? "ok 3" : "not ok 3", "\n";
print Montest::est_pair(2) == 1 ? "ok 4" : "not ok 4", "\n";
```

Le script de test s'exécutera lorsque vous taperez `make test`.

Utilisation des fonctions venant d'une bibliothèque C externe

Jusqu'ici, nos deux exemples ne reposaient sur aucun code C se trouvant à l'extérieur du fichier XS. Nous utiliserons maintenant quelques fonctions de la bibliothèque mathématique C :

```
void
arrondi(arg)
        double arg
    CODE:
        if (arg > 0.0) {
            arg = floor(arg + 0.5);
        } else if (arg < 0.0) {
            arg = ceil(arg - 0.5);
        } else {
            arg = 0.0;
        }
```

```
OUTPUT:
    arg
```

Remarquez que l'`arrondi` que nous définissons ci-dessus ne renvoie pas de valeur mais change plutôt la valeur de son argument sur place.

Les fonctions *floor*(3) et *ceil*(3) font partie de la bibliothèque mathématique C. Si vous compiliez un programme C et que vous aviez besoin de faire une édition de liens avec la bibliothèque mathématique, vous ajouteriez **`-lm`** à la ligne de commande, c'est donc ce que vous mettez dans la ligne `LIBS` du fichier *Makefile.PL* :

```
'LIBS' => ['-lm'],      # Se lie avec la biblio       mathématique 'm'
```

Générez le *Makefile* et lancez *make*. Changez le bloc `BEGIN` pour lancer neuf tests et ajouter ce qui suit dans *test.pl* :

```
$i = -1.5; Montest::arrondi($i); print $i == -2.0 ? "ok 5" : "not ok 5", "\n";
$i = -1.1; Montest::arrondi($i); print $i == -1.0 ? "ok 6" : "not ok 6", "\n";
$i = 0.0; Montest::arrondi($i); print $i == 0.0 ? "ok 7" : "not ok 7", "\n";
$i = 0.5; Montest::arrondi($i); print $i == 1.0 ? "ok 8" : "not ok 8", "\n";
$i = 1.2; Montest::arrondi($i); print $i == 1.0 ? "ok 9" : "not ok 9", "\n";
```

Le lancement de `make test` devrait maintenant afficher que ces neuf tests sont ok.

La documentation, *perlxstut*, livrée avec Perl, présente plusieurs autres exemples d'extensions Perl, y compris un exemple utilisant *h2xs* pour rendre automatiquement disponible une bibliothèque C entière à Perl.

Insérer Perl (utiliser Perl depuis C)

Vous pouvez accéder à un interpréteur Perl depuis du C en insérant Perl à l'intérieur de votre programme C. Puisque Perl est lui-même un programme, l'insertion consiste à prendre les morceaux importants de Perl est à les intégrer dans les vôtres.

Remarquez que l'insertion n'est pas nécessaire si votre seul but est d'utiliser un programme Perl pouvant fonctionner de manière indépendante et que cela ne vous dérange pas de lancer un processus séparé. Vous pouvez utiliser une fonction comme *popen*(3) de C pour échanger vos données entre votre programme C et tout programme Perl externe, exactement comme vous pouvez utiliser `open(PIPE, "| programme")` ou `IPC::Open2` et `IPC::Open3` de Perl pour échanger des données entre votre programme Perl et tout autre programme. Mais si vous voulez éviter le surcoût du lancement d'un processus séparé, vous pouvez insérer un interpréteur dans votre programme C.

Lors du développement d'applications devant tourner longtemps (par exemple pour l'insertion dans un serveur web), c'est une bonne idée de maintenir un seul interpréteur persistant plutôt que de créer et de détruire des interpréteurs encore et encore. La raison majeure est la vitesse, puisque Perl ne sera chargé qu'une seule fois en mémoire. En utilisant un interpréteur Perl persistant, le module `mod_perl` d'Apache évite de charger Perl en mémoire à nouveau à chaque fois que quelqu'un charge une page web d'Apache. La page de manuel *perlembed* donne un exemple d'interpréteur persistant, ainsi qu'un exemple de la manière dont un programme Perl peut gérer de multiples interpréteurs simultanés (un autre atout majeur pour les serveurs web).

Compilation des programmes insérés

Lorsque vous insérez Perl dans C, généralement votre programme C allouera, « exécutera » puis désallouera un objet `PerlInterpreter`, qui est une `struct` C définie dans la bibliothèque *libperl* qui a été construite dans le processus de configuration de Perl pour votre système. La bibliothèque *libperl* (ainsi que *EXTERN.h* et *perl.h*, dont vous aurez également besoin) se trouve dans un répertoire qui variera d'un système à l'autre. Vous devriez être capable de trouver le nom de ce répertoire avec :

```
% perl -MConfig -e "print $Config{archlib}"
```

Vous devriez compiler votre programme exactement de la même manière que votre exécutable *perl* a été compilé. Tout d'abord, vous aurez besoin de savoir quel compilateur C a été utilisé pour construire Perl sur votre machine. Vous pouvez l'apprendre avec :

```
% perl -MConfig -e "print $Config{cc}"
```

Vous pouvez deviner quoi mettre sur la ligne de commande avec le module standard `ExtUtils::Embed`. Si vous aviez un programme C appelé *interp.c* et que votre compilateur C était *cc*, vous pourriez le compiler pour l'insérer comme ceci :

```
% cc -o interp interp.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
```

Ajout d'un interpréteur Perl à votre programme C

Comme *perl* (le programme C) s'avère être un bon exemple d'insertion de Perl (le langage), une démonstration simple de l'insertion peut être trouvée dans le fichier *miniperlmain.c*, dans le code source de Perl. Voici une version non portable de *miniperlmain.c*, contenant les principes essentiels de l'insertion :

```
#include <EXTERN.h>               /* depuis la distribution de Perl */
#include <perl.h>                 /* depuis la distribution de Perl */

static PerlInterpreter *mon_perl;  /*** L'interpréteur Perl ***/

int main(int argc, char **argv, char **env)
{
    mon_perl = perl_alloc();
    perl_construct(mon_perl);
    perl_parse(mon_perl, NULL, argc, argv, (char **)NULL);
    perl_run(mon_perl);
    perl_destruct(mon_perl);
    perl_free(mon_perl);
}
```

Lorsque ceci sera compilé avec la ligne de commande donnée plus haut, vous serez capable d'utiliser *interp* exactement comme un interpréteur perl ordinaire :

```
% interp -e "printf('%x', 3405707998)"
cafefade
```

Vous pouvez également utiliser des instructions Perl stockées dans un fichier en plaçant le nom du fichier dans `argv[1]` avant d'appeler `perl_run`.

Appel d'un sous-programme Perl depuis C

Si un programme Perl contient un sous-programme que vous voulez appeler depuis un programme C, vous pouvez créer un interpréteur Perl, puis utiliser l'une des fonctions commençant par `call_`, documentées dans la page de manuel *perlcall*, pour invoquer ce sous-programme. Supposons que ceci soit votre programme Perl, appelé *affiche_heure.plx* :

```
print "Je peux pas être affiché.";

sub affiche_heure {
    print time;
}
```

Dans cet exemple, nous utiliserons `call_argv` pour invoquer le sous-programme `affiche_heure` depuis ce programme C appelé *affiche_heure.c* :

```
#include <EXTERN.h>;
#include <perl.h>

static PerlInterpreter *mon_perl;

int main(int argc, char **argv, char **env)
{
    char *args[] = { NULL };
    mon_perl = perl_alloc();
    perl_construct(mon_perl);
    perl_parse(mon_perl, NULL, argc, argv, NULL);

    /*** court-circuite perl_run() ***/

    call_argv("affiche_heure", G_DISCARD | G_NOARGS, args);

    perl_destruct(mon_perl);
    perl_free(mon_perl);
}
```

Ici, nous supposons que `affiche_heure` est un sous-programme Perl qui ne prend pas d'argument (c'est le `G_NOARGS`) et pour lequel nous pouvons ignorer la valeur de retour (c'est le `G_DISCARD`). Ces drapeaux, et d'autres, sont présentés dans *perlcall*. Nous compilons et lançons *affiche_heure* comme ceci :

```
% cc -o affiche_heure affiche_heure.c `perl -MExtUtils::Embed -e ccopts -e
ldopts`
% affiche_heure affiche_heure.pl
996412873
```

Dans ce cas particulier, nous n'appelons pas `perl_run` mais, en général, on considère que c'est une bonne formule pour que les méthodes `DESTROY` et les blocs `END` soient exécutés au bon moment.

Si vous voulez passer des arguments au sous-programme Perl, vous pouvez ajouter des chaînes à la liste `args`, terminée par un `NULL`, passée à `call_argv`. Pour d'autres types de données, ou pour examiner les valeurs de retour, vous aurez besoin de manipuler la pile

de Perl. Nous aborderons ceci un peu plus tard ; pour les rouages et le cambouis, lisez la page de manuel *perlcall*, livrée avec Perl.

Évaluation d'une instruction Perl depuis C

Perl fournit deux fonctions pour évaluer des extraits de code Perl : `eval_sv` et `eval_pv`, décrites dans la page de manuel *perlapi*. De façon discutable, celles-ci sont les seules routines dont vous aurez jamais besoin pour exécuter du code Perl depuis votre programme C. Le code exécuté peut être aussi long que vous le désirez, contenir plusieurs instructions et employer `use`, `require` ou `do` pour inclure d'autres fichiers Perl.

`eval_pv` vous permet d'évaluer des chaînes Perl individuelles, puis d'extraire des variables pour la coercition en type C. Le programme suivant, *chaine.c*, exécute trois chaînes Perl, en extrayant un `int` depuis la première, un `float` depuis la deuxième et un `char *` depuis la troisième :

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *mon_perl;

main (int argc, char **argv, char **env)
{
    STRLEN n_a;
    char *insertion[] = { "", "-e", "0" };

    mon_perl = perl_alloc();
    perl_construct( mon_perl );

    perl_parse(mon_perl, NULL, 3, insertion, NULL);
    perl_run(mon_perl);

    /** Traite $a en tant qu'entier **/
    eval_pv("$a = 3; $a **= 2", TRUE);
    printf("a = %d\n", SvIV(get_sv("a", FALSE)));

    /** Traite $a en tant que flottant **/
    eval_pv("$a = 3.14; $a **= 2", TRUE);
    printf("a = %f\n", SvNV(get_sv("a", FALSE)));

    /** Traite $a en tant que chaîne **/
    eval_pv("$a = 'lreP rueugnoM ertua nu erocnE'; $a = reverse($a);", TRUE);
    printf("a = %s\n", SvPV(get_sv("a", FALSE), n_a));

    perl_destruct(mon_perl);
    perl_free(mon_perl);
}
```

Toutes les fonctions avec `Sv` dans le nom, convertissent des scalaires Perl en type de C. Elles sont décrites dans les pages de manuel *perlguts* et *perlapi*. Si vous compilez et exécutez ce programme, vous verrez ce qu'engendre, comme résultats, l'utilisation de `SvIV`

pour créer un `int`, `SvNV` pour créer un `float` et `SvPV` pour créer une chaîne C :

```
a = 9
a = 9.9596000
a = Encore un autre Mongueur Perl
```

Dans l'exemple précédent, nous avons créé une variable globale pour stocker temporairement la valeur calculée de notre expression évaluée. Il est également possible (et dans la plupart des cas, c'est une meilleure solution) d'utiliser la valeur de retour d'`eval_pv` plutôt que de s'en débarrasser :

```
SV *val = eval_pv("reverse 'lreP rueugnoM ertua nu erocnE'", TRUE);
printf("%s\n", SvPV(val,n_a));
```

La page de manuel *perlembed*, livrée avec Perl, comprend une démonstration de `eval_sv` vous permettant d'utiliser les fonctionnalités d'expressions régulières de Perl depuis votre programme C.

Bricolage de la pile de Perl depuis C

En essayant d'expliquer les piles, la plupart des manuels informatiques[3] marmonnent quelque chose à propos d'une pile d'assiettes de cafétéria : la dernière chose que vous avez posée sur la pile est la première que vous retirez. Ça ira pour nos objectifs : votre programme C poussera des arguments sur « la pile de Perl », fermera les yeux pendant que la magie opère, puis retirera les résultats — les valeurs renvoyées par votre sous-programme Perl – de la pile.

Nous allons présenter ici un exemple sans trop d'explications. Pour vraiment comprendre ce qui se passe, vous aurez besoin de savoir comment faire des conversions entre les types C et les types Perl, avec `newSVic`, `sv_setnv`, `newAV` et tous leurs amis décrits dans les pages de manuels *perlguts* et *perlapi*. Vous aurez ensuite besoin de lire *perlcall* pour apprendre comment manipuler la pile de Perl.

Comme C ne possède pas de fonction interne pour l'exponentiation d'entiers, rendons l'opérateur ** de Perl accessible en C. (Ceci est moins utile qu'il n'y paraît, puisque Perl implémente ** avec la fonction *pow*(3) de C.) Tout d'abord, nous allons créer une fonction d'exponentiation dans un fichier de bibliothèque appelé *puissance.pl* :

```
sub expo {
    my ($a, $b) = @_;
    return $a ** $b;
}
```

Maintenant, nous allons créer un programme C, *puissance.c*, avec une fonction appelé `PuissancePerl`, poussant les deux arguments dans la pile, invoquant expo et retirant la valeur de retour :

```
#include <EXTERN.h>
#include <perl.h>

static PerlInterpreter *mon_perl;
```

3. Plus le livre Perl de circonstance.

```
/* "Les vrais programmeurs savent écrire du code assembleur dans
   n'importe quel langage." */

static void
PuissancePerl(int a, int b)
{
    dSP;                     /* initialise un pointeur sur la pile */
    ENTER;                          /* tout est créé après ceci */
    SAVETMPS;                       /* ...est une variable temporaire. */
    PUSHMARK(SP);                   /* se souvient du pointeur de pile */
    XPUSHs(sv_2mortal(newSViv(a))); /* pousse la base sur la pile */
    XPUSHs(sv_2mortal(newSViv(b))); /* pousse l'exposant sur la pile */
    PUTBACK;                 /* rend local le pointeur de pile global */
    call_pv("expo", G_SCALAR);      /* appelle la function */
    SPAGAIN;                        /* rafraîchit le pointeur de pile */
                             /* retire la valeur de retour de la pile */
    printf ("%d à la puissance %d vaut %d.\n", a, b, POPi);
    PUTBACK;
    FREETMPS;                       /* désalloue la valeur de retour */
    LEAVE;                   /* ...et les arguments « mortels » de XPUSH */
}

int main (int argc, char **argv, char **env)
{
    char *my_argv[] = { "", "puissance.pl" };

    mon_perl = perl_alloc();
    perl_construct( mon_perl );

    perl_parse(mon_perl, NULL, 2, my_argv, (char **)NULL);
    perl_run(mon_perl);

    PuissancePerl(3, 4);            /*** Calcule 3 ** 4 ***/

    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

Vous pouvez compiler *puissance.c* pour donner l'exécutable *puissance* ainsi :

```
% cc -o puissance puissance.c `perl -MExtUtils::Embed -e ccopts -e ldopts`
% puissance
3 à la puissance 4 vaut 81.
```

Maintenant votre programme *puissance* peut prendre place en étant également différent.

Morale de l'histoire

Vous pouvez parfois écrire du code plus rapide en C mais vous pouvez toujours écrire plus rapidement du code en Perl. Puisque vous pouvez utiliser l'un depuis l'autre (et inversement), vous n'avez qu'à combiner leurs forces comme il convient. (Et passez le bonjour aux monstres marins des *terra incognita*.)

IV

Culture Perl

22

CPAN

CPAN (*Comprehensive Perl Archive Network*, le réseau d'archives détaillées de Perl) est le dépôt central pour tout ce qui concerne Perl. Il contient la sagesse engrangée par la communauté Perl tout entière : des centaines de modules et de scripts Perl, une documentation représentant la valeur de plusieurs livres et la distribution complète de Perl. Si quelque chose est écrit en Perl, utile et libre, il est certainement sur CPAN. Il existe des miroirs de CPAN dans le monde entier et vous pouvez en trouver près de chez vous grâce au multiplexeur de CPAN, situé à l'adresse *http://www.perl.com/CPAN*. Le multiplexeur se souviendra du miroir que vous avez choisi pour que vous y soyez automatiquement redirigé lorsque vous reviendrez sur *http://www.perl.com/CPAN/* (faites attention au slash à la fin). Vous pouvez également commencer à *www.cpan.org*. L'interface est différente mais les données sont les mêmes.

Une fois dans le répertoire principal de CPAN, vous verrez quelques sous-répertoires :

authors

Ce répertoire contient de nombreux sous-répertoires à raison de un par contributeur de logiciel. Par exemple, si vous désirez trouver le superbe module CGI de Lincoln Stein[1] et que vous êtes parvenus à savoir de source sûre qu'il en est l'auteur, vous pouvez regarder dans *authors/Lincoln_Stein*. Si vous ne savez pas qui l'a écrit, vous pouvez chercher dans le répertoire *modules*, décrit ci-dessous.

doc

Ce répertoire contient toutes les formes de documentation de Perl, y compris toutes les pages de manuel officielles de Perl dans plusieurs formats et arrangement différents, comme du texte, du HTML, du PostScript et du pod, le format natif de Perl, documenté au chapitre 26.

modules

Ce répertoire contient des modules écrits soit en Perl, soit en combinant Perl et C. Voir la discussion à propos du répertoire *modules* ci-dessous.

1. Qui fait maintenant partie de la distribution standard de Perl.

ports

Ce répertoire contient le code source et également quelques fois des images exécutables précompilées des portages de Perl vers des systèmes d'exploitations qui ne sont pas directement supportés dans la distribution standard ou pour lesquels il est de notoriété publique que les compilateurs sont difficiles à se procurer. Ces portages résultent des efforts individuels de leurs auteurs respectifs et peuvent ne pas fonctionner précisément comme ce livre l'indique. De nos jours, peu de systèmes nécessitent un portage spécifique. Il est toutefois intéressant de regarder le document d'index de ce répertoire car il comprend également des informations détaillant le moment où chaque vendeur de systèmes a commencé à livrer Perl.

scripts

Ce répertoire contient une petite collection de divers programmes Perl provenant du monde entier. Ils sont très utiles en tant que programmes seuls et comme exemples (bien que le code ne soit pas assujetti aux vérifications de contrôle de qualité). En ce moment, il n'y a pas beaucoup de programmes répertoriés mais nous espérons que cette zone s'enrichira avec le temps. Le projet *Perl Power Tool* (PPT, Puissants outils en Perl) se trouve également ici. Le projet PPT a pour but de recréer en Perl tous les utilitaires standards d'Unix. La plupart des outils standards sont déjà intégrés, plus quelques-uns qui ne sont pas standards.

src

Dans ce répertoire, vous trouverez le code source de la distribution standard de Perl. En fait, de deux distributions standards de Perl. L'une est indiquée comme *stable* et l'autre comme *devel* (version de développement). (La page d'index de ce répertoire explique tous les détails.) Il s'agit seulement de liens vers les versions appropriées. Au moment où nous écrivons, *stable.tar.gz* est un lien symbolique vers *perl-5.6.1.tar.gz*[2], mais il est susceptible de pointer vers une version plus élevée au moment où vous lirez ceci. Cet énorme fichier contient le code source complet et la documentation de Perl. La configuration et l'installation devraient être relativement simples sur la plupart des plates-formes. Si ce n'est pas le cas, voyez le répertoire *ports* décrit précédemment.

Le répertoire modules de CPAN

Bien que CPAN contiennent le code source complet de Perl, plus quelques distributions binaires pour les systèmes privés de compilateurs C, ainsi que quelques programmes, CPAN est surtout connu pour sa collection de modules.

Lorsque nous parlons de « modules », nous voulons dire trois choses : de purs modules Perl à 100 % (décrits aux chapitres 11 et 12), des extensions (des modules dépendant de code C, décrits au chapitre 21) et des pragmas (des modules contenant des instructions spéciales pour le compilateur de Perl, décrits au chapitre 31). Il existe également sur

2. Le schéma général est que si le deuxième nombre de la version est pair, il s'agit d'une version maintenue ; s'il est impair, il s'agit d'une version de développement. L'extension *.tar.gz*, qui est parfois écrite *.tgz*, indique qu'il s'agit du format standard de l'Internet pour une archive *tar* GNU-zippée, couramment appelée « tarball ».

CPAN des *bundles* (groupements) de modules. Les bundles sont des collections de modules qui interagissent d'une manière ou d'une autre et qui sont généralement la production d'un développeur de modules désirant fournir une solution autonome à un ensemble de problèmes. Si un module dépend d'un autre (et éventuellement d'une version spécifique), les développeurs grouperont souvent les modules dans un bundle. Voir par exemple `Bundle-XML`.

Une façon de parcourir les modules de CPAN est de vous rendre à l'adresse *http://search.cpan.org*, qui fournit un frontal de moteur de recherche pour CPAN. Une autre manière est de vous rendre sur votre miroir local de CPAN et d'entrer dans le répertoire *modules*, où vous verrez trois sous-répertoires : *by-authors*, *by-category* et *by-module* (par auteur, par catégorie et par module). Le répertoire *by-module* peut être le plus utile si votre navigateur possède une fonctionnalité de recherche — bien que (lamentablement) certains modules ne sont disponibles que dans les répertoires sous *author*. Si vous recherchez par catégorie, vous obtiendrez les choix suivants :

Perl core modules, langage extensions, and documentation tools

(Modules centraux de Perl, extensions de langage et outils pour la documentation)

Cette catégorie comprend des pragmas et d'autres modules standards, des modules qui vous aident à écrire différemment en Perl, des modules en rapport avec le compilateur de Perl, des filtres pour le code source et des modules en rapport avec pod, le format de documentation de Perl. Cette catégorie comprend également des modules pour générer du bytecode Java.

Development support

(Support du développement)

Cette catégorie comprend des modules pour en créer d'autres et examiner comment Perl lance des programmes.

Operating system interfaces and hardware drivers

(Interfaces de systèmes d'exploitation et pilotes matériels)

Vous trouverez ici des modules pour interagir avec des entités étranges comme des systèmes d'exploitations, des PalmPilots ou des ports série.

Networking, device control and interprocess communication

(Communication réseau, contrôle de périphériques et communication interprocessus)

Ceci comprend des modules implémentant des protocoles réseau, manipulant des données réseau, maniant des modems et contrôlant les appareils électro-ménagers de votre maison.

Data types and data type utilities

(Types de données et utilitaires pour les types de données)

Cette catégorie contient des modules pour les maths, les statistiques, les algorithmes, les structures de données (et leurs sauvegardes persistante), les dates et les heures, la programmation orientée objet, le PDL (le langage de données de Perl (*Perl Data Language*)) et le POE (l'environnement objet de Perl (*Perl Object Environment*), un ordonnanceur événementiel orienté objet).

Database interfaces

(Interfaces de bases de données)

Vous trouverez ici des modules vous permettant d'exploitant plusieurs douzaines de systèmes de bases de données en Perl, la plupart d'entre elles avec le système DBI de Perl. Cette catégorie inclut les modules DBD spécifiques à chaque base de données.

User interfaces (character and graphical)

(Interfaces utilisateur (en mode caractères et graphique))

Cette catégorie comprend des modules pour manipuler les terminaux utilisateur (édition de la ligne de commande et graphiques en mode caractères dans le style de *curses*(3)), ainsi que Perl/Tk et les liens avec Gtk, Gnome, Sx et Qt pour écrire vos propres interfaces graphiques en Perl.

Interfaces to or emulations of other programming languages

(Interfaces vers ou émulations d'autres langages de programmation)

Cette catégorie contient des modules pour utiliser d'autres langages de programmation depuis Perl ou vous permettre de prétendre que Perl n'est pas ce qu'il prétend être. Si vous êtes intéressés par utiliser Perl en C ou C en Perl, voir le chapitre 21.

Filenames, filesystems, and file locking

(Noms de fichier, systèmes de fichiers et verrous de fichier)

Cette catégorie comprend des modules pour inspecter, créer, verrouiller et tout ce qui concerne la manipulation de fichiers et de répertoires.

String processing, language text processing, parsing, and searching

(Traitement de chaînes et traitement, analyse et recherche de langages textuels)

Cette catégorie contient des modules pour manipuler du texte : placer des césures, enrober, analyser, rechercher des variations grammaticales et faire des recherches. Elle comprend des modules pour manipuler du PostScript, des fontes, du XML et du RTF.

Option, argument, parameter, and configuration file processing

(Traitement d'options, d'arguments, de paramètres et de fichiers de configuration)

Cette catégorie contient des modules pour traiter les arguments de la ligne de commande (le **-x** dans *mon_prog_perl -x*) et pour gérer des fichiers de configuration (comme les fichiers commençant par un point).

Internationalization and locale

(Internationalisation et locales)

Cette catégorie comprend des modules pour tailler sur mesure vos programmes pour un pays et une langue particuliers.

Authentication, security, and encryption

(Authentification, sécurité et cryptage)

Vous trouverez ici des modules pour gérer des mots de passe d'utilisateurs, calculer des signatures de messages, crypter des données et authentifier des utilisateurs.

World Wide Web, HTML, HTTP, CGI, MIME

(*N.d.T.* : vous avez compris sans traduction, non ?)

Cette catégorie contient des modules vous permettant de créer des pages web à base de CGI, des arpenteurs web et des systèmes de gestion de contenus basés sur le web. D'autres modules vous permettent de gérer les cookies, d'analyser du HTML et des messages MIME et de manipuler des caches web. Il existe également une section spéciale uniquement consacrée aux modules Perl que vous pouvez intégrer dans le serveur web Apache.

Server and daemon utilities

(Utilitaires pour les serveurs et les démons)

Cette catégorie comprend des modules pour créer des serveurs de réseau et d'événements.

Archiving, compression, and conversion

(Archivage, compression et conversion)

Vous trouverez ici des modules pour manipuler des fichiers *zip* et *tar* et pour faire des conversions entre des formats de fichiers (même le format de fichier d'Apple II).

Images, pixmap, and bitmap manipulation, drawing, and graphing

(Manipulations d'images, de cartes de pixels et de bits, dessins et graphiques)

Cette catégorie contient des modules pour créer des graphiques, des GIF, du VRML et travailler avec Gimp.

Mail and Usenet news

(*N.d.T.* : vous avez compris sans traduction, non ?)

Dans cette catégorie, vous trouverez des modules pour envoyer, recevoir et filtrer les mails et les news.

Control flow utilities

(Utilitaires de contrôle de flux)

Cette catégorie contient des modules pour exécuter du code Perl de temps à autre.

Filehandle, directory handle, and input/output stream utilities

(Utilitaires pour les handles de fichiers, les handles de répertoires et les flux d'entrées/sorties)

Ici se trouvent des modules pour les entrées depuis des fichiers et les sorties dans des fichiers, y compris des fichiers journaux (*log*). Ceci comprend tous les modules `IO::` et un module `Expect` pour automatiser les dialogues avec des services réseaux ou d'autres programmes interactifs.

Microsoft Windows modules

(*N.d.T.* : vous avez compris sans traduction, non ?)

Cette catégorie comprend des modules pour manipuler la base de registre de Windows, ASP, ODBC, OLE et d'autres technologies spécifiques à Windows.

Miscellaneous modules

(Modules divers)

Vous trouverez ici des modules pour l'astronomie, la biologie, la chimie, la validation (ou l'invalidation) de cartes de crédit, les amortissements d'hypothèques, l'audio, la vidéo, la norme MIDI, la météo et les jeux.

Utilisation des modules de CPAN

La plupart des modules que vous trouverez sur CPAN sont dans un format « tarball ». C'est-à-dire qu'ils possèdent une extension de fichier en *tar.gz* et qu'ils sont décompressés dans un répertoire avec le code du module et tous les fichiers annexes, comprenant généralement un fichier *README* et un fichier *Makefile.PL*.

Quatre étapes rendent accessibles les modules de CPAN depuis vos programmes : la décompression, le dépaquetage, la construction et l'installation. La manière dont fonctionne chacune de ces étapes dépend de votre système d'exploitation et du module que vous installez, nous ne pouvons donc pas vous donner de recette à toute épreuve qui fonctionnerait à chaque fois. Lorsque vous doutez, lisez les fichiers *README* et *INSTALL* qui sont, espérons-le, distribués avec le module. Lisez également la page de manuel de *perlmodinstall*.

Mais il se peut que vous n'ayez jamais à réfléchir à la procédure d'installation si vous utilisez le module `CPAN` (livré avec la distribution de Perl) ou PPM (*Perl Package Manager*, le gestionnaire de paquetages Perl, livré avec la distribution ActiveState de Perl). Pour utiliser le module `CPAN` (à ne pas confondre avec CPAN, lui-même), tapez :

```
% perl -MCPAN -e "shell"
```

sur votre ligne de commande pour commencer le processus de configuration. Après avoir répondu à une variété de question sur la manière dont vous préférez récupérer des fichiers, vous pouvez installer un module en tapant :

```
install Un::Module
```

dans le shell du module `CPAN`, ou en tapant :

```
% perl -MCPAN -e "install 'Un::Module'"
```

sur votre ligne de commande ordinaire.

Si vous ne disposez pas des avantages pratiques du module `CPAN` ou de PPM, vous devrez passer à la main les étapes décrites dans les paragraphes suivants. Les instructions sont données pour Unix, Windows et Macintosh ; pour d'autres systèmes d'exploitation, consultez la page de manuel de *perlmodinstall*.

Décompression et dépaquetage des modules de CPAN

La plupart des milliers d'utilitaires sur CPAN sont compressés pour occuper moins de place. Une fois que vous avez récupéré le tarball d'un module, vous devez tout d'abord le transformer en une arborescence de répertoires sur votre système en décompressant (« dézippant ») et en dépaquetant le tarball. Sur Unix, vous pouvez utiliser *gzip* et *tar* pour faire cela. (Sur la plupart des systèmes, *tar* fera les deux.) Sur Windows, *WinZip* décompressera et dépaquetera à la fois les tarballs. Sur un Macintosh, vous pouvez utiliser *StuffIt*, *DropStuff*, *MacGzip* ou *suntar*.

Construction des modules de CPAN

Une minorité de modules de CPAN viennent avec du code C que vous devrez compiler pour votre système, ce qui est naturellement un problème pour les systèmes qui ne disposent pas d'un compilateur C. La procédure standard pour construire un module de CPAN (avec ou sans code C) réside dans les trois commandes suivantes, exécutées à partir de la ligne de commande. (Si vous n'avez pas de ligne de commande ou d'équivalent de *make*, vous devrez avoir recours à des mesures plus drastiques et plus dépendantes du système. Les utilisateurs de Windows possèdent une ligne de commande mais peuvent avoir besoin d'utiliser l'utilitaire *nmake* à la place de *make*.)

```
% perl Makefile.PL

% make

% make test
```

La commande *perl Makefile.PL* essayera de créer un *Makefile*, qu'utiliseront les commandes *make* qui suivent pour déterminer quels utilitaires doivent être construits et dans quel ordre. La dernière commande, *make test*, lance la série de tests que par bonheur l'auteur du module a incluse.

Installation des modules de CPAN dans la bibliothèque Perl

Si vous avez suivi les étapes précédentes, vous avez maintenant un module qui a été construit et testé mais pas encore installé dans la bibliothèque de Perl. Lorsque Perl a été installé sur votre système, un répertoire *lib* a été créé pour contenir des modules, des pragmas et les fichiers qui s'y rapportent. Il s'agit de la bibliothèque Perl, qui se situe généralement à un endroit comme */usr/local/lib/perl5* sur les système Unix et *C:\PERL\LIB* par défaut sur les systèmes Windows. Les modules installés après que Perl a été construit sont stockés dans le sous-répertoire *site_perl* de la librairie Perl. Vous pouvez voir les noms de tous les répertoires de votre bibliothèque (et un tas d'autres choses) en écrivant :

```
% perl -V
```

Pour installer le module, tapez :

```
% make install
```

Cela nécessite normalement d'être super-utilisateur, même pour installer des modules dans les répertoires de la bibliothèque Perl spécifiques à votre site.

Avec un peu de travail, vous pouvez installer le module dans un répertoire hors de votre bibliothèque Perl (comme votre répertoire maison). Au lieu de taper normalement *perl Makefile.PL* pour créer un *Makefile*, vous pouvez plutôt utiliser cette incantation :

```
% perl Makefile.PL LIB=/mon/rep/perllib \
        INSTALLMAN1DIR=/mon/rep/man/man1 \
        INSTALLMAN3DIR=/mon/rep/man/man3 \
            INSTALLBIN=/mon/rep/bin \
         INSTALLSCRIPT=/mon/rep/scripts
```

Ceci installera les modules quelque part dans le répertoire */mon/rep/perllib* et tous les fichiers restants là où ils doivent aller. (Si vous vous trouvez en train de taper cela souvent, vous pouvez même écrire un petit programme Perl pour le faire à votre place. Perl est très bon pour accomplir des choses comme celles-là.)

Vous pouvez ensuite indiquer à Perl de rechercher les modules dans votre répertoire spécial en ajoutant :

```
use lib "/mon/rep/perllib";
```

avant que votre programme ne tente de charger le module. Vous pouvez également positionner la variable d'environnement `PERL5LIB` à ce répertoire ou utiliser l'option de Perl **`-I`**. Voir le pragma *use lib* au chapitre 31, qui donne des exemples pour faire cela.

Création de modules pour CPAN

Si vous avez un module dont vous pensez qu'il puisse être utile aux autres, envisagez de rendre le monde meilleur en le téléchargeant sur CPAN. Le serveur qui gère les soumissions de nouveaux modules est appelé PAUSE (*Perl Authors Upload Server*, le serveur de téléchargement des auteurs Perl) et on peut le trouver sur *https://pause.kbx.de/pause/*. Avant de pouvoir télécharger votre module, vous devrez obtenir un compte sur PAUSE. C'est un peu comme devenir un « développeur Perl répertorié ».

Si vous vous considérez vous-même comme un développeur Perl répertorié, vous devriez en savoir assez pour documenter vos modules. Perl possède une convention pour inclure la documentation à l'intérieur de votre code source. (De cette manière, vous ne l'égarez jamais.) Cette documentation incluse se trouve dans un format appelé « pod » (pour *Plain Old Documentation*, bonne vieille documentation) et est décrite au chapitre 26.

Vous devriez envisager de sécuriser votre module du point de vue des threads. Voir le chapitre 17, *Threads*.

Vous devriez également vous inquiéter quelque peu si votre joli petit module fait ou non des choses risquant de perturber la sécurité des gens qui l'utilisent, car les autres peuvent avoir de véritables bonnes raisons d'être plus concernés par la sécurité que vous ne l'êtes (POUR L'INSTANT). Voir le chapitre 23, *Sécurité*, pour tout savoir comment éviter d'être tenu responsable de la 3^{e} Guerre Mondiale et d'autres nuisances du même acabit.

Les modules destinés à être distribués sur CPAN doivent inclure un programme Perl appelé *Makefile.PL* qui, lorsqu'on le lance, génère un *Makefile*, ainsi qu'un fichier *README* décrivant brièvement ce que fait le module et comment l'installer. Le *Makefile* s'attend à comporter également une série de tests. Vous pouvez créer tous ces fichiers d'un coup grâce à l'utilitaire *h2xs* :

```
h2xs -X -n Truc::Machin
```

(Remplacez **`-X`** par **`-A`** si vous construisez un module comprenant un composant XS. XS est décrit au chapitre 21.) Le programme *h2xs* crée un seul répertoire avec des squelettes de fichiers que vous devrez garnir de chair. Lorsque vous avez terminé, vous pouvez télécharger le répertoire dans un tarball sur PAUSE.

Le *Makefile.PL* généré par *h2xs* ressemblera à quelque chose comme ceci :

```
use ExtUtils::MakeMaker;
```

```
# See lib/ExtUtils/MakeMaker.pm for details of how to influence
# the contents of the Makefile that is written.[3]
WriteMakefile(
    NAME         => 'Mon_test',
    VERSION_FROM => 'Mon_test.pm', # finds $VERSION
    LIBS         => [''],    # e.g., '-lm'
    DEFINE       => '',      # e.g., '-DHAVE_SOMETHING'
    INC          => '',      # e.g., '-I/usr/include/other'
);
```

La première chose que fait le *Makefile.PL* est de charger le module `ExtUtils::MakeMaker`. La fonction `WriteMakefile` de `MakeMaker` possède une pléthore d'options (avec une pléthore valant approximativement 88) pour vous aider à personnaliser ce qui se passe après que l'utilisateur a téléchargé votre module depuis CPAN et a tapé *perl Makefile.PL* pour commencer sa construction. La bonne nouvelle à propos de tout ceci est que, puisque l'utilisateur est supposé lancer le *perl* qui sera utilisé plus tard, vous disposez d'une large étendue d'informations de configuration disponibles (*via* le module `Config` ou la variable spéciale `$^O`) pour vous aider à décider comment diriger `MakeMaker`. D'un autre côté, `MakeMaker` est vraiment excellent pour trouver des valeurs par défaut satisfaisantes pour presque tout, ainsi le fichier squelette écrit par *h2xs* peut très bien contenir tout ce dont vous avez besoin avec peut-être un ou deux ajustements. Pour plus d'informations à ce sujet, voir la documentation en ligne très complète de `ExtUtils::MakeMaker`.

Lorsque vous écrivez un module Perl destiné à un usage général, vous devriez permettre que la version de Perl de l'utilisateur diffère de la vôtre. Vous devriez toujours inclure une description en anglais de toutes les dépendances (les versions particulières de Perl, des pré-requis du système ou d'autres modules) dans le fichier *README*. De tout façon, cela ne suffira peut-être même pas, puisque quelqu'un qui utilise le module `CPAN` pour télécharger et installer automatiquement votre module ne verrait jamais cet avertissement. Vous devriez donc vérifier les dépendances dans le *Makefile.PL*. Voilà comment vous pourriez vous assurer que la personne qui télécharge votre module lance Perl 5.6 ou une version supérieure :

```
eval { require 5.6.0 }
    or die << 'FIN_DOC';
############
### This module requires lvaluable subroutines, which are not available
### in versions of Perl earlier than 5.6. Please upgrade![4]
############
FIN_DOC
```

Tests internes

Les instructions standards pour installer un module disent à l'utilisateur de lancer *make test* après avoir construit le module avec *make*. Merci donc d'inclure un script de test dé-

3. *N.d.T.* : Voir */lib/ExtUtils/MakeMaker.pm* pour les détails sur comment influencer le contenu du *Makefile* généré.

4. *N.d.T.* : ce module nécessite des sous-programmes pris comme lvalues, qui ne sont pas disponibles dans les versions de Perl antérieures à 5.6. Merci de mettre à niveau !

cent avec tout module que vous téléchargez vers CPAN. Vous devriez émuler le style `ok`/`not ok` que Perl emploie dans sa propre série de tests, afin que l'utilisateur détermine facilement le résultat de chaque test unitaire. Le fichier *test.pl* généré par *h2xs* vous aidera à démarrer. Le chapitre 21 donne des exemple de tests que vous pouvez ajouter à la fin de *test.pl*.

Si vous avez beaucoup de tests unitaires, vous pouvez avoir envie de singer la série de tests de Perl en créant un sous-répertoire appelé *t/* dans le répertoire du module et d'ajouter l'extension *.t* au nom de vos différents scripts de test. Lorsque vous lancerez *make test*, tous les fichiers de tests seront exécutés automatiquement.

Tests externes

Les modules téléchargés vers CPAN sont testés par une variété de volontaires sur différentes plates-formes. Ces testeurs de CPAN sont notifiés par email de chaque téléchargement nouveau et répondent à la liste de diffusion par `PASS`, `FAIL`, `NA` (non applicable sur cette plate-forme) ou `UNKNOWN` (ne sait pas), avec les remarques appropriées. Vous pouvez trouver la liste de diffusion des testeurs de CPAN sur *cpan-testers@perl.org* ; les résultats des tests sont postés sur *http:// testers.cpan.org/*.

Il ne s'agit que de la phase de test préliminaire, bien entendu. La véritable phase de test commence lorsque quelqu'un branche votre petit module sur un serveur web qui croule sous un million de pages vues par jour. Ou lors de l'utilisation de votre module pour mettre au point l'avion que vous emprunterez un jour prochain.

Alors foncez, sautez l'écriture de ces petits tests minables. Et voyez si l'on s'en fiche...

23

Sécurité

Que vous ayez affaire à un utilisateur assis à son clavier et tapant des commandes ou à quelqu'un envoyant des informations sur le réseau, la prudence s'impose pour ce qui est des données arrivant dans vos programmes car l'autre personne peut, de manière malveillante ou accidentelle, vous envoyer des données qui feront plus de mal que de bien. Perl fournit un mécanisme spécial de vérification de la sécurité, appelé mode de marquage, permettant d'isoler des données marquées (*N.d.T.* : comme étant douteuses) afin de ne pas les utiliser mal à propos. Par exemple, si vous faites confiance par erreur à un nom de fichier marqué, vous pouvez vous retrouver en train d'ajouter une entrée à votre fichier de mots de passe alors que vous pensiez le faire sur un fichier de journal. Le mécanisme de marquage est exposé à la section *Gestion des données non sûres*.

Dans les environnements multitâches, des actions, dans les coulisses, par des acteurs que l'on ne voit pas peuvent affecter la sécurité de votre propre programme. Si vous présupposez que vous êtes propriétaire exclusif d'objets externes (en particulier des fichiers) comme si vos processus étaient les seuls sur le système, vous vous exposez à des erreurs substantiellement plus subtiles que celles provenant directement de la gestion de données ou de code dont la provenance est douteuse. Perl vous aide alors quelque peu en détectant des situations hors de votre contrôle. Mais pour celles dont vous avez le contrôle, il vous faudra comprendre quelles sont les approches à l'épreuve des étrangers invisibles. La section *Gestion des problèmes de synchronisation* discute de ces questions.

Si la donnée que vous avez reçue d'un étranger s'avère être un morceau de code source à exécuter, vous devez être encore plus prudent que vous ne le seriez avec une donnée. Perl fournit des vérifications pour intercepter du code furtif déguisé en donnée pour que vous ne l'exécutiez pas à votre insu. Si vous ne voulez pas exécuter de code étranger, le module `Safe` vous permet pourtant de mettre le code suspect en quarantaine là où il ne peut plus faire aucun mal et où il pourrait même faire du bien. Ce sont là les thèmes de la section *Gestion du code non sûr*.

Gestion des données non sûres

Perl facilite une programmation sécurisée même lorsque le programme est utilisé par quelqu'un de moins fiable que le programme lui-même. C'est-à-dire que certains programmes ont besoin d'accorder des privilèges limités à leurs utilisateurs, sans accorder inutilement d'autres privilèges. Les programmes *setuid* et *setgid* tombent dans cette catégorie sur Unix, comme d'autres programmes tournant avec des modes de privilèges variés sur d'autres systèmes d'exploitation qui implémentent de telles notions. Même sur les systèmes où ce n'est pas le cas, les mêmes principes s'appliquent aux serveurs réseaux et à tout programme lancé par ces serveurs (comme les scripts CGI, les gestionnaires de listes de diffusion et les démons répertoriés dans */etc/inetd.conf*). Tous ces programmes exigent un niveau de vigilance plus élevé que la moyenne.

Même les programmes lancés à partir de la ligne de commande sont parfois de bons candidats pour le mode de marquage, tout particulièrement s'ils sont destinés à être lancés par un utilisateur privilégié. Les programmes agissant sur des données non fiables, comme ceux qui génèrent des statistiques à partir de fichiers journaux ou qui utilisent `LWP::*` ou `Net::*` pour récupérer des données distantes, devraient probablement être lancés avec le marquage explicitement activé ; les programmes imprudents risquent d'être changés en « chevaux de Troie ». Puisque les programmes ne retirent aucune émotion à prendre des risques, il n'y a pas de raison pour qu'ils ne soient pas précautionneux.

Au regard des shells de la ligne de commande sur Unix, qui ne sont véritablement que des infrastructures pour appeler d'autres programmes, Perl est facile à programmer de manière sécurisée car il est direct et autonome. À l'inverse de la plupart des langages de programmation shells, qui sont basés sur de nombreuses et mystérieuses passes de substitution sur chaque ligne du script, Perl emploie un modèle d'évaluation plus conventionnel comportant moins d'inconvénients cachés. De plus, comme le langage intègre plus de fonctionnalités internes, il dépend moins de programmes externes (dont la fiabilité peut être mise en doute) pour accomplir son devoir.

Sous Unix, le berceau de Perl, le meilleur moyen de compromettre la sécurité du système est de couvrir de flatteries un programme privilégié pour qu'il fasse quelque chose pour lequel il n'est pas prévu. Pour parer de telles attaques, Perl a développé une approche unique pour faire face aux environnements hostiles. Perl active automatiquement le mode de marquage chaque fois qu'il détecte que son programme tourne avec des ID d'utilisateur ou de groupe réels et effectifs différents.[1] Même si le fichier contenant votre script Perl ne possède pas de bit setuid ou setgid activé, ce script peut toujours se retrouver lancé en mode de marquage. Cela se produit si votre script a été invoqué par un autre programme, qui *lui-même* était lancé sous des ID différents. Les programmes Perl qui ne sont pas conçus pour fonctionner en mode de marquage tendent à expirer prématurément lorsqu'ils sont surpris à violer la politique de marquage de sécurité. Perl n'est pas si crédule.

1. Le bit setuid des permissions UNIX est le mode 04000, et le bit setgid est 02000 ; l'un comme l'autre peuvent être positionnés pour donner à l'utilisateur du programme certains des privilèges de son (ou de ses) propriétaire(s). D'autres systèmes d'exploitation peuvent conférer des privilèges spéciaux aux programmes par d'autres moyens, mais le principe reste le même.

Vous pouvez également activer explicitement le mode de marquage en utilisant l'option de la ligne de commande **-T**. Vous devriez le faire pour les démons, les serveurs et les programmes qui tournent pour le compte de quelqu'un d'autre, comme les scripts CGI. Les programmes qui se lancent à distance et anonymement par quelqu'un sur le Net sont exécutés dans le plus hostile des environnements. Vous ne devriez pas craindre de dire « Non ! » de temps en temps. Contrairement à la croyance populaire, vous pouvez faire preuve d'une grande prudence sans pour autant vous transformer en prude faussement effarouchée.

Sur les sites les plus consciencieux en matière de sécurité, lancer tous les scripts CGI avec l'option **-T** n'est pas seulement une bonne idée : c'est la règle. Nous ne prétendons pas que lancer les scripts en mode de marquage est suffisant pour les rendre sécurisés. Ce n'est pas le cas et cela prendrait un livre entier pour seulement mentionner tout ce qu'il faudrait faire. Mais si vous n'exécutez pas vos scripts CGI en mode de marquage, vous faites inutilement l'impasse sur la protection la plus forte que Perl puisse vous offrir.

Pendant qu'il se trouve en mode de marquage, Perl prend des précautions particulières appelées vérifications du marquage (*taint checks*) pour se prémunir contre les pièges qu'ils soient triviaux ou subtils. Certaines de ces précautions sont assez simples, comme de vérifier que les variables d'environnement dangereuses ne soient pas positionnées et que les répertoires de votre *PATH* ne soient pas modifiables par quelqu'un d'autre ; les programmeurs prudents ont toujours procédé ainsi. D'autres vérifications sont toutefois mieux assurées par le langage lui-même et ce sont celles-là qui contribuent à rendre un programme Perl privilégié plus sûr que le programme C correspondant ou un script CGI en Perl plus sécurisé que le même script dans un langage sans vérifications de marquage. (C'est-à-dire, pour autant que nous le sachions, tous les langages à part Perl.)

Le principe est simple : vous ne pouvez pas utiliser des données dérivées de l'extérieur d'un programme pour affecter quelque chose d'autre en dehors de ce programme — du moins, pas accidententellement. Tout ce qui provient de l'extérieur du programme est marqué, ce qui comprend tous les arguments de la ligne de commande, les variables d'environnement et les fichiers en entrée. Les données marquées ne peuvent être utilisées directement ou indirectement dans toute opération invoquant un sous-shell ou modifiant des fichiers, des répertoires ou des processus. Toute variable positionnée dans une expression ayant précédemment référencé une valeur marquée devient elle-même marquée, même s'il est logiquement impossible que la valeur marquée ait influencé la variable. Le marquage étant associé à chaque valeur scalaire, certaines valeurs individuelles d'un tableau ou d'un hachage peuvent être marqués et d'autres non. (Seules les valeurs dans un hachage peuvent être marquées et non les clefs.)

Le code suivant illustre comment le marquage fonctionnerait si vous exécutiez toutes ces instructions dans l'ordre. Les instructions indiquées « Non sûres » lèveraient une exception, contrairement à celles qui sont « OK ».

```
$arg = shift(@ARGV);      # $arg est maintenant marqué (à cause de @ARGV).
$cache = "$arg, 'truc'";  # $cache est aussi marqué (à cause de $arg).
$ligne = <>;              # Marqué (lu à partir d'un fichier externe).
$chemin = $ENV{PATH};     # Marqué à cause de %ENV, mais voir plus loin.
$perso = 'abc';           # Non marqué.

system "echo $perso";     # Non sûr avant que le PATH ne soit positionné.
```

```
system "echo $arg";           # Non sûr : utilise sh avec $arg marqué.
system "echo", $arg;          # OK une fois que le PATH est positionné
                              # (n'utilise pas sh).
system "echo $cache";         # Non sûr de deux façons : marqué et PATH.

$vieux_chemin = $ENV{PATH};   # $vieux_chemin est marqué (à cause de %ENV).
$ENV{PATH} = 'bin:/usr/bin';  # (R                on d'autres programmes.)
$nouveau_chemin = $ENV{PATH}; # $nouveau_chemin n'est PAS marqué.

delete @ENV{qw{IFS
               CDPATH
               ENV
               BASH_ENV}};    # Rend %ENV plus sûr.

system "echo $perso";  # OK, est sûr une fois que le PATH est réinitialisé.
system "echo $cache";  # Non sûr via $cache qui est marqué.

open(OUF, "< $arg");   # OK (ouvertures en lecture seule non vérifiées).
open(OUF, "> $arg");   # Non sûr (tentative d'écriture dans arg qui est
                       # marqué).

open(OUF, "echo $arg|")       # Non sûr via $arg qui est marqué, mais...
    or die "impossible de faire un pipe sur echo : $!";

open(OUF,"-|")                # Considéré OK : voir ci-dessous pour
    or exec 'echo', $arg      # l'exemption de marquage sur une liste
    or die "exec echo impossible : $!";  # exec-utée.

open(OUF,"-|", "echo", $arg)  # Comme précédemment, considéré OK.
    or die "impossible de faire un pipe sur un echo : $!";

$cri = `echo $arg`;    # Non sûr via $arg qui est marqué.
$cri = `echo abc`;     # $cri est marqué à cause des apostrophes inverses.
$cri2 = `echo $cri`;   # Non sûr via $cri qui est marqué.

unlink $perso, $arg;          # Non sûr via $arg qui est marqué.
umask $arg;                   # Non sûr via $arg qui est marqué.

exec "echo $arg";      # Non sûr via $arg qui est marqué et passé au shell.
exec "echo", $arg;     # Considéré OK ! (Mais voir ci-dessous.)
exec "sh", '-c', $arg; # Considéré OK, mais à tort !
```

Si vous essayez de faire quelque chose de non sûr, vous obtiendrez une exception (qui devient une erreur fatale si elle n'est pas interceptée) comme « `Insecure dependency` » ou « `Insecure $ENV{PATH}` » (« Dépendance non sûre » ou « `$ENV{PATH}` non sûr ». Voir la section *Nettoyage de votre environnement* plus loin.

Si vous passez une *`LISTE`* à un `system`, un `exec` ou un `open` de pipe, les arguments ne sont pas inspectés pour vérifier s'ils sont marqués car, avec une *`LISTE`* d'arguments, Perl n'a pas besoin d'invoquer le shell, qui est potentiellement dangereux, pour lancer la commande. Vous pouvez toujours écrire facilement un `system`, un `exec` ou un `open` de

pipe non sûr en utilisant la forme avec *LISTE*, comme le démontre le dernier exemple ci-dessus. Ces formes sont exemptes de vérification car vous êtes supposés savoir ce que vous faites lorsque vous les employez.

Parfois, vous ne pouvez toutefois pas dire combien d'arguments vous êtes en train de passer. Si vous fournissez à ces fonctions un tableau[2] qui ne contient qu'un élément, c'est alors exactement comme si vous passiez une chaîne en premier et le shell pourrait alors être utilisé. La solution est de passer un chemin explicite dans l'emplacement pour un objet indirect :

```
system @args;               # N'appellera pas le shell sauf si @args == 1.
system { $args[0] } @args;  # Court-circuite le shell même avec une liste
                            # d'un seul argument.
```

Détection et blanchiment des données marquées

Pour tester si une variable scalaire contient ou non une donnée marquée, vous pouvez utiliser la fonction `est_marquee` suivante. Elle utilise le fait que `eval` *`CHAINE`* lève une exception si vous essayez de compiler des données marquées. Cela ne fait rien que la variable `$nada` employée dans l'expression à compiler soit toujours vide ; elle sera encore marquée si `$arg` l'est. Le `eval` *`BLOC`* extérieur n'effectue aucune compilation. Il n'est ici que pour intercepter l'exception levée si l'on donne une donnée marquée à l'`eval` intérieur. Puisqu'il est garanti que la variable `$@` n'est pas vide après chaque `eval` si une exception a été levée et vide sinon, nous renvoyons le résultat du test vérifiant si sa longueur était nulle :

```
sub est_marquee {
    my $arg = shift;
    my $nada = substr($arg, 0, 0);  # longueur nulle
    local $@;  # préserve la version de l'appelant
    eval { eval "# $nada" };
    return length($@) != 0;
}
```

Mais tester le marquage ne vous en dit pas plus. Généralement, vous savez parfaitement quelles variables contiennent des données marquées — il vous suffit de nettoyer le marquage des données. Le seul moyen officiel de court-circuiter le mécanisme de marquage consiste à référencer des variables de sous-motifs déterminés par une précédente recherche de correspondance d'expression régulière.[3] Lorsque vous écrivez un motif contenant des parenthèses capturantes, vous pouvez accéder aux sous-chaînes capturées par le biais des variables de correspondance comme `$1`, `$2` et `$+` ou en évaluant le motif dans un contexte de liste. Quel que soit le moyen employé, on présume que vous étiez au cou-

2. Ou une fonction produisant une liste.

3. Un moyen officieux consiste à stocker une chaîne marquée comme clef d'un hachage et de récupérer à nouveau cette clef. Car les clefs ne sont pas des SV complets (nom interne des valeurs scalaires), elles ne portent pas sur elle la propriété du marquage. Ce comportement peut changer un jour ou l'autre, ne comptez donc pas dessus. Soyez prudent lorsque vous manipulez des clefs, de peur que vous n'enleviez accidentellement le marquage de vos données et que vous ne fassiez quelque chose de dangereux avec.

rant de ce que vous étiez en train de faire lorsque vous avez écrit le motif et que vous l'avez écrit de manière à éviter tout danger. Vous devez donc réfléchir un peu — ne blanchissez jamais aveuglément ou c'est tout le mécanisme qui sera invalidé.

Il vaut mieux vérifier que la variable ne contient que des caractères valides au lieu de regarder si elle contient des caractères invalides. Ceci car il est beaucoup trop facile d'en rater auxquels vous n'auriez jamais pensé. Par exemple, voici un test destiné à s'assurer que `$chaine` ne contient que des caractères « de mot » (alphabétiques, numériques et soulignés), des tirets, des signes et des points :

```
if ($chaine =~ /^([-\@\w.]+)$/) {
    $chaine = $1;                      # $chaine est maintenant non marquée.
} else {
    die "Mauvaises données dans $chaine"; # Enregistrez ceci quelque part.
}
```

Ceci rend `$chaine` raisonnablement fiable pour l'utiliser ensuite dans une commande externe, puisque `/\w+/` ne correspond généralement pas à des métacaractères du shell, et que les autres caractère ne veulent rien dire de particulier pour le shell.[4] Il aurait été dangereux d'utiliser `/(.+)/` car ce motif laisse tout passer, ce que Perl ne vérifie pas. Au moment du nettoyage, soyez extrêmement prudent avec vos motifs. Le blanchiment des données en utilisant des expressions régulières représente le *seul* mécanisme interne approuvé pour nettoyer de sales données. Et parfois, cette approche est entièrement mauvaise. Si vous êtes en mode de marquage car vous tournez avez set-id et non parce que vous avez intentionnellement activé `-T`, vous pouvez limiter les risques en lançant avec `fork` un fils avec des privilèges moindres ; voir la section *Nettoyage de votre environnement*.

Le pragma `use re 'taint'` désactive le nettoyage implicite de toutes les correspondances de motifs jusqu'à la fin de la portée lexicale courante. Vous pourriez utiliser ce pragma si vous vouliez seulement extraire quelques sous-chaînes d'une donnée potentiellement marquée ; mais puisque ne vous êtes pas inquiété de la sécurité, vous feriez mieux de laisser les sous-chaînes marquées pour vous préserver des accidents malchanceux à venir.

Imaginez que vous trouviez une correspondance avec quelque chose comme ceci, avec `$chemin_complet` marqué :

```
($rep, $fichier) = $chemin_complet =~ m!(.*/)(.*)!s;
```

Par défaut, `$rep` et `$fichier` seraient maintenant marqués. Mais vous ne vouliez probablement pas faire cela de manière si cavalière car vous n'avez jamais vraiment réfléchi aux problèmes de sécurité. Par exemple, vous ne seriez pas follement heureux si `$fichier` contenait la chaîne « `; rm -rf * ;` », pour ne citer qu'un exemple tout à fait remarquable. Le code suivant laisse les deux variables renvoyées marquées si `$chemin_complet` l'était :

4. À moins que vous n'utilisiez intentionnellement des locales corrompues. Perl présuppose que les définitions de locales de votre système sont potentiellement compromises. Ainsi, lorsque le pragma `use locale` est actif, les motifs contenant une classe de caractères symbolique, comme `\\w` ou `[[:alpha:]]`, produisent des résultats marqués.

```
{
    use re 'taint';
    ($rep, $fichier) = $chemin_complet =~ m!(.*/)(.*)!s;
}
```

Une bonne stratégie consiste à laissez les sous-correspondances marquées par défaut dans le fichier source entier et de ne permettre d'enlever le marquage que sélectivement dans des portées imbriquées si le besoin s'en fait sentir :

```
use re 'taint';
# le reste du fichier laisse maintenant $1, etc. marqués
{
    no re 'taint';
    # ce bloc enlève maintenant le marquage dans les
    # correspondances d'expressions régulières
    if ($num =~ /^(\d+)$/) {
        $num = $1;
    }
}
```

Une entrée depuis un handle de fichier ou de répertoire est automatiquement marquée, sauf si elle provient du handle de fichier spécial `DATA`. Si vous le souhaitez, vous pouvez désigner d'autres handles comme des sources dignes de confiance *via* la fonction `untaint` du module `IO::Handle` :

```
use IO::Handle;

IO::Handle::untaint(*UN_HF);     # Soit de manière procédurale
UN_HF->untaint();                # soit en utilisant un style OO.
```

Désactiver le marquage sur un handle de fichier complet est un geste dangereux. Comment savez-vous *vraiment* qu'il est sûr ? Si vous vous apprêtez à faire cela, vous devriez au moins vérifier que personne à part le propriétaire ne puisse écrire dans le fichier.[5] Si vous êtes sur un système de fichiers Unix (et un qui restreint prudemment *chown*(2) au super-utilisateur), le code suivant fonctionne :

```
use File::stat;
use Symbol 'qualify_to_ref';
sub handle_semble_sur(*) {
    my $hf = qualify_to_ref(shift, caller);
    my $info = stat($hf);
    return unless $info;

    # le propriétaire n'est ni le super-utilisateur, ni « moi »,
    # dont l'uid réel est dans la variable $<
    if ($info->uid != 0 && $info->uid != $<) {
        return 0;
    }
```

5. Bien que vous puissiez également enlever le marquage d'un handle de répertoire, cette fonction ne marche que sur un handle de fichier. Car, étant donné un handle de répertoire, il n'existe pas de manière portable d'extraire son descripteur de fichier avec `stat`.

```
    # vérifie si le groupe ou les autres peuvent écrire dans le fichier
    # utilise 066 pour détecter également la possibilité de lire
    if ($info->mode & 022) {
        return 0;
    }
    return 1;
}

use IO::Handle;
UN_HF->untaint() if handle_semble_sur(*UN_HF);
```

Nous avons appelé `stat` sur le handle de fichier et non sur le nom de fichier pour éviter une situation de concurrence (*race condition*) dangereuse. Voir la section *Gestion des situations de concurrence* plus loin dans ce chapitre.

Remarquez que cette routine n'est qu'un bon point de départ. Un version légèrement plus paranoïaque aurait également vérifier tous les répertoires parents, même si vous ne pouvez pas faire un `stat` sur un handle de répertoire de manière fiable. Mais si l'un des répertoires parents est en écriture pour tout le monde, vous savez que vous êtes en danger qu'il y ait ou non des situations de concurrence.

Perl a ses propres notions en ce qui concerne les opérations qui sont dangereuses mais il est toujours possible de se plonger dans les ennuis avec d'autres opérations qui se moquent d'utiliser ou non des valeurs marquées. Il ne suffit pas toujours d'être prudent avec les entrées. Les fonctions de Perl pour les sorties ne testent pas si leurs arguments sont marqués ou non, mais, dans certains environnements, cela compte. Si vous ne faites pas attention à ce que vous écrivez en sortie, vous pourriez tout bonnement finir par cracher des chaînes ayant une signification inattendue pour quiconque traitant la sortie. Si vous tournez sur un terminal, les séquences d'échappement spéciales et les codes de contrôle pourraient provoquer un affichage erratique sur le terminal. Si vous êtes dans un environnement web et que vous recrachez sans précaution tout ce que l'on vous donne, vous pourriez produire de manière imprévisible des balises HTML qui déformeraient drastiquement l'apparence de la page. Pire encore, certaines balises peuvent même en retour exécuter du code dans le navigateur.

Imaginez le cas courant d'un livre d'or où les visiteurs entrent leurs propres messages à afficher lorsque d'autres viennent appeler. Un invité malicieux pourrait fournir des balises HTML disgracieuses ou mettre des séquences `<SCRIPT>...</SCRIPT>` exécutant du code (comme du JavaScript) en retour dans le navigateur des visiteurs suivants.

Tout comme vous avez vérifié que tous les caractères sont autorisés, lorsque vous recevez une donnée marquée qui pourrait accéder aux ressources de votre système, vous pouvez effectuer une vérification analogue dans un environnement web lorsque vous recevez des données d'un internaute. Par exemple, pour supprimer tous les caractères absentes de la liste des bons caractères, essayez quelque chose comme :

```
$nouvelle_entree_du_livre_d_or =~ tr[_a-zA-Z0-9 ,.!?()@+*-][]dc;
```

Vous n'utiliseriez certainement pas cela pour nettoyer un nom de fichier, puisque vous ne voulez probablement pas de noms de fichiers avec des espaces ou des slashes, pour commencer. Mais il n'est pas suffisant de préserver votre livre d'or vierge de toute balise HTML sournoise et de toute entité perfide. Chaque cas de blanchiment de donnée est quelque peu différent, passez donc toujours du temps à décider ce qui est permis et ce

qui ne l'est pas. Le mécanisme de marquage est prévu pour intercepter les erreurs stupides, pas pour se dispenser de réfléchir.

Nettoyage de votre environnement

Lorsque vous exécutez un autre programme depuis votre script Perl, peu importe comment, Perl vérifie pour s'assurer que votre variable d'environnement `PATH` soit sécurisée. Puisqu'il provient de votre environnement, votre `PATH` commence par être marqué, ainsi si vous essayez de lancer un autre programme, Perl lève une exception « `Insecure $ENV{PATH}` ». Lorsque vous le positionnez à une valeur connue, non marquée, Perl s'assure que chaque répertoire dans ce `PATH` ne soit pas accessible en écriture par quelqu'un d'autre que le propriétaire et le groupe du répertoire ; sinon, Perl lance une exception « `Insecure directory` ».

Vous seriez surpris d'apprendre que Perl se préoccupe de votre `PATH` même lorsque vous spécifiez le nom de chemin complet de la commande que vous voulez exécuter. Il est vrai qu'avec un nom de fichier absolu, le `PATH` n'est pas utilisé pour localiser l'exécutable. Mais Perl se méfie de ce que le programme exécuté pourrait en lancer un *autre* avec un `PATH` non sûr. Perl vous force donc à positionner un `PATH` sécurisé avant d'appeler un autre programme, quelle que soit la manière dont vous l'appelez.

Le `PATH` n'est pas la seule variable d'environnement pouvant causer du chagrin. Comme certains shells utilisent les variables `IFS`, `CDPATH`, `ENV` et `BASH_ENV`, Perl s'assure que toutes celles-ci sont soit vides, soit sans marquage avant de lancer une autre commande. Positionnez ces variables à quelque chose que l'on sait être sûr ou alors supprimez-les de l'environnement :

```
delete ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Rend %ENV plus sûr
```

Des fonctionnalités accommodantes dans un environnement normal peuvent devenir des problèmes de sécurité dans un environnement hostile. Même si vous vous rappelez avoir interdit les noms de fichiers contenant des sauts de ligne, il est important de comprendre que `open` accède à autre chose de plus que seulement des noms de fichiers. Avec les ornements appropriés dans l'argument représentant le nom de fichier, les appels à `open` avec un ou deux arguments peuvent également lancer des commandes arbitraires externes *via* des pipes, « forker » des copies supplémentaires du processus courant, dupliquer des descripteurs de fichiers et interpréter le nom de fichier spécial « `-` » comme un alias de l'entrée ou de la sortie standard. Il peut également ignorer les espaces au début ou à la fin qui peuvent masquer de tels argument fantaisistes à vos motifs de test. Bien qu'il soit vrai que les vérifications de marquage de Perl intercepteront les arguments marqués utilisés dans les ouvertures de pipe (sauf si vous utilisez un argument de liste séparé) et toutes les ouvertures de fichiers qui ne soient pas en lecture seule, l'exception levée par ceci est toujours susceptible de faire que votre programme se conduise mal.

Si vous avez l'intention d'utiliser une donnée dérivée de l'extérieur dans une partie d'un nom de fichier à ouvrir, incluez au moins un mode explicite séparé par un espace. Cependant, le plus sûr est certainement d'utiliser un `sysopen` de bas niveau ou la forme à trois arguments de `open` :

```
# Ouverture magique --- peut être n'importe quoi
open(HF, $fichier)      or die "open $fichier magique, impossible : $!";
```

```
# Garantit l'ouverture d'un fichier en lecture seule et non un pipe
# ou un fork, mais interprète toujours les descripteurs de
# fichiers et « - » et ignore les espaces aux extrémités du nom.
open(HF, "< $fichier)   or die "open $fichier, impossible : $!";

# Ouverture WYSIWYG : désactive toutes les fonctionnalités accommodantes.
open(HF, "<", $fichier) or die "open $fichier, impossible : $!";

# Mêmes propriétés que la version à 3 arguments WYSIWYG.
require Fcntl;
sysopen(HF, $fichier, O_RDONLY) or die "sysopen $fichier, impossible : $!";
```

Et même ces étapes ne sont pas assez bonnes. Perl ne vous empêche pas d'ouvrir des noms de fichiers marqués en lecture, vous devez donc être prudent avec ce que vous montrez aux gens. Un programme qui ouvre en lecture un nom de fichier arbitraire, fourni par l'utilisateur, pour ensuite révéler le contenu de ce fichier, est toujours un problème de sécurité. Que se passe-t-il s'il s'agit d'une lettre privée ? Et s'il s'agit du fichier de mots de passe de votre système ? Ou des informations sur les salaires ou votre portefeuille d'actions ?

Examinez de près les noms de fichier fournis par un utilisateur potentiellement hostile[6] avant de les ouvrir. Par exemple, vous pourriez vouloir vérifier qu'il n'y a pas de répertoires sournois dans votre `PATH`. Les noms comme « `../../../../../../../etc/passwd` » sont des coups fourrés du genre qui sont de notoriété publique. Vous pouvez vous protéger en vous assurant qu'il n'y a pas de slash dans le nom du chemin (en supposant qu'il s'agit du séparateur de répertoire de votre système). Une autre astuce courante est de mettre des sauts de ligne ou des points-virgules dans les noms de fichiers qui seront interprétés ensuite par de pauvres interpréteurs de commandes simples d'esprit, pouvant être bluffés en commençant une nouvelle commande au beau milieu du nom de fichier. C'est pourquoi le mode de marquage déconseille les commandes externes non inspectées.

Accès aux commandes et aux fichiers avec des privilèges restreints

La discussion qui suit est relative à des facilités astucieuses concernant la sécurité sur les systèmes Unix. Les utilisateurs d'autres systèmes peuvent en toute sécurité (ou plutôt en toute insécurité) passer ce paragraphe.

Si vous tournez avec set-id, essayez d'arranger cela, à chaque fois que c'est possible, pour que les opérations dangereuses soient accomplies avec les privilèges de l'utilisateur et non ceux du programme. C'est-à-dire qu'à chaque fois que vous allez appeler `open`, `sysopen`, `system`, des apostrophes inverses et tout autre opération sur un fichier ou sur un processus, vous pouvez vous protégez en repositionnant votre UID ou votre GID effectif avec l'UID ou le GID réel. En Perl, vous pouvez faire cela pour les scripts setuid en écrivant `$> = $<` (ou `$EUID = $UID` si vous avez fait `use English`) et pour les scripts setgid en écrivant `$( = $)` (`$EGID = $GID`). Si les deux ID sont positionnés, vous devez réini-

6. Et sur le Net, les seuls utilisateurs que vous pouvez en toute confiance considérer comme n'étant pas potentiellement hostiles sont plutôt ceux qui sont *activement* hostiles.

tialiser les deux. De toute façon, c'est parfois infaisable car vous pourriez encore avoir besoin de ces privilèges accrus ultérieurement dans votre programme.

Pour ces cas-là, Perl procure un moyen relativement sûr d'ouvrir un fichier ou un pipe depuis un programme set-id. Tout d'abord, lancez un fils avec la syntaxe spéciale de `open` qui connecte le père et le fils par un pipe. Dans le fils, repositionnez ensuite ses ID d'utilisateur et de groupe à leur valeur originale ou à des valeurs connues et sûres. Vous devrez également modifier tous les attributs de processus du fils sans affecter le père, vous permettant de changer le répertoire de travail courant, positionner le masque de création de fichiers ou bricoler les variables d'environnement. Le processus fils, qui ne s'exécute plus sous des privilèges accrus, appelle finalement `open` et passe les données auxquelles il a accédé sous couvert d'un utilisateur ordinaire mais dément, à son père, puissant mais paranoïaque de manière justifiée.

Bien que `system` et `exec` n'utilisent pas le shell lorsque vous leur fournissez plus d'un argument, l'opérateur apostrophes inverses n'admet pas de telle convention d'appel alternative. En utilisant la technique lançant un processus fils, nous pouvons facilement émuler les apostrophes inverses sans nous soucier des séquences d'échappements du shell et avec des privilèges réduits (et donc plus sûrs) :

```
use English;       # pour employer $UID, etc.
die "open fork impossible : $!"
    unless defined ($pid = open(DEPUIS_FILS, "-|"));
if ($pid) {        # père
    while (<DEPUIS_FILS>) {
        # faire quelque chose
    }
    close DEPUIS_FILS;
} else {
    $EUID = $UID;  # setuid(getuid())
    $EGID = $GID;  # setgid(getgid()) et initgroups(2) sur getgroups(2)
    chdir("/")      or die "chdir à / impossible : $!";
    umask(077);
    $ENV{PATH} = "/bin:/usr/bin";
    exec 'mon_prog', 'arg1', 'arg2';
    die "exec mon_prog impossible : $!";
}
```

Il s'agit du meilleur moyen, et de loin, d'appeler un programme depuis un script set-id. Vous êtes sûr de ne pas utiliser le shell pour exécuter quoi que ce soit et vous abandonnez vos privilèges avant d'appeler vous-même `exec` pour lancer le programme. (Mais puisque les formes avec liste de `system`, `exec` et `open` sur un pipe sont spécifiquement exemptes des vérifications de marquage, vous devez toujours prendre garde à ce que vous leur passez.)

Si nous n'avez pas besoin d'abandonner vos privilèges et voulez seulement implémenter les apostrophes inverses ou un `open` sur un pipe sans risquer que le shell intercepte vos arguments, il vous suffit d'utiliser ceci :

```
open(DEPUIS_FILS, "-|") or exec("mon_prog", "arg1", "arg2")
    or die "impossible de lancer mon_prog : $!";
```

puis de lire dans `DEPUIS_FILS` dans le père. Avec la version 5.6.1 de Perl, vous pouvez l'écrire comme ceci :

```
open(DEPUIS_FILS, "-|", "mon_prog", "arg1", "arg2");
```

La technique consistant à lancer un fils n'est pas seulement utile pour lancer des commandes depuis un programme set-id. Elle s'avère également bonne pour ouvrir des fichiers sous l'ID de quiconque a lancé le programme. Imaginez que vous avez un programme setuid qui a besoin d'ouvrir un fichier en écriture. Vous ne voulez pas lancer le open avec vos privilèges supplémentaires mais vous ne pouvez pas non plus les abandonner définitivement. Arrangez-vous alors pour avoir une copie « forkée » qui abandonne ses privilèges pour effectuer le open pour vous. Lorsque vous voulez écrire dans le fichier, écrivez au fils et ce dernier écrira dans le fichier à votre place.

```
use English;

defined ($pid = open(ECRIVAIN_SUR, "|-"))
    or die "fork impossible : $!";

if ($pid) {
    # Vous êtes dans le père.
    # Écrivez les données au fils ECRIVAIN_SUR.
    print ECRIVAIN_SUR "donnes_en_sortie\n";
    close ECRIVAIN_SUR
        or die $! ? "Erreur système en fermant ECRIVAIN_SUR : $!"
                  : "Statut d'attente $? depuis ECRIVAIN_SUR";
} else {
    # Vous êtes dans le fils.
    # Abandonnez donc vos privilèges supplémentaires.
    ($EUID, $EGID) = ($UID, $GID);

    # Ouvrez le fichier avec les droits de l'utilisateur original.
    open(HF, ">/un/chemin/de/fichier"
        or die "open /un/chemin/de/fichier en écriture, impossible : $!";

    # Copiez depuis le père (maintenant stdin) vers le fichier.
    while (<STDIN>) {
        print HF, $_;
    }
    close(HF)   or die "Échec de close : $!";
    exit;        # N'oubliez pas de faire disparaître ECRIVAIN_SUR.
}
```

Lorsqu'il échoue dans l'ouverture du fichier, le fils affiche un message d'erreur et se termine. Lorsque le père écrit dans le handle de fichier du fils maintenant décédé, cela déclenche un signal de pipe brisé (SIGPIPE) qui est fatal s'il n'est pas intercepté ou ignoré. Voir la section *Signaux* au chapitre 16, *Communication interprocessus*.

Gestion des problèmes de synchronisation

Parfois, le comportement de vos programmes est très sensible à la synchronisation d'événements extérieurs hors de votre contrôle. Il s'agit toujours d'un problème lorsque d'autres programmes, particulièrement ceux qui sont inamicaux, pourraient se battre avec votre programme pour les mêmes ressources (comme des fichiers ou des périphé-

riques). Dans un environnement multitâche, vous ne pouvez pas prédire l'ordre dans lequel les processus en attente d'activation vont être autorisés à accéder au processeur. Les flux d'instructions parmi tous les processus éligibles s'entrecroisent, ainsi un processus dispose tout d'abord d'un peu de CPU, ensuite un autre, et ainsi de suite. Il semble aléatoire de savoir celui dont c'est le tour d'être actif et combien de temps il sera autorisé à l'être. Avec un seul programme, il n'y a pas de problème mais avec plusieurs, partageant des ressources communes, cela peut en être un.

Les programmeurs de « threads » sont particulièrement sensibles à ces questions. Ils apprennent vite à ne plus écrire :

```
$var++ if $var == 0;
```

lorsqu'ils veulent dire :

```
{
    lock($var);
    $var++ if $var == 0;
}
```

Le premier exemple donne des résultats imprévisibles lorsque plusieurs threads d'exécution tente de lancer ce code en même temps. (Voir le chapitre 17, *Threads*.) Si vous pensez aux fichiers comme à des objets partagés et aux processus comme à des threads candidats pour l'accès à ces objets partagés, vous pourrez voir comment les mêmes questions sont soulevées. Un processus, après tout, n'est en réalité qu'un thread qui s'y croit. Ou vice versa.

L'imprévisibilité des synchronisations affecte à la fois les situations privilégiées et celles qui ne le sont pas. Nous décrirons tout d'abord comment se débrouiller avec un bogue de longue date dans les anciens noyaux Unix affectant tous les programmes set-id. Nous irons ensuite discuter des situations de concurrence en général, comment elles se transforment en brèches de sécurité et les étapes que vous devez suivre pour éviter de tomber dans ces trous.

Bogues de sécurité du noyau Unix

Au-delà des problèmes évidents découlant du passage de privilèges spéciaux à des interpréteurs aussi souples et impénétrables que les shells, les anciennes versions d'Unix souffrent d'un bogue du noyau qui rend tout script set-id non sûr avant même d'arriver jusqu'à l'interpréteur. Le problème ne réside pas dans le script lui-même mais dans une situation de concurrence dans ce que fait le noyau lorsqu'il trouve un script set-id exécutable. (Le bogue n'existe pas sur les machines qui ne reconnaissent pas `#!` dans le noyau.) Entre le moment où le noyau ouvre un tel fichier pour savoir quel interpréteur lancer et celui où l'interpréteur (maintenant set-id) démarre et ouvre à nouveau le fichier, il existe une chance pour des entités malicieuses de changer le fichier en question, surtout si le système implémente les liens symboliques.

Heureusement, cette « caractéristique » du noyau peut parfois être désactivée. Malheureusement, il existe deux moyens pour ce faire. Le système peut rendre hors-la-loi les scripts avec les bits set-id positionnés, ce qui n'est pas d'une grande aide. Il peut également ignorer les bits set-id sur les scripts. Dans ce dernier cas, Perl peut émuler les mécanismes de setuid et de setgid quand il remarque les bits set_id (par ailleurs inutiles)

sur les scripts Perl. Il effectue ceci grâce à un exécutable spécial appelé *suidperl*, qui est automatiquement appelé pour vous si besoin.[7]

Cependant, si la fonctionnalité du noyau concernant les scripts set-id n'est *pas* désactivée, Perl se plaint bruyamment de ce que le script n'est pas sûr. Vous devrez soit la désactiver, soit enrober le script dans un programme C. Il s'agit seulement d'un programme compilé qui ne fait rien d'autre qu'appeler votre programme Perl. Les programmes compilés ne sont pas sujets au bogue qui tourmente les scripts set-id.

Voici un simple enrobage écrit en C :

```
#define VRAI_FICHIER "/chemin/du/script"
main(ac, av)
    char **av;
{
    execv(VRAI_FICHIER, av);
}
```

Compilez cet enrobage en un exécutable binaire et rendez-*le* set-id au lieu de le faire sur votre script. Assurez-vous d'employer un nom de chemin absolu, car le C n'est pas assez malin pour vérifier le marquage sur votre `PATH`.

(Une autre approche possible consiste à employer le générateur expérimental de code C du compilateur Perl. Un image compilée de votre script ne connaîtra pas la situation de concurrence. Voir au chapitre 18, *Compiler*.)

Ces dernières années, certains fournisseurs ont finalement sorti des systèmes libérés de ce bogue de set-id. Sur de tels systèmes, lorsque le noyau passe le nom du script set-id à l'interpréteur, il ne s'agit plus d'un nom de fichier sujet à des interventions, mais d'un fichier spécial représentant le descripteur de fichier, comme */dev/fd/3*. Ce fichier spécial est déjà ouvert sur le script de manière à ce qu'il ne puisse plus avoir de situation de concurrence que des scripts diaboliques pourraient exploiter.[8] La plupart des versions modernes d'Unix emploient cette approche pour éviter la situation de concurrence inhérente à l'ouverture d'un même fichier deux fois.

Gestion des situations de concurrence

Ce qui nous amène directement au thème des situations de concurrence. Que sont-elles réellement ? Les situations de concurrence se retrouvent fréquemment dans les discussions à propos de la sécurité. (Bien qu'on les retrouvent moins que dans les programmes non sûrs. Malheureusement.) C'est parce qu'elles sont une source fertile d'erreurs subtiles de programmation et que de telles erreurs peuvent souvent se transformer en *exploits* (terme poli pour désigner le fait de bousiller la sécurité de quelqu'un) pour la

7. Si besoin *et* si c'est permis — si Perl détecte que le système de fichiers dans lequel réside le script a été monté avec l'option `nosuid`, cette dernière sera toujours respectée. Vous ne pouvez pas utiliser Perl pour coutourner de cette façon la politique de sécurité de votre administrateur système.

8. Sur ces systèmes, Perl doit être compilé avec **`-DSETUID_SCRIPTS_ARE_SECURE_NOW`**. Le programme *Configure* qui compile Perl essaie de deviner cette situation, ce qui fait que vous ne devriez jamais le spécifier explicitement.

sécurité. Une situation de concurrence arrive lorsque le résultat de plusieurs événements corrélés dépend de l'ordre de ces événements mais que cet ordre ne peut être garanti à cause des effets d'une synchronisation non déterministe. Chaque événement se précipite pour être le premier accompli et l'état final du système est laissé à l'imagination de chacun.

Imaginez que vous ayez un processus écrivant par dessus un fichier existant et un autre processus lisant le même fichier. Vous ne pouvez prédire si vous lisez les anciennes données, les nouvelles ou un mélange aléatoire des deux. Vous ne pouvez même pas savoir si vous avez lu toutes les données. Le lecteur pourrait avoir gagné la course jusqu'à la fin du fichier avant de se terminer. En même temps, si l'écrivain a continué après que le lecteur a rencontré la fin de fichier, le fichier aura grossi plus loin que l'endroit où le lecteur s'était arrêté et ce dernier ne l'aura jamais su.

Ici, la solution est simple : il suffit que les deux parties verrouillent le fichier avec `flock`. Le lecteur demande généralement un verrou partagé et l'écrivain un verrou exclusif. Tant que toutes les parties demandent et respectent ces verrous consultatifs, les lectures et écritures ne peuvent pas s'enchevêtrer et il n'y aucun chance de trouver des données mutilées. Voir la section *Verrouillage de fichier* au chapitre 16.

Vous risquez de rencontrer une forme bien moins évidente de situation de concurrence chaque fois que vous laissez des opérations sur des noms de fichier gouverner les opérations ultérieures sur ce fichier. Lorsqu'on les utilise sur des noms de fichier plutôt que sur des descripteurs de fichier, les opérateurs de test de fichier représentent parfois un terrain fertile menant droit à une situation de concurrence. Considérez ce code :

```
if (-e $fichier) {
    open(HF, "< $fichier")
        or die "impossible d'ouvrir $fichier en lecture : $!";
} else {
    open(HF, "> $fichier")
        or die "impossible d'ouvrir $fichier en écriture : $!";
}
```

Le code paraît aussi simple qu'il ne l'est, mais il est toujours sujet à des situations de concurrence. Il n'y a aucune garantie que la réponse renvoyée par le test **`-e`** soit toujours valide au moment où l'un des `open` est appelé. Dans le bloc `if`, un autre processus pourrait avoir supprimé le fichier avant qu'il ne puisse être ouvert et vous ne trouveriez pas le fichier dont vous pensiez qu'il allait se trouver là. Dans le bloc `else`, un autre processus pourrait avoir créé le fichier avant que le tour du second `open` n'arrive pour qu'il puisse le créer et le fichier que vous pensiez donc ne pas être là, le serait. La simple fonction `open` crée de nouveaux fichiers mais écrase ceux qui existaient. Vous pouvez penser que vous voulez écraser tout fichier existant mais penser que le fichier existant pourrait très bien être un alias nouvellement créé ou un lien symbolique vers un fichier autre part sur le système que vous ne voudriez vraiment pas écraser. Vous pouvez penser que vous savez ce que signifie un nom de fichier à n'importe quel instant mais vous ne pouvez jamais en être vraiment sûr, tant que d'autres processus ayant accès au répertoire du fichier tournent sur le même système.

Pour résoudre ce problème d'écrasement, vous devrez employer `sysopen`, qui fournit des contrôles individuels pour soit créer un nouveau fichier, soit pour en écraser complètement un qui existe. Et nous abandonnerons ce test **`-e`** d'existence de fichier puis-

qu'il n'est plus utile ici et ne fait qu'accroître notre exposition aux situations de concurrence.

```
use Fcntl qw/O_WRONLY O_CREAT O_EXCL/;
open (HF, "<", $fichier)
    or sysopen(HF, $fichier, O_WRONLY | O_CREAT | O_EXCL)
    or die "impossible de créer le nouveau fichier $fichier : $!";
```

Maintenant, même si le fichier se met à éclore d'une manière ou d'une autre entre le moment où le `open` échoue et celui où le `sysopen` essaie d'ouvrir un nouveau fichier en écriture, aucun mal ne peut être fait car avec les flags que nous lui avons fournis, `sysopen` refusera d'ouvrir un fichier qui existe déjà.

Si quelqu'un essaie de persuader par la ruse votre programme de mal se conduire, il a une bonne chance de réussir en faisant apparaître et disparaître les fichiers quand vous ne vous y attendez pas. Un moyen de réduire le risque de tromperie consiste à vous promettre à vous-même que vous ne manipulerez jamais un nom de fichier plus d'une fois. Dès que vous avez ouvert le fichier, oubliez son nom (à part peut-être pour les messages d'erreur) et ne manipulez plus que le handle représentant le fichier. C'est beaucoup plus sûr car, même si quelqu'un pourrait s'amuser avec vos noms de fichier, il ne peut le faire avec les handles. (Ou s'il le peut, c'est parce que vous l'avez laissé faire — voir *Passage de handles de fichiers* au chapitre 16.)

Plus haut dans ce chapitre, nous avons présenté une fonction `handle_semble_sur` qui appelait la fonction `stat` de Perl sur un handle de fichier (et non un nom de fichier) pour vérifier son appartenance et ses permissions. L'emploi du handle de fichier est critique pour l'exactitude de la solution — si nous avions passé le nom du fichier, il n'y aurait eu aucune garantie que le fichier dont nous examinions les attributs ait été le même que celui que nous venions d'ouvrir (ou que nous étions sur le point d'ouvrir). Un minable mécréant aurait pu supprimer notre fichier et le remplacer rapidement par un fichier subversif, à un moment donné entre le `stat` et le `open`. Peut importe lequel aurait été appelé en premier ; il y aurait toujours une opportunité pour un jeu déloyal entre les deux. Vous pouvez penser que le risque est très réduit puisque la fenêtre l'est aussi mais il existe de nombreux scripts pirates dans le monde qui seraient tout à fait heureux de lancer votre programme des milliers de fois pour s'en jouer la seule fois où il n'a pas été assez prudent. Un script pirate habile peut même baisser la priorité de votre programme pour qu'il soit interrompu plus souvent que d'ordinaire, simplement pour accélérer un peu les choses. Les gens travaillent dur sur ces questions — c'est pourquoi on les appelle des *exploits*.

En appelant `stat` sur un handle de fichier déjà ouvert, nous n'accédons au nom du fichier qu'une seule fois et évitons ainsi la situation de concurrence. Une bonne stratégie pour éviter les situations de concurrence entre deux événements consiste à combiner les deux d'une manière ou d'une autre pour n'en faire qu'un seul événement, en rendant l'opération atomique.[9] Puisque nous n'accédons au fichier qu'une seule fois par son

9. Oui, vous pouvez encore accomplir des opérations atomiques dans une zone dénucléarisée. Lorsque Démocrite a donné le nom « atome » aux parties indivisibles de la matière, il voulait littéralement dire quelque chose qui ne peut être divisé : *a-*(non) + *tomos* (divisible). Une opération atomique est une action qui ne peut être interrompue. (Essayez seulement d'interrompre un jour une bombe atomique.)

nom, il ne peut y avoir de situation de concurrence entre plusieurs accès, ainsi ça ne fait pas grand chose si le nom change. Même si notre pirate supprime le fichier que nous avons ouvert (oui, cela peut arriver) et le remplace par un autre pour s'amuser avec nous, nous avons toujours un handle sur le fichier réel et original.

Fichiers temporaires

À part autoriser les dépassements de tampons (ce contre quoi les scripts Perl sont virtuellement immunisés) et faire confiance à des données en entrée indignes de confiance (ce contre quoi le mode de marquage préserve), créer de manière incorrecte des fichiers temporaires est l'un des trous de sécurité les plus fréquemment exploités. Heureusement, les attaques contre les fichiers temporaires exigent des pirates qu'ils aient un compte d'utilisateur valide sur le système qu'ils tentent de pirater, ce qui réduit drastiquement le nombre de mécréants potentiels.

Les programmes imprudents ou désinvoltes utilisent des fichiers temporaires avec toutes sortes de moyens non sûrs, comme en les plaçant dans des répertoires accessibles en écriture par le monde entier, en utilisant des noms de fichier prévisibles et en ne s'assurant pas que le fichier n'existe pas déjà. Dès que vous trouvez un programme avec du code comme ceci :

```
open(TMP, ">/tmp/truc.$$")
    or die "impossible d'ouvrir /tmp/truc.$$ : $!";
```

vous venez de trouver les trois erreurs d'un coup. Ce programme est un accident en sursis.

Le moyen mis en œuvre par l'exploit consiste pour le pirate à tout d'abord planter un fichier du même nom que celui que vous utiliserez. Ajouter le PID à la fin ne suffit pas à garantir l'unicité ; aussi surprenant que cela puisse paraître, ce n'est vraiment pas difficile deviner des PID.[10] Maintenant arrive le programme avec l'appel imprudent à `open` et au lieu de créer un nouveau fichier temporaire pour ses propres besoins, il écrase celui du pirate à la place.

Quel mal cela peut-il donc faire ? Beaucoup ! Le fichier du pirate n'est pas vraiment un fichier pur, voyez-vous. C'est un lien symbolique (ou parfois un lien en dur), pointant certainement vers un fichier critique dans lequel le pirate ne pourrait normalement pas écrire tout seul, comme */etc/passwd*. Le programme pensait avoir ouvert un fichier tout neuf dans */tmp* mais à la place, il a écrasé un fichier existant quelque part ailleurs.

Perl offre deux fonctions abordant ce problème, si elles sont utilisées correctement. La première est `POSIX::tmpnam`, qui ne fait que renvoyer le nom de fichier que vous escomptiez ouvrir par vous-même :

```
# Essaie des noms jusqu'à en obtenir un qui soit tout neuf
use POSIX;
do {
    $nom = tmpnam();
} until sysopen(TMP, $nom, O_RDWR | O_CREAT | O_EXCL, 0600);
# Maintenant faites des E/S en utilisant le handle TMP.
```

10. Sauf si vous êtes sur un système comme OpenBSD, qui affecte de nouveaux PID de manière aléatoire.

La seconde est IO::FILE::new_tmpfile, qui vous redonne un handle déjà ouvert :

```
# Ou sinon laissez le module le faire à votre place.
use IO::File;
my $hf = IO::File::new_tmpfile();    # Il s'agit de tmpfile(3) de POSIX
# Maintenant faites des E/S en utilisant le handle $hf.
```

Aucune des deux approches n'est parfaite, mais entre les deux, la première est la meilleure. Le problème majeur avec la seconde est que Perl est tributaire des faiblesses de l'implémentation de *tmpfile*(3) dans la bibliothèque C de votre système et que vous n'avez aucune garantie que cette fonction ne fasse pas quelque chose d'aussi dangereux que le open que nous essayons de corriger. (Et c'est le cas, assez tristement, de certaines implémentations.) Un problème mineur est que cela ne vous donne pas du tout le nom du fichier. Bien que ce soit mieux si vous pouvez manipuler un fichier temporaire sans connaître son nom — car de cette manière vous ne provoquerez jamais de situation de concurrence en essayant de l'ouvrir à nouveau — bien souvent, vous ne pouvez pas.

Le problème majeur avec la première approche est que vous n'avez aucun contrôle sur l'endroit où se trouve le chemin du fichier, au contraire de la fonction *mkstemp*(3) de la bibliothèque C. Une première raison est qu'il faut éviter de mettre le fichier sur un système de fichiers monté en NFS. Il n'est pas garanti que le flag O_EXCL fonctionne correctement sous NFS, plusieurs processus demandant une création exclusive pratiquement au même moment peuvent donc tous réussir. Une autre raison est que le chemin renvoyé se situe probablement dans un répertoire où les autres ont les droits en écriture, quelqu'un pourrait avoir planté un lien symbolique pointant vers un fichier inexistant, vous forçant à créer votre fichier à un endroit qu'il préfère.[11] Si vous avez votre mot à dire là-dessus, ne mettez pas vos fichiers temporaires dans un répertoire où quelqu'un d'autre peut écrire. Si vous y êtes obligé, assurez-vous d'utiliser le flag O_EXCL dans le sysopen et essayez d'employer des répertoires avec le flag « seul-le-propriétaire-peut-effacer » (le « sticky bit ») positionné.

Depuis la version de Perl 5.6.1, il existe une troisième manière. Le module standard File::Temp prend en compte toutes les difficultés que nous avons mentionnées. Vous pourriez utiliser les options par défaut ainsi :

```
use File::Temp "tempfile";
$handle = tempfile();
```

Ou vous pourriez spécifier certaines options ainsi :

```
use File::Temp "tempfile";
($handle, $nom_fichier) = tempfile("priseXXXXXX",
                                   DIR => "/var/spool/aventure",
                                   SUFFIX = '.dat');
```

Le module File::Temp fournit également des émulations se préoccupant de la sécurité pour les autres fonctions que nous avons mentionnées (bien que l'interface native soit meilleure car elle vous donne un handle de fichier ouvert et pas seulement un nom de fichier, qui est sujet à des situations de concurrence). Voir le chapitre 32, *Modules stan-*

11. Une solution à ceci, qui ne fonctionne que sous certains systèmes d'exploitation, consiste à appeler sysopen en utilisant un OU avec le flag O_NOFOLLOW. Ceci fait échouer la fonction si le composant final du chemin est un lien symbolique.

dards, pour une plus longue description des options et de la sémantique de ce module.

Une fois que vous avez votre handle de fichier, vous pouvez en faire tout ce que vous voulez. Il est ouvert à la fois en lecture et en écriture, vous pouvez donc écrire dans le handle, faire un `seek` pour revenir au début et ensuite, si vous le voulez, écraser tout ce que vous veniez d'y mettre ou le lire à nouveau. La chose que vous voudrez vraiment, *vraiment* éviter de faire est de jamais ouvrir ce fichier à nouveau, car vous ne pouvez être sûrs qu'il s'agisse vraiment du même fichier que vous aviez ouvert la première fois.[12]

Lorsque vous lancez un autre programme depuis votre script, Perl ferme normalement pour vous tous les handles de fichiers pour éviter une autre vulnérabilité. Si vous utilisez `fcntl` pour nettoyer votre flag close-on-exec (comme dans la description d'`open` au chapitre 29, *Fonctions*), les autres programmes que vous appelez hériteront de ce nouveau descripteur de fichier ouvert. Sur les systèmes qui implémentent le répertoire */dev/fd*, vous pourriez fournir à un autre programme un nom de fichier qui soit vraiment le descripteur de fichier en le construisant de cette façon :

```
$nom_virtuel = "/dev/fd" . fileno(TMP);
```

Si vous n'aviez besoin que d'appeler un sous-programme Perl ou un programme qui attend un nom de fichier comme argument et si vous saviez que ce sous-programme ou que ce programme l'utilisait avec un `open` ordinaire, vous pourriez alors passer le handle de fichier en employant la notation de Perl pour indiquer un handle de fichier :

```
$nom_virtuel = "=&" . fileno(TMP);
```

Lorsque le « nom » de fichier est passé avec un `open` ordinaire en Perl à un ou deux arguments (et non trois, ce qui enlèverait toute cette magie bien utile), vous obtenez l'accès au descripteur dupliqué. D'une certaine manière, ceci est plus portable que de passer un fichier depuis */dev/fd* car cela fonctionne partout où Perl fonctionne ; tous les systèmes n'ont pas de répertoire */dev/fd*. D'un autre côté, la syntaxe spéciale d'`open` en Perl pour accéder aux descripteurs de fichiers par des nombres ne fonctionne qu'avec des programmes Perl et non avec des programmes écrits dans d'autres langages.

Gestion du code non sûr

La vérification de marquage n'est qu'une sorte de couverture de survie dont vous avez besoin si vous voulez débusquer des données fausses que vous auriez dû déceler vous-même mais que vous n'avez pas pu dépister avant de les abandonner au système. C'est un peu comme les avertissements optionnels que Perl peut vous donner — ils peuvent ne pas indiquer de problème réel mais en moyenne le mal que l'on se donne à gérer les fausses alertes est moindre que celui de ne pas détecter les vrais problèmes. Avec le marquage, ce dernier mal est encore plus insistant car l'utilisation de données fausses ne donne pas seulement les mauvaises réponses ; cela peut saccager votre système et vos deux dernières années de travail avec. (Et peut-être vos deux prochaines si vous n'avez pas fait de bonnes sauvegardes.) Le mode de marquage est utile lorsqu'il s'agit d'avoir

12. Excepté après coup, en faisant un `stat` sur les deux handles de fichiers et en comparant le deux premières valeurs renvoyées pour chacun (la paire périphérique/inode). Mais c'est trop tard car le mal est déjà fait. Tout ce que vous pouvez faire est de constater les dégâts et d'abandonner (et peut-être envoyer furtivement un email à l'administrateur système).

confiance en vous-même pour écrire du code honnête mais ne que vous ne faites pas systématiquement confiance à quiconque vous passe des données pour ne pas essayer de vous pousser à faire quelque chose de regrettable.

Les données sont une chose. Mais c'est tout à fait une autre affaire lorsque vous ne pouvez même pas faire confiance au code que vous êtes en train d'exécuter. Et si l'applet que vous avez récupérée sur le Net contenait un virus, une bombe à retardement ou un cheval de Troie ? La vérification de marquage est ici inutile car, alors que les données qui sont fournies au programme peuvent être très bien, c'est le code qui est indigne de confiance. Vous vous mettez dans la situation de quelqu'un qui reçoit un appareil mystérieux de la part d'un étranger, avec une note disant en substance : « Placez ceci contre votre crâne et appuyez sur la détente. » Vous pouvez penser que cela va sécher vos cheveux, mais peut-être plus pour très longtemps.

Dans ce domaine, la prudence est synonyme de paranoïa. Il vous faut un système qui vous permette de placer le code suspect en quarantaine. Il peut continuer à exister et même accomplir certaines fonctions mais vous ne le laissez pas se promener et faire tout ce dont il a envie. En Perl, vous pouvez imposer un genre de quarantaine en utilisant le module `Safe`.

Compartiments sécurisés

Le module `Safe` vous laisse mettre en place un bac à sable (*sandbox*), un compartiment spécial dans lequel toutes les opérations système sont interceptées et où l'accès à l'espace de noms est strictement contrôlé. Les détails techniques de bas niveau de ce module sont changeants nous adopterons donc ici une approche plus philosophique.

Restriction de l'accès à l'espace de noms

Au niveau le plus fondamental, un objet `Safe` ressemble à un coffre-fort sauf que l'idée de base est de laisser les truands à l'intérieur et non à l'extérieur. Dans le monde Unix, il existe un appel système connu sous le nom de *chroot*(2) qui peut consigner en permanence un processus dans un sous-répertoire du système de fichiers — dans son enfer personnel si vous voulez. Une fois que le processus y est installé, il ne peut absolument pas atteindre de fichiers à l'extérieur de ce répertoire car il ne sait rien pas *nommer* de fichiers se trouvant à l'extérieur.[13] Un objet `Safe` ressemble à cela mais au lieu d'être restreint à un sous-ensemble de la structure de répertoires du système de fichiers, il est restreint à un sous-ensemble de la structure des paquetages de Perl, qui est tout aussi hiérarchique que la structure du système de fichiers.

Une autre manière de voir un objet `Safe` est quelque peu similaire à ces salles d'observations avec des miroirs sans tain dans lesquelles la police met ses suspects. Les gens à l'extérieur peuvent regarder ceux dans la salle mais non l'inverse.

Lorsque vous créez un objet `Safe`, vous pouvez lui donner un nom de paquetage si vous le voulez. Sinon, un nouveau nom sera choisi pour vous :

13. Certains sites font cela pour exécuter des scripts CGI, en utilisant des montages en boucle et en lecture seule. C'est quelque peu ennuyeux à configurer mais si quelqu'un s'échappe un jour, il découvrira qu'il n'a nulle part où aller.

```
use Safe;
my $salle = Safe->new("Donjon");
$Donjon::truc = 1;    # Pourtant, l'accès direct est déconseillé.
```

Si vous qualifiez pleinement les variables et les fonctions utilisant le nom de paquetage donné à la méthode `new`, vous pouvez y accéder depuis l'extérieur, du moins dans l'implémentation actuelle. Cela peut toutefois changer, puisque le projet actuel est de cloner la table de symboles dans un nouvel interpréteur. Une solution pouvant être légèrement plus compatible consiste à positionner les choses avant de créer l'objet `Safe`, comme montré ci-dessous. Ceci est susceptible de continuer à fonctionner et c'est une manière habile de configurer un objet `Safe` qui doit démarrer avec beaucoup « d'états ». (Admettons-le, `$Donjon::truc` n'a pas beaucoup d'états.)

```
use Safe;
$Donjon::truc = 1;    # L'accès direct est toujours déconseillé.
my $salle = Safe->new("Donjon");
```

Mais `Safe` fournit également un moyen d'accès aux événements globaux du compartiment si vous ne connaissez pas le nom de paquetage du compartiment. Ainsi, pour une compatibilité ascendante maximale (toutefois, la vitesse ne le sera pas, maximale), nous vous suggérons d'employez la méthode `reval` :

```
use Safe;
my $salle = Safe->new();
$salle->reval('$truc = 1');
```

(En fait, il s'agit de la même méthode que vous utiliserez pour exécuter du code suspect.) Lorsque vous passez du code dans le compartiment pour le compiler et l'exécuter, ce code pense qu'il se trouve vraiment dans le paquetage `main`. Ce que le monde extérieur appelle `$Donjon::truc`, le code à l'intérieur pense qu'il s'agit de `$main::truc` ou `$::truc` ou simplement `$truc` si `use strict` n'est pas actif. Cela ne marchera pas si l'on écrit `$Donjon::truc` à l'intérieur du compartiment car ceci accéderait en fait à `$Donjon::Donjon::truc`. En donnant à l'objet `Safe` sa propre notion de `main`, les variables et les sous-programmes dans le reste de votre programme sont protégés.

Pour compiler et exécuter du code dans le compartiment, utilisez la méthode `reval` (« `eval` restreint ») en passant une chaîne de code comme argument. Comme avec toute autre construction `eval` *`CHAINE`*, les erreurs à la compilation et les exceptions à l'exécution pour `reval` ne tueront pas votre programme. Elles ne feront qu'abandonner le `reval` et laisser l'exception dans `$@`, assurez-vous donc de la vérifier après chaque appel à `reval`.

En utilisant les initialisations données plus tôt, ce code affichera que « `truc vaut maintenant 2` » :

```
$salle->reval('$truc++; print "truc vaut maintenant $main::truc\n"');
if ($@) {
    die "Impossible de compiler le code dans le compartiment : $@";
}
```

Si vous ne voulez que compiler du code sans l'exécuter, enrober votre chaîne dans une déclaration de sous-programme :

```
$salle->reval(q{
    our $truc;
```

```
    sub affiche_truc {
        print "truc vaut maintenant $main::truc\n";
    }
    }, 1);
die if $@;       # vérification de la compilation
```

Cette fois, nous avons passé à `reval` un second argument qui, s'il est vrai, indique à `reval` de compiler le code avec le pragma `strict` actif. Depuis l'intérieur de la chaîne de code, vous ne pouvez pas désactiver `strict` car importer et ne pas importer sont justement deux des choses que vous ne pouvez pas faire normalement dans un compartiment `Safe`. Il existe encore beaucoup de choses que vous ne pouvez pas faire normalement dans un compartiment `Safe` — voir le paragraphe suivant.

Une fois que vous avez créé la fonction `affiche_truc` dans le compartiment, ces instructions sont à peu près équivalentes :

```
$salle->reval('affiche_truc()');     # Meilleure façon.
die if $@;

$salle->varglob('affiche_truc')->(); # Appel via un glob anonyme.

Donjon::affiche_truc();              # Appel direct, fortement déconseillé.
```

Restriction de l'accès aux opérateurs

L'autre chose importante à propos d'un objet `Safe` est que Perl limite les opérations disponibles à l'intérieur du bac à sable. (Vous pourriez très bien laisser votre enfant prendre un seau et une pelle dans le bac à sable mais vous refuseriez certainement de le laisser jouer avec un bazooka.) Il ne suffit pas de protéger seulement le reste de votre programme ; vous devez aussi protéger le reste de votre ordinateur.

Lorsque vous compilez du code dans un objet `Safe`, avec `reval` ou `rdo` (la version restreinte de l'opérateur `do` *`FICHIER`*), le compilateur consulte une liste spéciale, par compartiment, de contrôle d'accès pour décider si chaque opération individuelle est considérée ou non comme sûre pour la compilation. De cette manière vous n'avez pas à (trop) vous inquiéter à propos des séquences d'échappement imprévues du shell, de l'ouverture de fichiers alors que vous ne le vouliez pas, des assertions de code étranges dans les expressions régulières ou de la plupart des problèmes d'accès depuis l'extérieur que la plupart des gens craignent habituellement. (Ou qu'ils devraient craindre.)

L'interface spécifiant quels opérateurs devraient être permis ou restreints est actuellement en cours de réécriture, nous ne montrerons donc ici que la manière d'en utiliser un ensemble par défaut. Pour plus de détails, consultez la documentation en ligne du module `Safe`.

Le module `Safe` n'offre pas une protection complète contre les attaques par saturation des serveurs (*denial-of-service attacks*), particulièrement lorsqu'on l'utilise dans ses modes les plus permissifs. Les attaques par saturation des serveurs prennent toutes les ressources d'un certain type disponibles sur le système, empêchant les autres processus d'accéder aux caractéristiques essentielles du système. Des exemples de telles attaques comprennent le bourrage de la table de processus du noyau, la main-mise sur la CPU en exécutant dans une frénétique boucle sans fin, l'épuisement de toute la mémoire disponible et le remplissage d'un système de fichiers. Ces problèmes sont très difficiles à

résoudre, particulièrement de manière portable. Voir la fin de la section *Du code déguisé en données* pour une discussion plus approfondie sur les attaques par saturation des serveurs.

Exemples sécurisés

Imaginez que vous ayez un programme CGI gérant un formulaire dans lequel l'utilisateur peut rentrer une expression Perl arbitraire et récupérer le résultat évalué.[14] Comme toutes les entrées venant de l'extérieur, la chaîne arrive marquée, Perl ne vous laissera donc pas encore lui appliquer un `eval` — vous devrez d'abord la nettoyer grâce à une reconnaissance de motif. Le problème est que vous ne serez jamais capables de concevoir un motif qui puisse détecter toutes les menaces possibles. Et vous n'oserez pas nettoyer ce que vous avez reçu et l'envoyer à l'`eval` interne. (Si vous osez, *nous* serons tentés de pénétrer dans votre système pour supprimer le script.)

C'est ici que `reval` entre en scène. Voici un script CGI qui traite un formulaire avec un simple champ, évalue (dans un contexte scalaire) la chaîne qu'il trouve et affiche le résultat formaté :

```
#!/usr/bin/perl -lTw
use strict;
use CGI::Carp 'fatalsToBrowser';
use CGI qw/:standard escapeHTML/;
use Safe;

print header(-type => "text/html;charset=UTF-8"),
    start_html("Résultat de l'expression Perl");
my $expr = param("EXPR") =~ /^([^;]+)/
            ? $1 # renvoie la portion maintenant nettoyée
            : croak("aucun champ EXPR valide dans le formulaire");
my $reponse = Safe->new->reval($expr);
die if $@;

print p("Le résultat de", tt(escapeHTML($expr)),
                  "est", tt(escapeHTML($reponse)));
```

Imaginez qu'un utilisateur malveillant vous passe « `print 'cat /etc/passwd'` » (ou pire) comme chaîne d'entrée. Grâce à l'environnement restreint qui interdit les apostrophes inverses, Perl décèlera le problème à la compilation et quittera immédiatement. La chaîne dans `$@` est « `quoted execution (``, qx) trapped by operation mask` » (exécution entre apostrophes inverses (``, `qx`) capturée par le masque des opérations), ainsi que les informations obligatoires à la fin, identifiant où le problème a eu lieu.

Puisque nous n'avons pas dit le contraire, les compartiments que nous avons créés utilisent tous l'ensemble par défaut d'opérations permises. La manière dont vous vous y prenez pour déclarer des opération spécifiques comme permises ou interdites, importe peu ici. Et puisque vous pouvez créer plusieurs objets `Safe` dans votre programme, vous pouvez conférer différents degrés de confiance aux différents morceaux de code selon leur provenance.

14. Merci de ne pas rire. Nous avons vraiment vu des pages web faire cela. Sans `Safe` !

Si vous désirez vous amuser avec Safe, voici une petite calculatrice interactive en Perl. Vous pouvez lui passer des expressions numériques et voir immédiatement leur résultats. Mais elle n'est pas limitée aux nombres. Elle ressemble plus à l'exemple de boucle pour eval au chapitre 29, où vous pouvez prendre tout ce qu'on vous donne, l'évaluer et renvoyer le résultat. La différence est que la version avec Safe n'exécute pas aveuglément tout ce que vous souhaitez. Vous pouvez lancer cette calculatrice interactivement sur votre terminal, y taper quelques bouts de code en Perl et vérifier les réponses pour avoir un aperçu des types de protection fournis par Safe.

```
#!/usr/bin/perl -w
# calcsécu - programme de démonstration pour jouer avec Safe
use strict;
use Safe;
my $salle = Safe->new();
while (1) {
    print "Entrée : ";
    my $expr = <STDIN>;
    exit unless defined $expr;
    chomp($expr);
    print "$expr donne ";
    local $SIG{__WARN__} = sub { die @_ };
    my $result = $salle->reval($expr, 1);
    if ($@ =~ s/at \(eval \d+\).*//) {
        printf "[%s]: %s", $@ =~ /trapped by operation mask/
            ? "Violation de la sécurité" : "Exception", $@;
    } else {
        print "[Résultat normal] $result\n";
    }
}
```

Avertissement : le module Safe est actuellement en cours de re-spécification pour exécuter chaque compartiment dans un interpréteur Perl complètement indépendant dans le même processus. (C'est la stratégie employée par le mod_perl d'Apache lorsqu'il exécute des scripts Perl précompilés.) Les détails sont encore vagues en ce moment mais notre boule de cristal suggère que désigner des choses à l'intérieur du compartiment en utilisant un paquetage nommé ne vous conduira pas bien loin après la réécriture imminente. Si vous utilisez une version de Perl plus récente que 5.6, vérifier les notes concernant la version dans *perldata*(1) pour voir ce qui a changé ou consultez la documentation du module Safe lui-même. (Bien entendu, c'est toujours ce que vous faites, n'est-ce pas ?)

Du code déguisé en données

Les compartiments Safe sont disponibles lorsque des choses vraiment effrayantes arrivent mais cela ne veux pas dire que vous devriez laisser tomber votre garde lorsque vous vous occupez chez vous de choses quotidiennes. Vous devez cultiver une vigilance sur ce qui vous entoure et regarder les choses du point de vue de quelqu'un voulant s'introduire. Vous devez prendre des mesures proactives comme garder les choses bien éclairées et tailler les buissons qui peuvent dissimuler divers problèmes se tapissant dans l'ombre.

Perl essaie de vous aider dans ce domaine également. L'analyse conventionnelle et le schéma d'exécution de Perl évitent les pièges dont les langages de programmation shell sont souvent la proie. Il existe beaucoup de fonctionnalités extrêmement puissantes dans les langages, mais par de par leur conception, ils sont syntaxiquement et sémantiquement limités de façon que le programmeur garde le contrôle. À quelques exceptions près, Perl n'évalue chaque token qu'une seule fois. Quelque chose semblant être utilisé comme une variable pour une simple donnée ne s'enracinera pas soudainement quelque part dans votre système de fichiers.

Malheureusement, ce genre de chose peut arriver si vous appelez le shell pour lancer d'autres programmes pour vous car vous tournez alors avec les règles du shell à la place de celles de Perl. Le shell est cependant facile à éviter — vous n'avez qu'à utiliser les formes avec un argument de liste pour les fonctions `system`, `exec` ou `open` sur un pipe. Bien que les apostrophes inverses ne disposent pas d'une forme avec un argument de liste qui soit sécurisée contre le shell, vous pouvez toujours les émuler comme décrit à la section *Accès aux commandes et aux fichiers avec des privilèges restreints*. (Alors qu'il n'y a pas de moyen syntaxique pour donner aux apostrophes inverses un argument de liste, une forme à plusieurs arguments de l'opérateur `readpipe` sous-jacent est en cours de développement ; mais au moment où nous écrivons ces lignes, elle n'est pas encore prête pour le devant de la scène.)

Lorsque vous utilisez une variable dans une expression (y compris lorsque vous l'interpolez dans une chaîne entre guillemets), il n'y a *aucune chance* pour que la variable contienne du code Perl qui fasse quelque chose auquel vous ne vous attendiez pas.[15] Au contraire du shell, Perl n'a jamais besoin de guillemets ou d'apostrophes protecteurs autour des variables, peu importe ce qu'elle peuvent contenir.

```
$nouveau = $ancien;           # pas besoin de guillemets ou d'apostrophes.
print "$nouveau items\n";     # $nouveau ne peut pas vous faire de mal.

$phrase = "$nouveau items\n"; # ici non plus.
print $phrase;                # toujours parfaitement ok.
```

Perl adopte une approche « WYSIWYG » (*what you see is what you get*, ce que vous voyez est ce que vous obtenez). Si vous ne voyez pas de niveau supplémentaire d'interpolation, alors il n'y en a pas. Il *est* possible d'interpoler arbitrairement des expressions Perl dans des chaînes mais seulement si vous demandez spécifiquement à Perl de le faire. (Et même ainsi, le contenu est toujours sujet aux vérifications de marquage si vous êtes en mode de marquage.)

```
$phrase = "Vous avez perdu @{[ 1 + int rand(6) ]} points\n";
```

L'interpolation n'est cependant pas récursive. Vous ne pouvez pas cacher une expression arbitraire dans une chaîne :

```
$compteur = '1 + int rand(6)';     # Du code aléatoire.
$dire = "$compteur points";        # Plutôt un littéral.
$dire = "@{[$compteur]} points;    # Également un littéral.
```

15. Cela dit, si vous générez une page web, il est possible d'émettre des balises HTML, comprenant du code JavaScript, qui pourrait faire quelque chose auquel le navigateur distant ne s'attendait pas.

Les deux affectations à `$dire` produiraient « `1 + rand(6) points` », sans évaluer le contenu interpolé de `$compteur` en tant que code. Pour obliger Perl à faire cela, vous devez appeler `eval` *`CHAINE`* explicitement :

```
$code = '1 + int rand(6)';
$lance_de_des = eval $code;
die if $@;
```

Si `$code` était marqué, cet `eval` *`CHAINE`* aurait levé sa propre exception. Bien entendu, il ne faut presque jamais évaluer du code aléatoire provenant de l'utilisateur — mais si vous le faites, vous devriez jetez un œil au module `Safe`. Vous avez dû en entendendre parler.

Il existe un endroit où Perl peut parfois traiter des données comme du code ; à savoir, lorsque le motif dans un opérateur `qr//`, `m//` ou `s///` contient l'une des nouvelles assertions des expressions régulières, `(?{` *`CODE`* `})` ou `(??{` *`CODE`* `})`. Celles-ci ne posent pas de problèmes de sécurité lorsqu'elles sont utilisées comme des littéraux dans des correspondances de motifs :

```
$cmpt = $n = 0;
while ($donne =~ /( \d+ (?{ $n++ }) | \w+ )/gx) {
    $cmpt++;
}
print "$cmpt mots trouvés, dont $n nombres.\n";
```

Mais du code existant interpolant des variables dans des correspondances a été écrit en présupposant que les données étaient des données, et non du code. Les nouvelles constructions pourraient avoir introduit un trou de sécurité dans des programmes auparavant sécurisés. Ainsi, Perl refuse d'évaluer un motif si une chaîne interpolée contient une assertion de code et lève à la place une exception. Si vous avez vraiment besoin de cette fonctionnalité, vous pouvez toujours l'activer avec le pragma de portée lexicale `use re 'eval'`. (Vous ne pouvez cependant toujours pas utiliser de données marquées en tant qu'assertions de code interpolé.)

Un type complètement différent de soucis de sécurité se produisant avec les expressions régulières réside dans les problèmes de saturation des serveurs (*denial-of-service*). Ceux-ci peuvent faire sortir votre programme trop tôt, ou tourner trop longtemps, ou encore épuiser toute la mémoire disponible — et même parfois faire un *core dump*, selon l'âge du capitaine.

Lorsque vous traitez des motifs fournis par l'utilisateur, vous n'avez pas à vous inquiéter à propos de l'interprétation de code Perl quelconque. Cependant, le moteur d'expressions régulières possède ses propres petits compilateur et interpréteur et le motif fourni par l'utilisateur est capable d'infliger un infarctus au compilateur d'expressions régulières. Si un motif interpolé n'est pas valide, une exception est levée à l'exécution, ce qui est fatal à moins de l'intercepter. Si vous essayez effectivement de l'intercepter, assurez-vous d'utiliser seulement `eval` *`BLOC`*, et non `eval` *`CHAINE`*, car le niveau d'évaluation supplémentaire de cette dernière forme permettrait en fait l'exécution de code Perl aléatoire. À la place, faites quelque chose comme ceci :

```
if (not eval { "" =~ /$corresp/; 1 }) {
    # (Maintenant faites ce que vous voulez avec un mauvais motif.)
} else {
```

```
    # Nous savons que le motif est au moins sécurisé pour la compilation.
    if ($donnee =~ /$corresp/) {...}
}
```

Un problème plus troublant de saturation des serveurs est qu'étant donné la bonne donnée et le bon motif de recherche, votre programme peut sembler se bloquer à jamais. Car certaines correspondances de motifs exigent un temps de calcul exponentiel et cela peut facilement excéder le taux de MTBF (*Mean Time Between Failures*, *N.d.T.* : temps moyen de fonctionnement entre deux incidents) de notre système solaire. Si vous êtes particulièrement chanceux, ces motifs demandant des calcul intensifs exigeront également un espace de stockage exponentiel. Si c'est le cas, votre programme épuisera toute la mémoire virtuelle disponible, écroulera votre système, embêtera vos utilisateurs et soit décédera par `die` avec une erreur ordinaire « `Out of memory!` » (dépassement de mémoire), soit laissera derrière lui un fichier *core dump* vraiment énorme, peut-être cependant pas aussi gros que le système solaire.

Comme la plupart des attaques par saturation des serveurs, celle-ci n'est pas facile à résoudre. Si votre plate-forme implémente la fonction *alarm*, vous pouvez donner un délai limite pour la correspondance de motifs. Malheureusement, Perl ne peut pas (actuellement) garantir que le simple acte de gérer un signal ne va jamais lever un *core dump*. (Nous avons prévu de le résoudre dans une prochaine version.) Vous pouvez cependant toujours l'essayer et même si le signal n'est pas géré élégamment, au moins le programme ne bouclera pas à jamais.

Si votre système implémente des limites de ressources par processus, vous pourriez configurer celles-ci dans votre shell avant d'appeler le programme Perl ou utiliser le module `BSD::Ressource` de CPAN pour le faire directement en Perl. Le serveur web Apache vous permet de fixer des limites de temps, de mémoire et de taille de fichiers sur les scripts CGI qu'il lance.

Finalement, nous espérons que nous vous avons laissé un sentiment non résolu d'insécurité. Souvenez-vous que si vous êtes paranoïaque, cela ne veut pas dire pour autant qu'il n'y a personne à vos trousses. Alors vous feriez mieux de vous en accommoder.

24

Techniques couramment employées

Adressez-vous à n'importe quel programmeur Perl et il sera heureux de vous donner des pages et des pages de conseil sur la manière de programmer. Nous ne sommes pas différents (au cas où vous ne l'ayez pas remarqué). Dans ce chapitre, plutôt que d'essayer de vous exposer les caractéristiques spécifiques de Perl, nous prendrons la direction opposée en distillant quelques descriptions des techniques idiomatiques en Perl. Nous espérons qu'en rassemblant divers petits morceaux qui semblent sans rapport les uns avec les autres, vous parviendrez à vous imbiber du sentiment de ce qu'est vraiment « penser en Perl ». Après tout, lorsque vous programmez, vous n'écrivez pas un paquet d'expressions, puis un paquet de sous-programmes, puis un paquet d'objets. Vous devez tout appréhender d'un seul coup, plus ou moins. Voilà donc à quoi ressemble ce chapitre.

Il y a tout de même une organisation rudimentaire dans ce chapitre, en ce que nous commencerons par les conseils négatifs et tracerons le chemin jusqu'au conseils positifs. Nous ne savons pas si vous vous sentirez mieux grâce à cela mais cela nous aidera, nous, à nous sentir mieux.

Étourderies courantes des débutants

La plus grande étourderie est d'oublier d'utiliser `use warnings`, qui relève de nombreuses erreurs. La deuxième est de ne pas utiliser `use strict` quand il le faut. Ces deux pragmas peuvent vous économiser des heures de casse-tête lorsque votre programme commence à grossir. (Et ce sera le cas.) Un autre faux pas consiste à oublier de consulter la FAQ en ligne. Imaginez que vous vouliez trouver si Perl dispose d'une fonction `round`. Vous devriez essayer de chercher d'abord dans la FAQ :

```
% perlfaq round
```

À part ces « méta-étourderies », il y a plusieurs types de pièges de programmation. Certains pièges dans lesquels presque tout le monde tombe et d'autres dans lesquels vous ne tombez que si vous êtes originaire d'une culture particulière possédant des pratiques différentes. Nous les avons répartis dans les sections suivantes.

Erreurs universelles

- Mettre une virgule après le handle de fichier dans une instruction `print`. Bien qu'il paraisse tout à fait rationnel et élégant d'écrire :

  ```
  print STDOUT, "adieu, monde ", $adj, " !\n";    # MAUVAIS
  ```

  c'est néanmoins incorrect, à cause de la première virgule. Ce qu'il faut à la place est la syntaxe d'objet indirect :

  ```
  print STDOUT "adieu, monde ", $adj, " !\n";     # ok
  ```

  La syntaxe est ainsi faite que l'on puisse écrire :

  ```
  print $handle_fichier "adieu, monde ", $adj, " !\n";
  ```

  où `$handle_fichier` est un scalaire contenant le nom d'un handle de fichier à l'exécution. Ce qui est différent de :

  ```
  print $pas_un_handle_fichier, "adieu, monde ", $adj, " !\n";
  ```

  où `$pas_un_handle_fichier` n'est qu'une chaîne qui est ajoutée à la liste de ce qui doit être imprimé. Voir « objet indirect » dans le glossaire.

- Utiliser `==` au lieu de `eq` et `!=` au lieu de `ne`. Les opérateurs `==` et `!=` sont des tests *numériques*. Les deux autres sont des tests sur les *chaînes*. Les chaînes `"123"` et `"123.00"` sont égales en tant que nombres, mais pas en tant que chaînes. De même, les chaînes non-numériques sont numériquement égales à zéro. À moins que vous ne traitiez des nombres, vous utilisez presque toujours à la place les opérateurs de comparaison de chaînes.

- Oublier le point-virgule final. Chaque instruction Perl est terminée par un point-virgule ou la fin d'un bloc. Les sauts de ligne ne sont pas des terminateurs d'instruction comme en *awk*, en Python ou en FORTRAN. Souvenez-vous que Perl est comme C.

  Une instruction contenant un document « ici-même » (*here document*) est tout particulièrement apte à perdre son point-virgule. Cela devrait ressembler à ceci :

  ```
  print <<'FINIS';
  Sans une grande roideur et une continuelle
  attention à toutes ses paroles, on est exposé à
  dire en moins d'une heure le oui et le non sur une
  même chose ou sur une même personne, déterminé
  seulement par un esprit de société et de commerce
  qui entraîne naturellement à ne pas contredire
  celui-ci et celui-là qui en parlen  différemment.
                                    -- Jean de La Bruyère
  FINIS
  ```

- Oublier qu'un *`BLOC`* exige des accolades. Les instructions seules ne forment pas de *`BLOC`* par elles-mêmes. Si vous créez une structure de contrôle comme un `while` ou un `if` qui demande un ou plusieurs *`BLOC`* s, vous *devez* utiliser des accolades autour de chaque *`BLOC`*. Souvenez-vous que Perl n'est comme C.

- Ne pas sauver `$1`, `$2`, etc., dans les expressions régulières. Souvenez-vous que chaque nouvelle correspondance (`m//`) ou substitution (`s///`) modifie (ou remet à zéro ou écrase) vos variables `$1`, `$2`. . . , ainsi que `$‘`, `$&`, `$’` et consorts. Une manière de les sauvegarder consiste à évaluer la correspondance dans un contexte de liste,

comme dans :

```
my ($un, $deux) = /(\w+) (\w+)/;
```

- Ne pas se rendre compte qu'un `local` change aussi la valeur de la variable telle que la voient les autres sous-programmes appelés depuis l'intérieur de la portée du `local`. Il est facile d'oublier que `local` est une instruction d'exécution qui travaille dans une portée dynamique, car il n'en existe aucun équivalent dans des langages comme C. Voir la section *Déclarations avec portée* au chapitre 4, *Instructions et déclarations*. De toute façon, il est en général préférable d'utiliser un `my`.
- Perdre la trace des appariements d'accolades. Un bon éditeur vous aidera à trouver les paires correspondantes. Prenez-en un. (Ou deux.)
- Employer des instructions de contrôle de boucle dans un `do {} while`. Bien que les accolades dans cette structure de contrôle ont l'air de faire partie d'un *`BLOC`* de boucle, ce n'est en fait pas le cas.
- Utiliser `$truc[1]` alors que vous voulez dire `$truc[0]`. Les tableaux commence par défaut à zéro en Perl.
- Écrire `@truc[0]` alors que vous vouliez dire `$truc[0]`. La référence à `@truc[0]` est une *tranche* de tableau, représentant un tableau constitué du seul élément `$truc[0]`. Parfois, cela ne fait aucune différence, comme dans :

  ```
  print "la réponse est @truc[0]\n";
  ```

  Mais la différence est énorme pour des choses comme :

  ```
  @truc[0] = <STDIN>;
  ```

  qui va absorber tout `STDIN` en affectant la *première* ligne à `$truc[0]` et en rejetant tout le reste, ce qui n'est probablement pas ce que vous aviez prévu. Prenez l'habitude de considérer que `$` représente une valeur unique, alors que `@` représente une liste de valeurs, et tout ira bien.
- Oublier les parenthèses avec un opérateur de liste comme `my` :

  ```
  my $x, $y = (4, 8); # FAUX my ($x, $y) = (4, 8); # ok
  ```

- Oublier de sélectionner le bon handle de fichier avant de positionner `$^`, `$~` ou `$|`. Ces variables dépendent du handle de fichier actuellement sélectionné, comme le détermine `select (`*`HANDLE_FICHIER`*`)`. Le handle de fichier initialement sélectionné est `STDOUT`. Vous devriez vraiment plutôt utiliser les méthodes de handles de fichiers du module `FileHandle`. Voir le chapitre 28, *Noms spéciaux*.

Conseils fréquemment ignorés

Les programmeurs Perl Pratiquants devraient prendre note de ce qui suit :

- Souvenez-vous que de nombreuses opérations se comportent différemment dans un contexte de liste et dans un contexte scalaire. Par exemple :

  ```
  ($x) = (4, 5, 6);      # Contexte de liste ; $x vaut 4
   $x = (4, 5, 6);       # Co     e scalaire ; $x vaut 6

   @a = (4, 5, 6);
   $x = @a;              # Contexte scalaire ; $x vaut 3
                         # (la longueur du tableau)
  ```

- Évitez si possible les mots simples, surtout s'ils sont entièrement en minuscules. Il est impossible de savoir du premier coup d'œil si un mot est une fonction ou une chaîne brute. Vous supprimerez toute confusion en protégeant les chaînes par des guillemets et des parenthèses autour des arguments d'appels de fonction. En fait, le pragma `use strict` au début de votre programme donne une erreur de compilation si l'on utilise un mot simple — ce qui est probablement une bonne chose.
- On ne peut pas distinguer au premier coup d'œil les fonctions internes qui sont des opérateurs unaires (comme `chop` et `chdir`), celles qui sont des opérateurs de liste (comme `print` et `unlink`) et celles qui n'ont pas d'argument (comme `time`). Il vous faut les apprendre au chapitre 29. Comme toujours, utilisez des parenthèses si vous n'êtes pas sûrs — et même si vous n'êtes pas sûrs d'être sûrs. Remarquez également que les sous-programmes définis par l'utilisateur sont par défaut des opérateurs de liste mais peuvent être déclarés comme opérateurs unaires avec un prototype valant `($)` ou sans arguments avec un prototype valant `()`.
- On a souvent du mal à se souvenir que certaines fonctions emploient par défaut `$_`, ou `@ARGV` ou autre chose, alors que d'autres non. Prenez le temps d'apprendre lesquelles, ou évitez les arguments par défaut.
- `<HF>` n'est pas le nom d'un handle de fichier mais un opérateur d'angle qui effectue une opération d'entrée par ligne sur le handle. Cette confusion se manifeste généralement quand des gens essayent un `print` sur l'opérateur angle :

```
print <HF> "salut"; # MAUVAIS, omettre les angles
```

- Souvenez-vous également que les données lues par l'opérateur d'angle ne sont affectées à `$_` que lorsque le fichier lu est la seule condition d'une boucle `while` :

```
while (<HF>) { }    # Données affectées à $_.
<HF>;               # données lues et éliminées !
```

- N'utilisez pas = lorsque vous avez besoin de =~ ; les deux constructions sont très différentes :

```
$x =  /truc/;       # recherche "truc" dans $_, met le résultat dans $x
$x =~ /truc/;       # recherche "truc" dans $x, élimine le résultat
```

- Utilisez `my` pour les variables locales chaque fois que vous pouvez vous en accommoder. `local` donne en fait une valeur temporaire à une variable globale, ce qui vous expose grandement aux effets secondaires dus à la portée dynamique
- N'utilisez pas `local` sur les variables exportées par un module. La localisation d'une variable exportée ne fait pas changer sa valeur. Le nom local devient l'alias d'une nouvelle valeur mais le nom externe reste l'alias de l'original.

Pièges du C

Les programmeurs Cérébraux du C devraient prendre note de ce qui suit :

- Les accolades sont obligatoires pour les blocs `if` et `while`.
- Vous devez utiliser `elsif` plutôt que « `else if` » ou de « `elif` ». Une syntaxe comme :

```
if (expression) {
    bloc;
```

```
}
else if (autre_expression) { # MAUVAIS
    autre_bloc;
}
```

est illégale. La partie `else` est toujours un bloc et un `if` tout nu n'est pas un bloc. Ne vous attendez pas à ce que Perl se comporte exactement comme le C. À la place, il faut écrire :

```
if (expression) {
    bloc;
}
elsif (autre_expression) {
    autre_bloc;
}
```

Remarquez également que « `elif` » est « `file` » écrit à l'envers. Seuls des fanatiques d'Algol pourraient tolérer un mot-clef identique à un autre mot écrit à l'envers.

- Les mots-clefs `break` et `continue` de C deviennent respectivement `last` et `next` en Perl. En revanche, ces derniers ne fonctionnent *pas* dans une construction `do {} while`.
- Il n'existe pas d'instruction `switch`. (Mais il est facile d'en élaborer une au vol ; voir les paragraphes *Blocs simples* et *Structures de cas* au chapitre 4.
- Les variables commencent par `$`, `@` ou `%` en Perl.
- Les commentaires commencent par `#` et non par `/*`.
- Vous ne pouvez pas obtenir l'adresse de n'importe quoi, bien qu'un opérateur similaire en Perl soit l'antislash, qui crée une référence.
- `ARGV` doit être en majuscules. `$ARGV[0]` est le `argv[1]` de C et le `argv[0]` de C devient `$0`.
- Les appels système comme `link`, `unlink` et `rename` renvoient vrai en cas de succès et non `0`.
- Les gestionnaires de signaux de `%SIG` traitent des noms de signaux et non des nombres.

Pièges du shell

Les programmeurs Shérifs du Shell devraient prendre note de ce qui suit :

- Les variables sont préfixées par `$` ou `@` tant du côté gauche d'une affectation que du côté droit. Une affectation « à la shell » comme :

  ```
  chameau='dromadaire';    # MAUVAIS
  ```

  ne sera pas analysée comme prévu. Vous devez mettre :

  ```
  $chameau='dromadaire';   # ok
  ```

- La variable de boucle pour un `foreach` exige également un `$`. Même si *csh* préfère :

  ```
  foreach bosse (une deux)
      remplir $bosse
  end
  ```

en Perl, ceci s'écrit plutôt :

```
foreach $bosse ("une", "deux") {
    remplir($bosse);
}
```

- L'opérateur apostrophe inverse interpole les variables sans se préoccuper de la présence d'apostrophes dans la commande.
- L'opérateur apostrophe inverse ne convertit pas la valeur de retour. En Perl, il faut explicitement éliminer le saut de ligne, comme ceci :

  ```
  chop($mon_hote = `hostname`);
  ```

- Les shells (surtout *csh*) offrent plusieurs niveaux de substitution sur la ligne de commande. Perl n'effectue de substitution que dans certaines constructions comme les guillemets, les apostrophes inverses, les signes inférieurs et supérieurs et les motifs de correspondance.
- Les shells ont tendance à interpréter les scripts petit à petit. Perl compile le programme entier avant de l'exécuter (hormis les blocs `BEGIN`, qui s'exécutent avant que la compilation soit achevée).
- Les arguments sont accessibles *via* `@ARGV` et non `$1`, `$2`, etc.
- L'environnement n'est pas automatiquement accessible sous forme de variables scalaires séparées. Utilisez le module standard `Env` si vous voulez que ce soit le cas.

Pièges des anciens Perl

Les programmeurs Pénitents de Perl 4 (et des Précédents) devraient prendre note des modifications suivantes entre la version 4 et la version 5 qui pourraient affecter les anciens scripts :

- `@` interpole maintenant systématiquement un tableau dans des chaînes protégées par guillemets. Certains programmes doivent maintenant utiliser l'antislash pour protéger les `@` qui ne devraient pas être interpolées.
- Les mots simples qui ressemblaient à des chaînes à Perl ont maintenant l'air d'appels à un sous-programme si un sous-programme de ce nom est défini avant que le compilateur ne les voie. Par exemple :

  ```
  sub Ciao { die "Hasta la vista, baby !" }
  $SIG{'QUIT'} = Ciao;
  ```

  Dans les précédentes versions de Perl, ce code mettait en place le gestionnaire de signal. Il appelle maintenant la fonction ! Vous pouvez utiliser l'option **`-w`** pour détecter cette construction à risque ou `use strict` pour la rendre illégale.
- Les identificateurs commençant par « _ » ne se trouvent plus forcément dans le paquetage `main`, sauf le symbole souligné brut lui-même (comme dans `$_`, `@_`, etc).
- Le double deux-points est maintenant un séparateur de paquetage valide dans un identificateur. Ainsi, l'instruction :

  ```
  print "$a::$b::$c\n"
  ```

  analyse maintenant `$a::` comme la référence à la variable, alors que dans les versions précédentes, seul le `$a` était considéré comme la référence à la variable. De

même :

```
print "$var::abc::xyz\n";
```

est maintenant interprété comme une variable unique `$var::abc::xyz`, alors que dans les versions précédentes, la variable `$var` aurait été suivie par le texte constant `::abc::xyz`.

- `s'$motif'remplacement'` n'interpole plus `$motif`. (Le `$` serait interprété comme une assertion de fin de ligne.) Ce comportement n'arrive que dans l'utilisation d'apostrophes en tant que séparateur de substitution ; dans les autres substitutions, `$motif` est toujours interpolé.
- Le deuxième et le troisième argument de `splice` sont maintenant évalués dans un contexte scalaire plutôt que dans contexte de liste.
- Des erreurs sémantiques apparaissent maintenant en raison de la précédence :

```
shift @liste + 20;      # maintenant analysé comme shift(@liste + 20),
                        # illégal !
$n = keys %map + 20;    # maintenant analysé comme keys(%map + 20),
                        # illégal !
```

Parce que si ces derniers fonctionnaient, ceci ne le pourrait pas :

```
sleep $sieste + 20;
```

- La précédence des opérateurs d'affectation est maintenant la même que celle de l'affectation. Les versions antérieures de Perl leur donnaient par erreur la précédence de l'opérateur associé. Il faut donc maintenant les encadrer entre parenthèses dans des expressions comme :

```
/truc/ ? ($a += 2) : ($a -= 2);
```

Sinon

```
/truc/ ? $a += 2 : $a -= 2;
```

serait analysé par erreur en :

```
(/truc/ ? $a += 2 : $a) -= 2;
```

En revanche,

```
$a += /truc/ ? 1 : 2;
```

fonctionne maintenant comme s'y attendrait un programmeur C.

- `open TRUC || die` est maintenant incorrect. Il faut des parenthèses autour du handle de fichier, car `open` a la précédence d'un opérateur de liste.
- Les éléments des listes d'arguments pour les formats sont maintenant évalués dans un contexte de liste. Cela signifie que vous pouvez maintenant interpoler des valeurs de liste.
- Vous ne pouvez pas aller par un `goto` dans un bloc que l'optimisation a supprimé. Damnation.
- Il n'est plus légal d'employer un espace comme nom de variable ou comme délimiteur de construction protégée. Enfer et damnation.
- La fonction `caller` renvoie maintenant une valeur fausse dans un contexte scalaire s'il n'y a pas d'appelant. Cela permet aux modules de déterminer s'ils sont requis ou lancés directement.

- `m//g` rattache maintenant son état à la chaîne recherchée plutôt qu'à l'expression régulière. Voir le chapitre 5, *Correspondance de motifs* pour plus de détails.
- `reverse` n'est plus autorisé en tant que nom de routine pour de tri pour `sort`.
- *taintperl* n'est plus un exécutable séparé. Il existe maintenant une option **`-T`** qui active le marquage quand il ne l'est pas automatiquement.
- Les chaînes entre guillemets ne peuvent plus se terminer par un `$` ou un `@` non protégé.
- La syntaxe archaïque `if` *`BLOC BLOC`* n'est plus supportée.
- Les indices de tableau négatifs comptent maintenant depuis la fin du tableau.
- Il est maintenant garanti que l'opérateur virgule, dans un contexte scalaire, procure un contexte scalaire à ses arguments.
- L'opérateur ** lie maintenant plus étroitement que le moins unaire.
- Diminuer `$#tableau` élimine maintenant immédiatement des éléments du tableau.
- Il n'est pas garanti que `delete` renvoie la valeur détruite des tableaux liés par `tie` car l'implémentation de cette fonctionnalité serait onéreuse pour certains modules.
- La construction `"ceci est $$x"`, qui interpolait l'ID du processus à cet endroit, essaie maintenant de déréférencer `$x`. Toutefois, `$$` tout seul, fonctionne toujours très bien.
- Le comportement de `foreach` a légèrement changé lorsqu'il parcours itérativement dans une liste qui n'est pas un tableau. Il affectait la liste à un tableau temporaire mais ne le fait plus pour des raisons d'efficacité. Cela signifie que l'on parcourt maintenant les valeurs actuelles et non les copies des valeurs. Les modifications de la variable de boucle peuvent changer les valeurs originelles, même après un `grep` ! Par exemple

  ```
  % perl4 -e '@a = (1,2,3); for (grep(/./, @a)) { $_++ }; print "@a\n"'
  1 2 3
  % perl5 -e '@a = (1,2,3); for (grep(/./, @a)) { $_++ }; print "@a\n"'
  2 3 4
  ```

  Pour conserver l'ancien comportement de Perl, vous devrez affecter explicitement la liste à un tableau temporaire et ensuite parcourir ce dernier. Par exemple, vous pourriez avoir besoin de changer :

  ```
  foreach $var (grep /x/, @liste) { ... }
  ```

  en

  ```
  foreach $var (my @tmp = grep /x/, @list) { ... }
  ```

  Autrement, la modification de `$var` écrasera aussi les valeurs de `@liste`. (Cela se produit le plus souvent lorsque vous utilisez `$_` comme variable de boucle et que vous appelez des sous-programmes dans la boucle qui ne mettent pas correctement `$_` en local).
- Certains messages d'erreur sont différents.
- Certains bogues ont été éliminés par inadvertance.

Efficacité

Alors que l'essentiel du travail de programmation peut se résumer à un fonctionnement correct, vous pouvez vous retrouver à vouloir parfois un meilleur rapport qualité/prix pour votre programme Perl. Le vaste ensemble d'opérateurs, les types de données et les structures de contrôle de Perl ne sont pas forcément intuitifs quant à la rapidité d'exécution et à l'optimisation de l'espace mémoire. De nombreux compromis ont été nécessaires pendant la conception de Perl et ces décisions sont profondément ancrées dans les entrailles du code. En général, plus un programme est court et simple, plus il est rapide mais il existe des exceptions. Ce paragraphe tente de vous aider à le faire fonctionner juste un petit peu mieux.

Si vous voulez qu'il tourne beaucoup mieux, vous pouvez jouer avec la sortie (*backend*) du compilateur de Perl, décrite au chapitre 18, *Compiler* ou réécrire votre boucle interne dans une extension C, comme l'illustre le chapitre 21, *Mécanismes internes et externes*.

Vous remarquerez que l'optimisation de la vitesse d'exécution est parfois coûteuse en termes d'espace mémoire ou d'efficacité pour le programmeur (on s'en rend compte en constatant que certaines des astuces ci-dessous sont incompatibles entre elles). Voilà le hic. Si la programmation était si simple, il n'y aurait pas besoin de quelque chose d'aussi compliqué qu'un être humain pour s'en occuper, n'est-ce pas?

Efficacité du temps d'exécution

- Utilisez des hachages au lieu de recherches linéaires. Par exemple, au lieu de chercher dans `@mots_cles` pour voir si `$_` est un mot-clé, construisez un hachage avec :

  ```
  my %mots_cles;
  for (@mots_cles) {
      $mots_cles{$_}++;
  }
  ```

  Vous pouvez alors savoir si `$_` contient un mot-clé en testant si `$mots_cles{$_}` donne une valeur non nulle.
- Évitez d'indicer alors qu'un `foreach` ou un opérateur de liste fait l'affaire. Non seulement l'indiçage est une opération supplémentaire, mais si votre variable d'indiçage se trouve être un nombre à virgule flottante après avaoir fait de l'arithmétique, une conversion supplémentaire vers un entier est nécessaire. Il y a souvent mieux à faire. Essayez d'utiliser les opérations `foreach`, `shift` et `splice`. Pensez à écrire `use integer`.
- Évitez `goto`. Il va chercher l'étiquette correspondante hors de l'endroit où vous vous trouvez.
- Évitez `printf` lorsque `print` fait l'affaire.
- Évitez `$&` et ses deux copains, `$‘` et `$’`. La moindre occurrence dans votre programme fait que toutes les correspondances sauvegardent la chaîne recherchée pour une référence ultérieure possible (cela dit, une fois que c'est fait, vous pouvez en mettre autant que vous voulez).
- Évitez d'utiliser `eval` sur une chaîne. Un `eval` d'une chaîne (mais pas d'un bloc) force la recompilation à chaque passage. L'analyseur syntaxique de Perl est plutôt

rapide pour un analyseur, ce qui ne veut pas dire grand-chose. De nos jours, il existe presque toujours un meilleur moyen d'arriver à vos fins. En particulier, tout code utilisant `eval` pour construire des noms de variable est obsolète, puisque vous arrivez maintenant à la même chose en utilisant directement des références symboliques :

```
no strict 'refs';
$nom = "variable";
$$nom = 7;          # Positionne $variable à 7
```

- Évitez `eval` *`CHAINE`* à l'intérieur d'une boucle. Mettez plutôt la boucle dans l'`eval`, pour éviter des compilations redondantes du code. Voir l'opérateur `study` au chapitre 29 pour en trouver un exemple.
- Évitez les motifs compilés à l'exécution. Utilisez le modificateur de motif `/`*`motif`*`/o` (une seule fois) pour éviter les recompilations du motif quand celui-ci ne change pas durant la vie du processus. Pour les motifs qui changent occasionnellement, vous pouvez utiliser le fait qu'un motif nul se réfère au motif précédent, comme ceci :

```
"chaine_trouvee" =~ /$motif_courant/; # Correspondance bidon
                                      # (doit réussir).
while (<>) {
    print if //;
}
```

  Ou alors, vous pouvez précompiler votre expression régulière en utilisant la construction de protection `qr`. Vous pouvez également utiliser `eval` pour recompiler un sous-programme réalisant la recherche de correspondance (si vous ne recompilez qu'occasionnellement). Cela fonction encore mieux si vous compilez une flopée de recherches de correspondances dans un seul sous-programme, en amortissant ainsi la surcharge dûe à l'appel de sous-programme.
- Un court-circuit entre deux choix possibles est souvent plus rapide que l'expression régulière correspondante. Ainsi :

```
print if /une-bosse/ || /deux/;
```

  est susceptible d'être plus rapide que

```
print if /une-bosse|deux/;
```

  du moins pour certaines valeurs de `une-bosse` et de `deux`. En effet, l'optimiseur aime loger certaines opérations simples de recherche de correspondance dans des parties hautes de l'arbre syntaxique et effectue une recherche de correspondance très rapide grâce à un algorithme de Boyer-Moore. Un motif compliqué en fait perdre le bénéfice.
- Rejetez les cas fréquents le plus tôt possible avec `next if`. Tout comme pour les expressions régulières simples, l'optimiseur préfère cela. Et cela a toujours un sens d'éviter le travail superflu. Vous pouvez typiquement éliminer les lignes de commentaire et les lignes vides avant même de faire un `split` ou un `chop` :

```
while (<>) {
    next if /^#/;
    next if /^$/;
    chop;
```

```
        @gorets = split(/,/);
        ...
    }
```

- Évitez les expressions régulières comportant de nombreux quantificateurs ou de grands nombres {*MIN*,*MAX*} pour des expressions entre parenthèses. De tels motifs peuvent engendrer un comportement de retour arrière d'une lenteur exponentielle à moins que les sous-motifs quantifiés ne correspondent à la première « passe ». Vous pouvez également utiliser la construction `(?>...)` pour forcer un sous-motif à correspondre complètement ou à échouer sans retour arrière.
- Essayez de maximiser la longueur de toute chaîne non optionnelle dans les expressions régulières. Ce principe n'est pas intuitif mais les longs motifs trouvent souvent plus vite une correspondance que les courts. En effet, l'optimiseur recherche les chaînes constantes et les passe à une recherche de Boyer-Moore, qui fonctionne mieux avec de longues chaînes. Compilez votre motif avec l'option de débogage **-Dr** pour voir ce que Dr. Perl pense être la chaîne constante la plus longue.
- Évitez les appels de sous-programmes coûteux dans les boucles courtes. L'appel de sous-programmes engendre une certaine surcharge, surtout quand vous leur passez de longues listes de paramètres ou que les valeurs renvoyées sont longues. Essayez, par ordre croissant de désespoir, de passer les valeurs par référence, de passer les valeurs en tant que variables globales de portée dynamique, d'insérer le sous-programme littéralement (*inlining*) ou de réécrire toute la boucle en C. (Si vous pouvez mettre le sous-programme hors d'état de nuire en utilisant un algorithme plus malin, c'est encore mieux que toutes ces solutions.)
- N'employez `getc` que pour les entrées/sorties caractère par caractère sur le terminal. En fait, ne l'utilisez pas du tout. Employez `sysread`.
- Évitez les `substr` fréquents sur les longues chaînes, surtout si la chaîne contient de l'UTF-8. Il n'y a pas de problème pour utiliser `substr` au début d'une chaîne et, pour certaines tâches, vous pouvez garder le `substr` au début : utilisez la forme à quatre arguments de `substr` et « croquez » la chaîne en remplaçant la partie que vous avez happée par `""` :

  ```
  while ($tampon) {
      traiter(substr($tampon, 0, 10, ""));
  }
  ```

- Utilisez `pack` et `unpack` au lieu de plusieurs `substr`.
- Utilisez `substr` en tant que lvalue au lieu de concaténer des sous-chaînes. Par exemple, pour remplacer les caractères allant du quatrième au septième de `$truc` avec le contenu de la variable `$machin`, ne faites pas :

  ```
  $truc = substr($truc,0,3) . $machin . substr($truc,7);
  ```

  Contentez-vous à la place d'identifier la partie de la chaîne à remplacer et affectez-y une valeur, comme dans :

  ```
  substr($truc, 3, 4) = $machin;
  ```

  Mais faites attention que si `$truc` est une chaîne énorme et que `$machin` n'est pas exactement de même longueur que le « trou », cela entraîne également beaucoup de copies. Perl essaie de minimiser cela en copiant soit le début, soit la fin mais il y a trop de travail à accomplir si le `substr` est au milieu.

- Utilisez `s///` au lieu d'une concaténation de sous-chaînes. Surtout si vous pouvez remplacer une constante par une autre de la même taille. La substitution se fait alors sur place.
- Utilisez les modificateurs et les opérateurs équivalents `and` et `or`, au lieu des constructions conditionnelles complètes. Les modificateurs d'instructions (comme `$alliance = 0 unless $fiance`) et les opérateurs logiques évitent la surcharge dûe à l'entrée et à la sortie d'un bloc. De plus, ils sont souvent plus lisibles.
- Utilisez `$truc = $a || $b || $c`. C'est beaucoup plus rapide (et plus court) que :

```
if ($a) {
    $truc = $a;
}
elsif ($b) {
    $truc = $b;
}
elsif ($c) {
    $truc = $c;
}
```

  De même, mettez les valeurs par défaut avec :

```
$pi ||= 3;
```

- Regroupez tous les tests qui demandent la même chaîne initiale. Au moment de tester une chaîne pour y chercher divers préfixes ressemblant un tant soit peu à un branchement conditionnel multiple (*N.d.T.* : *switch structure*), rassemblez tous les motifs `/^a/`, tous les `/^b/`, et ainsi de suite.
- Ne testez pas de choses pour lesquelles vous savez qu'on ne peut pas trouver de correspondance. Utilisez `last` ou `elsif` pour éviter que la recherche de correspondance échoue pour le cas suivant dans votre instruction switch.
- Utilisez des opérateurs spéciaux comme `study`, les opérations logiques sur les chaînes, les formats `pack 'u'` et `unpack '%'`.
- Attention à ne pas utiliser un marteau-pilon pour écraser une mouche. Des instructions mal conçues comme `(<STDIN>)[0]` peuvent exiger de Perl un travail inutile. C'est en plein accord avec la philosophie UNIX que Perl vous donne la corde pour vous pendre.
- Factorisez les opérations hors des boucles. L'optimiseur de Perl n'essaie pas d'enlever le code invariant des boucles. Il vous demande un certain bon sens.
- Les chaînes peuvent être plus rapides que les tableaux.
- Les tableaux peuvent être plus rapides que les chaînes. Tout dépend de la réutilisation ultérieure des chaînes ou des tableaux et de quelles opérations il s'agit. Des modifications importantes de chaque élément conviennent mieux aux tableaux, alors qu'une modification occasionnelle de quelques éléments convient mieux aux chaînes. Mais le plus souvent, vous devez essayer et aviser.
- Les variables `my` sont plus rapides que les variables `local`.
- Le tri selon une clé de tableau fabriquée peut être plus rapide qu'un joli sous-programme de tri. Une valeur de tableau donnée sera généralement comparée plusieurs fois; donc si le sous-programme doit effectuer de nombreux recalculs, il vaut mieux factoriser ce calcul dans une passe séparée avant le tri réel.

- Si vous supprimez des caractères, `tr/abc//d` est plus rapide que `s/[abc]//g`.
- `print` avec une virgule comme séparateur peut être plus rapide que la concaténation de chaînes. Par exemple :

```
print $nom_complet{$nom} . " a un nouveau répertoire home " .
    $home{$nom} . "\n";
```

  doit coller ensemble les deux hachages et les deux chaînes constantes avant de les passer aux routines de bas niveau, ce qui n'est pas le cas de :

```
print $nom_complet{$nom}, " a un nouveau répertoire home ",
    $home{$nom}, "\n";
```

  Cela dit, et selon les valeurs et l'architecture, la concaténation peut être plus rapide. Essayez.
- Préférez `join("", ...)` à une série de chaînes concaténées. Des concaténations multiples peuvent effectuer des copies de chaînes incessantes, ce qu'évite l'opérateur `join`.
- `split` sur une chaîne constante est plus rapide que `split` sur un motif. C'est-à-dire qu'il vaut mieux utiliser `split(/ /, ...)` plutôt que `split(/ +/, ...)` si vous savez qu'il n'y aura qu'un espace. Cependant, les motifs `/\s+/`, `/^/` et `/ /` sont particulièrement optimisés, de même que le `split` spécial sur du blanc.
- Il peut être rentable de pré-étendre un tableau ou une chaîne. Au fur et à mesure de l'accroissement des chaînes et des tableaux, Perl les étend en allouant une nouvelle copie avec de l'espace pour la croissance et en y copiant l'ancienne valeur. La pré-extension d'une chaîne avec l'opérateur `x` ou d'un tableau en positionnant `$#tableau` peut prévenir cette surcharge occasionnelle, ainsi que minimiser la fragmentation mémoire.
- Ne `undef`inissez pas de longues chaînes et de longs tableaux s'ils doivent être réutilisés dans le même but. Ceci afin de ne pas devoir réallouer quand la chaîne ou le tableau devra être ré-étendu.
- Préférez `"\0" x 8192` à `unpack("x8192",())`.
- `system("mkdir ...")` peut être plus rapide sur plusieurs répertoires si l'appel système *mkdir*(2) n'est pas disponible.
- Évitez d'employer `eof()` si les valeurs de retour l'indiquent déjà.
- Mettez dans un cache les entrées des fichiers (comme les fichiers *passwd* et *group*) qui sont susceptibles d'être réutilisées. Il est particulièrement important de mettre dans un cache les entrées provenant du réseau. Par exemple, pour mettre dans un cache la valeur de retour de `gethostbyaddr` lorsque vous convertissez des adresses numériques (comme `204.148.40.9`) en noms (comme « `www.ora.com` »), vous pouvez utiliser quelque chose comme :

```
sub numennom {
    local($_) = @_;
    unless (defined $num_en_nom{$_}) {
        my(@a) = gethostbyaddr(pack('C4', split(/\./)), 2);
        $num_en_nom{$_} = @a > 0 ? $a[0] : $_;
    }
    return $num_en_nom{$_};
}
```

- Évitez les appels système inutiles. Les appels au système d'exploitation sont souvent coûteux. Ainsi, par exemple, n'appelez pas l'opérateur `time` lorsqu'une valeur, `$maintenant`, en cache fait l'affaire. Utilisez le handle de fichier spécial _ pour éviter des appels inutiles à *stat*(2). Sur certains systèmes, même un appel système minimal peut exécuter un millier d'instructions.
- Évitez les appels inutiles à `system`. L'opérateur `system` doit créer un sous-processus et exécuter le programme spécifié — ou pis encore, exécuter un shell pour exécuter le programme spécifié. On en arrive facilement à un million d'instructions.
- Inquiétez-vous de lancer des sous-processus, mais seulement s'ils sont fréquents. Le lancement d'un simple processus *pwd*, *hostname* ou *find* ne vous fera pas beaucoup de mal — après tout, un shell n'arrête pas de lancer des sous-processus toute la journée. Il nous arrive d'encourager l'approche boîte à outils. Mais si, mais si.
- Gardez la trace de votre répertoire de travail courant au lieu d'appeler sans arrêt *pwd*. (Un module est fourni dans la bibliothèque standard à cet effet. Voyez le module `Cwd` au chapitre 30, *Bibliothèque standard*.)
- Évitez les métacaractères du shell dans les commandes — passez au besoin des listes à `system` et à `exec`.
- Mettez le *sticky bit* sur l'interpréteur Perl sur les machines ne disposant pas de pagination à la demande :

```
chmod +t /usr/bin/perl
```

- Pour mettre dans un cache les résultats de fonction, utilisez le module `Memoize` de CPAN.

Efficacité de l'espace mémoire

- Vous pouvez utilisez `vec` pour un stockage compact des tableaux d'entiers si les entiers sont de taille fixe. (Les entiers de taille variable peuvent être stockés dans des chaînes UTF8.)
- Préférez les valeurs numériques aux valeurs de chaînes équivalentes — elles demandent moins de mémoire.
- Employez `substr` pour stocker des chaînes de longueur constante dans une chaîne plus grande.
- Employez le module `Tie::SubstrHash` pour un stockage très compact d'un hachage, si les longueurs de clef et de valeur sont fixées.
- Utilisez `__END__` et le handle de fichier `DATA` pour éviter de stocker des données de programme à la fois comme chaîne et comme tableau.
- Préférez `each` à `keys` lorsque l'ordre n'a pas d'importance.
- Supprimez ou `undef`inissez les variables globales qui ne sont plus utilisées.
- Utilisez un type de DBM pour stocker les hachages.
- Utilisez des fichiers temporaires pour stocker les tableaux.
- Utilisez des pipes pour décharger des traitements vers d'autres outils.
- Évitez les opérations de listes et les absorptions de fichiers.

- Évitez d'utiliser `tr///`. Chaque expression `tr///` doit stocker une table de traduction d'une taille non négligeable.
- Ne déroulez pas vos boucles, n'insérez pas littéralement (*inline*) vos sous-programmes.

Efficacité pour le programmeur

- Utilisez des valeurs par défaut.
- Utilisez les raccourcis géniaux donnés par les options de ligne de commande comme **`-a`**, **`-n`**, **`-p`**, **`-s`** et **`-i`**.
- Utilisez `for` pour signifier `foreach`.
- Lancez des commandes système avec des apostrophes inverses.
- Utilisez `<*>`, etc.
- Utilisez des motifs créés à l'exécution.
- Utilisez `*`, `+` et `{}` sans modération dans vos motifs.
- Traitez des tableaux complets à tour de bras et avalez des fichiers entiers à gogo.
- Utilisez `getc`.
- Utilisez **`$‘`**, **`$&`** et **`$’`**.
- Ne vérifiez pas les valeurs de retour sur `open`, puisque `<`*`HANDLE`*`>` et `print` *`HANDLE`* ne feront rien lorsqu'ils recevront un handle invalide.
- Ne fermez pas vos fichiers avec `close` — ils le seront au prochain `open`.
- Ne passez d'arguments aux sous-programmes. Utilisez des variables globales.
- Ne nommez pas vos paramètres de sous-programmes. Vous pouvez y accéder directement par `$_[`*`EXPR`*`]`.
- Utilisez ce à quoi vous pensez en premier.

Efficacité pour le mainteneur

- N'utilisez pas de valeurs par défaut.
- Utilisez `foreach` pour signifier `foreach`.
- Utilisez des étiquettes de boucles significatives avec `next` et `last`.
- Utilisez des noms de variables significatifs.
- Utilisez des noms de sous-programmes significatifs.
- Mettez ce qui est important en premier sur la ligne en utilisant `and`, `or` et les modificateurs d'instruction (comme `exit if $fini`).
- Fermez vos fichiers dès que vous en avez fini avec eux.
- Utilisez des paquetages, des modules et des classes pour camoufler les détails d'implémentation.
- Passez des arguments en tant que paramètres de sous-programme.

- Nommez vos paramètres de sous-programmes avec `my`.
- Mettez des parenthèses dans un souci de clarté.
- Mettez de nombreux commentaires (utiles).
- Incluez de la documentation pod dans votre code.
- Utilisez `use warnings`.
- Utilisez `use strict`.

Efficacité pour le porteur

- Agitez sous son nez un généreux pourboire sous son nez.
- Évitez les fonctions qui ne sont pas implémentées partout. Vous pouvez utiliser des tests avec `eval` pour voir ce qui est disponible.
- Utilisez le module `Config` ou la variable `$^O` pour trouver sur quelle machine vous êtes en train de tourner.
- Ne vous attendez pas à ce que les flottants et les doubles existent en natif pour `pack` et `unpack` sur des machines exotiques.
- Employez l'ordre des octets pour le réseau (les formats « `n` » et « `N` » de `pack`) pour envoyer des données binaires sur le réseau.
- N'envoyez pas de données binaires sur le réseau. Envoyez de l'ASCII. Ou mieux, envoyez de l'UTF8. Ou encore mieux, envoyez de l'argent.
- Vérifiez `$]` ou `$^V` pour savoir si la version courante implémente toutes les fonctionnalités que vous utilisez.
- N'employez pas `$]` ni `$^V`. Utilisez `require` ou `use` avec un numéro de version.
- Mettez une bidouille `eval exec` même si vous n'en avez pas l'usage, votre programme pourra ainsi fonctionner sur le peu de systèmes qui disposent de shells à la Unix mais qui ne reconnaissent pas la notation `#!`.
- Mettez la ligne `#!/usr/bin/perl` même si vous ne l'utilisez pas.
- Testez les variantes de commandes Unix. Certains programmes *find* ne comprennent pas l'option *-xdev*, par exemple.
- Évitez les variantes des commandes Unix si l'équivalent peut être fait en interne. Les commandes Unix ne fonctionnent pas très bien sous MS-DOS ou VMS.
- Mettez tous vos scripts et toutes vos pages de manuel sur un système de fichiers réseau unique, monté sur toutes vos machines.
- Publiez votre module sur CPAN. Vous recevrez énormément de commentaires s'il n'est pas portable.

Efficacité pour l'utilisateur

- Au lieu d'imposer aux utilisateurs d'entrer les données ligne par ligne, mettez-les dans leur éditeur favori.
- Encore mieux, utilisez une interface graphique (*GUI*) comme Tk, où ils peuvent contrôler l'ordre des événements. (Perl/Tk est disponible sur CPAN.)

- Donnez aux utilisateurs quelque chose à lire pendant que vous continuez à faire votre travail.
- Employez l'autochargement afin que le programme *paraisse* tourner plus vite.
- Donnez la possibilité d'avoir des messages d'aide à chaque invite.
- Donnez un conseil d'utilisation utile si les utilisateurs ne donne pas d'entrées correctes.
- Affichez l'action par défaut à chaque invite, et peut-être quelques autres possibilités.
- Choisissez les valeurs par défaut pour les débutants. Permettez aux experts de les modifier.
- Utilisez une entrée d'un seul caractère là où cela a un sens.
- Modelez la dynamique d'interaction d'après d'autres choses que l'utilisateur connaît bien.
- Rendez les messages d'erreurs explicites sur ce qu'il faut corriger. Fournissez toutes les informations pertinentes comme le nom de fichier et le code d'erreur, comme ceci :

```
open(FICHIER, $fichier)
    or die "$0: Ouverture de $fichier en lecture impossible : $!\n";
```

- Utilisez `fork && exit` pour vous détacher lorsque le reste du script fait du traitement en batch.
- Permettez aux arguments de provenir soit de la ligne de commande, soit de l'entrée standard.
- Ne mettez pas de restrictions arbitraires dans votre programme.
- Préférez les champs de longueur variable aux champs de longueur fixe.
- Utilisez des protocoles réseau orientés texte.
- Dites à tout le monde d'utiliser des protocoles réseau orientés texte !
- Dites à tout le monde de dire à tout le monde d'utiliser des protocoles réseau orientés texte !
- Soyez paresseux par procuration.
- Soyez sympa.

Programmation stylée

Vous aurez certainement vos propres préférences en matière de formatage, mais quelques lignes de conduite rendront vos programmes plus faciles à lire, à comprendre et à maintenir.

Le plus important est de toujours lancer vos programmes sous le pragma `use warnings`. (Vous pouvez désactiver les avertissements dont vous ne voulez pas avec `no warnings`.) Vous devriez également tourner sous le pragma `use strict` ou avoir une bonne excuse pour ne pas le faire. Le pragma `use sigtrap` et même `use diagnostics` peuvent aussi être bénéfiques.

Quant à l'esthétique du code, la seule chose à laquelle Larry prête une certaine importance est que l'accolade fermante d'un *BLOC* multiligne devrait se trouver sur la même colonne que le mot-clé qui a commencé la construction. Par ailleurs, il a d'autres préférences qui ne sont pas si impératives. Les exemples de ce livre respectent ces conventions (ou du moins le devraient) :

- Indentation sur 4 colonnes.
- Une accolade ouvrante doit se trouver sur la même ligne que le mot-clé qui la précède, si possible ; sinon, il faut les aligner verticalement.

```
while ($condition) {     # ici, court : sur la même ligne
    # faire quelque chose
}

# si la condition déborde, aligner les accolades
# l'une au-dessus de l'autre.
while ($une_condition and $autre_condition
       and $encore_une_longue_condition)
{
    # faire quelque chose
}
```

- Mettez un espace avant l'accolade ouvrante d'un *BLOC* multilignes.
- Un *BLOC* court peut être mis sur une seule ligne, accolades comprises.
- Omettez le point-virgule dans un *BLOC* court d'une seule ligne.
- Entourez la plupart des opérateurs avec des espaces.
- Entourez un indice « complexe » (à l'intérieur de crochets) par des espaces.
- Mettez des lignes blanches entre les morceaux de code qui font des choses différentes.
- Mettez un saut de ligne entre une accolade fermante et `else`.
- Ne mettez pas d'espace entre un nom de fonction et sa parenthèse ouvrante.
- Ne mettez pas d'espace avant un point-virgule.
- Mettez un espace après chaque virgule.
- Divisez les longues lignes après un opérateur (mais avant `and` et `or`, même lorsqu'ils s'écrivent `&&` et `||`).
- Alignez verticalement les items correspondants.
- Omettez les ponctuations redondantes tant que la clarté n'en souffre pas.

Larry a ses raisons concernant chacun de ces points mais il ne prétend pas que le cerveau de tout le monde fonctionne (ou ne fonctionne pas) comme le sien.

Voici d'autres questions de style plus substantielles, qui donnent matière à réflexion :

- Ce n'est pas parce que vous *pouvez* faire quelque chose d'une certaine manière que vous *devez* le faire ainsi. Perl est conçu pour fournir plusieurs façons de faire tout ce que l'on veut, il convient donc de choisir la plus lisible. Par exemple :

```
open(TRUC,$truc) or die "ouverture de $truc impossible : $!";
```

vaut mieux que

```
die "Ouverture de $truc impossible : $!" unless open(TRUC,$truc);
```

car la deuxième manière cache l'essentiel de l'instruction dans un modificateur. À l'inverse,

```
print "Début de l'analyse\n" if $verbeux;
```

vaut mieux que

```
$verbeux and print "Début de l'analyse\n";
```

puisque l'essentiel n'est pas de savoir si l'utilisateur a ou non tapé `-v`.

- De même, ce n'est pas parce qu'un opérateur vous permet d'utiliser des arguments par défaut que vous devez en faire usage. Ils sont réservés aux programmeurs paresseux qui écrivent des scripts vite fait, ne servant qu'une fois. Si vous voulez que vos programmes soient lisibles, pensez à fournir les arguments.
- Dans la même lignée, ce n'est pas parce que vous *pouvez* omettre les parenthèses dans de nombreux cas que vous devez le faire :

  ```
  return print reverse sort num values %hachage;
  return print(reverse(sort num (values(%hachage))));
  ```

  En cas de doute, mettez des parenthèses. Au pire, cela permettra aux pauvres idiots de taper sur la touche % dans *vi*.

  Même si *vous* n'avez aucun doute, imaginez l'état mental de la personne qui devra maintenir le code après vous et qui mettra probablement des parenthèses au mauvais endroit.
- Ne vous livrez pas à un exercice stupide de contorsionniste pour sortir en haut ou en bas d'une boucle. Perl fournit l'opérateur `last` qui vous permet d'en sortir au milieu. Vous pouvez optionnellement l'« exdenter » un peu pour le rendre plus visible :

  ```
  LIGNE:
      for (;;) \{
          instructions;
        last LIGNE if $machin;
          next LIGNE if /^#/;
          instructions;
      }
  ```

- N'ayez pas peur d'utiliser des étiquettes de boucles — elles sont là tant pour améliorer la lisibilité que pour permettre les ruptures de boucles sur plusieurs niveaux. Voir l'exemple précédent.
- Évitez d'utiliser `grep`, `map` ou les apostrophes inverses dans un contexte vide, c'est-à-dire lorsque vous ne vous occupez pas des valeurs de retour. Ces fonctions ont toutes des valeurs de retour, qui ne demandent qu'à être utilisées. Sinon, employez une boucle `foreach` ou la fonction `system`.
- Dans un but de portabilité, quand vous employez des fonctionnalités qui peuvent ne pas être implémentées sur toutes les machines, testez-les dans un `eval` pour savoir si elles échouent. Si vous connaissez la version ou le *patch level* (*N.d.T.* : niveau de correction) d'une fonctionnalité particulière, vous pouvez tester `$]` (`$PERL_VERSION` avec le module `English`) pour savoir si telle fonctionnalité est pré-

sente. Le module `Config` permet également de retrouver les valeurs déterminées par le programme *Configure* quand Perl a été installé.

- Choisissez des identificateurs mnémoniques. Si vous ne vous souvenez pas de la signification du terme mnémonique, il y a un problème.
- Alors que des identificateurs courts comme `$reussi` conviennent souvent parfaitement, employez des soulignés pour séparer les mots. Il est généralement plus facile de lire un `$nom_de_variable_comme_ceci` qu'un `$NomDeVariableCommeCeci`, surtout pour des lecteurs dont la langue d'origine n'est pas l'anglais. Même chose pour `$NOM_DE_VARIABLE_COMME_CECI`.
- Les noms des paquetages représentent parfois une exception à cette règle. Perl réserve de façon informelle les noms en minuscule pour des modules de pragmas comme `integer` et `strict`. Les autres modules devraient commencer par une majuscule et mélanger les casses, mais de préférence sans caractère souligné en raison des restrictions de nommage de certains systèmes de fichiers primitifs.
- Il peut s'avérer utile d'utiliser les casses de caractères pour indiquer la visibilité ou la nature d'une variable. Par exemple :

```
$TOUT_EN_MAJUSCULES     # constantes uniquement (attention aux
                        # conflits avec les vars de Perl!)
$Quelques_Majuscules    # vars globales/statiques sur tout un paquetage
$pas_de_majuscules      # vars my() ou local(), dont la portée ne
                        #dépasse pas une fonction
```

  Pour diverses raisons plutôt vagues, les noms de méthodes et de fonctions semblent plus lisibles lorsqu'ils sont entièrement en minuscules. Par exemple, `$obj->en_chaine()`.

  Vous pouvez faire précéder le nom d'un caractère souligné pour indiquer qu'une variable ou une fonction ne devrait pas être utilisée hors du paquetage qui l'a définie. (Perl ne force pas ceci ; ce n'est qu'une forme de documentation.)
- Dans le cas d'une expression régulière réellement inextricable, utilisez le modificateur `/x` et insérez des espaces pour améliorer le rapport signal/bruit.
- N'employez pas de slash comme délimiteurs lorsque votre expression en comporte déjà trop ou contient trop d'antislashs.
- N'utilisez pas les apostrophes ou les guillemets comme délimiteurs lorsque votre chaîne en contient déjà. Utilisez plutôt les pseudo-fonctions `q//`, `qq//` ou `qx//`.
- Utilisez les opérateurs `and` et `or` pour éviter de mettre trop de parenthèses autour des opérateurs de liste, et pour réduire l'incidence des opérateurs de ponctuation comme `&&` et `||`. Appelez vos sous-programmes comme s'il s'agissait de fonctions ou d'opérateurs de liste afin d'éviter un excès d'esperluettes et de parenthèses.
- Utilisez des documents « ici-même » (*here documents*) au lieu d'instructions `print` répétées.
- Alignez verticalement les choses correspondantes, surtout si elles sont trop longues pour tenir sur une ligne :

```
$IDX = $ST_MTIME;
$IDX = $ST_ATIME      if $opt_u;
$IDX = $ST_CTIME      if $opt_c;
```

```
$IDX = $ST_SIZE       if $opt_s;

mkdir $rep_tmp, 0700 or die "mkdir $rep_tmp impossible : $!";
chdir($rep_tmp)      or die "chdir $rep_tmp impossible : $!";
mkdir 'tmp',    0777 or die "mkdir $rep_tmp/tmp impossible : $!";
```

- Ce que nous vous répétons trois fois est vrai :

  Vérifiez toujours les codes de retour des appels système.
  Vérifiez toujours les codes de retour des appels système .
  VÉRIFIEZ TOUJOURS LES CODES DE RETOUR DES APPELS SYSTÈME.

- Les messages d'erreur devaient aller vers `STDERR` et devraient indiquer de quel programme provient l'erreur, ainsi que l'appel de fonction et les arguments. Le plus important est qu'ils devraient contenir, lorsque les appels système ont échoué, le message d'erreur système standard sur ce qui ne va pas. En voici un exemple simple mais suffisant :

  ```
  opendir(R, $rep) or die "opendir $rep impossible : $!";
  ```

- Alignez vos traductions si cela a un sens :

  ```
  tr [abc]
     [xyz];
  ```

- Pensez à la *réutilisabilité*. Pourquoi gâcher votre énergie cérébrale sur des scripts jetables alors que vous êtes susceptible de les réécrire plus tard? Vous pouvez rendre votre code générique. Vous pouvez écrire un module ou une classe d'objets. Vous pouvez faire tourner proprement votre code en activant `use strict` et `-w`. Vous pouvez donner votre code aux autres. Vous pouvez changer votre vision globale du monde. Vous pouvez. . . oh et puis tant pis.
- Soyez cohérent.
- Soyez sympa.

Perl avec aisance

Nous avons abordé quelques idiomes dans les paragraphes précédents (pour ne pas citer les chapitres précédents) mais il y en a beaucoup d'autres que vous verrez couramment si vous lisez des programmes réalisés par des programmeurs Perl accomplis. Lorsque nous parlons de Perl idiomatique dans ce contexte, nous ne parlons pas seulement d'un ensemble arbitraire d'expressions Perl au sens figé. Nous parlons plutôt de code Perl montrant une compréhension des méandres du langage, que vous pouvez avaler tel quel sans vous poser de question. Et quand l'avaler.

Nous n'imaginons même pas énumérer tous les idiomes que vous pourriez rencontrer — cela prendrait tout un livre, aussi épais que celui-ci. Et peut-être même deux. (Voir *Perl en action*, par exemple.) Mais voici quelques idiomes importants, où l'on pourrait définir « important » comme « ce qui incite les persiflages des gens qui pensent déjà savoir comment les langages informatiques devraient fonctionner ».

- Utilisez `=>` en lieu et place d'une virgule partout où vous pensez que cela améliore la lisibilité :

  ```
  return bless $pagaille => $classe;
  ```

Vous devez lire cela comme « Consacrer cette pagaille dans la classe spécifiée. » Faites juste attention à ne pas l'employer après un mot dont vous ne voudriez pas qu'il soit automatiquement mis entre guillemets :

```
sub truc()   { "TRUC" }
sub machin() { "MACHIN"}
print truc => machin;    # affiche trucMACHIN et non TRUCMACHIN.
```

Un autre endroit approprié à l'emploi de `=>` est à proximité d'une virgule littérale que l'on pourrait confondre visuellement :

```
join(", " => @tableau);
```

Perl vous offre plus d'une manière de le faire (*N.d.T.* : *There's More Than One Way To Do It*, le slogan de Perl) vous pouvez donc faire preuve de créativité. Faites-en preuve !

- Utilisez le pronom singulier pour améliorer la lisibilité :

```
for (@lignes) {
    $_ .= "\n";
}
```

La variable `$_` est la version de Perl d'un pronom et il signifie essentiellement « il ».[1] Ainsi, le code ci-dessus signifie « pour chaque ligne, ajoutez-*lui* un saut de ligne. » De nos jours, vous pouvez même écrire cela comme :

```
$_ .= "\n" for @lignes;
```

Le pronom `$_` est tellement important pour Perl qu'il est obligatoire de l'utiliser dans `grep` et `map`. Voici un moyen de configurer un cache pour les résultats communs d'une fonction coûteuse :

```
%cache = map { $_ => couteuse($_) } @args_communs;
$xval = $cache{$x} || couteuse($x);
```

- Omettez le pronom pour améliorer encore plus la lisibilité.[2]
- Utilisez des contrôles de boucles avec des modificateurs d'instructions.

```
while (<>) {
    next if /^=pour\s+(index|après)/;
    $cars += length;
    $mots += split;
    $lignes += y/\n//;
}
```

Il s'agit d'un extrait de code que nous avons utilisé pour compter les pages de ce livre. Lorsque vous vous apprêtez à travailler souvent avec la même variable, il est souvent plus lisible de laisser complètement tomber les pronoms, contrairement à la croyance populaire.

---

1. *N.d.T.* : et puisqu'en français, nous avons la chance de faire la différence entre les genres, il signifie également « elle ». De plus, comme la langue de Molière nous a également fait cadeau d'une différenciation des pronoms selon leur fonction grammaticale, `$_` peut finalement vouloir dire « il », « elle », « le », « la » ou « lui ». Pauvres anglo-saxons : ils n'ont pas plus d'une manière de le dire !

2. Dans ce paragraphe, plusieurs items énumérés d'affilée se réfèrent à l'exemple qui suit, puisque certains de nos exemples illustrent plus d'un idiome.

Cet extrait démontre également l'utilisation idiomatique de `next` avec un modificateur d'instruction pour court-circuiter une boucle.

La variable `$_` est toujours la variable de contrôle de boucle dans `grep` et `map` mais la référence du programme pointant dessus est souvent implicite :

```
$longueur_hachage = grep { length } @aleatoire;
```

Nous prenons ici une liste de scalaires aléatoires et ne choisissons que ceux qui ont une longueur supérieure à `0`.

- Utilisez `for` pour positionner l'antécédent d'un pronom :

```
for ($episode) {
    s/fred/barney/g;
    s/wilma/betty/g;
    s/pebbles/bambam/g;
}
```

Mais que se passe-t-il s'il n'y a qu'un seul élément dans la boucle ? Il s'agit d'un moyen pratique de positionner « le », c'est-à-dire `$_`. En linguistique, cela s'appelle une topicalisation. Ce n'est pas de la triche, c'est de la communication.

- Référencez implicitement le pronom pluriel, `@_`.
- Utilisez les opérateurs de contrôle de flux pour fixer les valeurs par défaut :

```
sub aboyer {
    my Chien $medor = shift;
    my $qualite = shift || "jappant";
    my $quantite = shift || "sans_arrêt";
    ...
}
```

Nous avons employé ici l'autre pronom de perl, `@_`, qui signifie « eux ».[3] Les arguments d'une fonction arrivent toujours comme « eux ». L'opérateur `shift` sait qu'il doit travailler sur `@_` si vous l'omettez, tout comme l'employé de Disneyland crierait « Au suivant ! » sans préciser quelle file d'attente est supposée avancer. (Il n'y a pas lieu de spécifier car il n'y a qu'une seule queue qui importe.)

Le `||` peut être utilisé pour fixer les valeurs par défaut malgré ses origines d'opérateur booléen, puisque Perl renvoie la première valeur qui soit vraie. Les programmeurs Perl font souvent preuve d'une attitude assez cavalière avec la vérité ; la ligne ci-dessus ne fonctionnerait pas si, par exemple, vous essayiez de spécifier une quantité nulle. Mais tant que vous ne voulez jamais positionner `$qualite` ou `$quantite` à une valeur fausse, l'idiome marche parfaitement. Il n'y a pas lieu de devenir superstitieux et de lancer partout des appels à `defined` et `exists`. Tant que la valeur ne devient pas accidentellement fausse, tout ira bien.

- Utilisez les formes d'affectation des opérateurs, y compris les opérateurs de contrôle de flux :

```
$xval = $cache{$x} ||= couteuse($x);
```

Ici nous n'initialisons pas du tout notre cache. Nous ne faisons que nous reposer sur l'opérateur `||=` pour appeler `couteuse($x)` et ne l'affecter à `$cache{$x}` que si

3. ou « elles », ou « les », ou « leur », ou « eux ».

`$cache{$x}` est faux. Le résultat de ceci est la nouvelle valeur de `$cache{$x}` quelle qu'elle soit. Encore une fois, nous avons adopté une approche quelque peu cavalière de la vérité, en ce que si nous mettons en cache une valeur fausse, `couteuse($x)` sera à nouveau appelée. Le programmeur sait peut-être que cela fonctionne car `couteuse{$x}` n'est pas coûteuse lorsqu'elle renvoie une valeur fausse. Ou peut-être que le programmeur a simplement tendance à bâcler son travail. Ce qui peut être analysé comme une forme de créativité.

- Utilisez les contrôles de boucles en tant qu'opérateurs, pas seulement en tant qu'instructions. Et. . .
- Utilisez les virgules comme de petits points-virgules :

```
while (<>) {
    $commentaires++, next  if /^#/;
    $blancs++, next        if /^\s*$/;
    last                   if /^__END__/;
    $code++;
}
print "commentaires = $commentaires\nblancs = $blancs\ncode = $code\n";
```

  Ceci montre que nous comprenons que les modificateurs d'instructions modifient les instructions, alors que `next` est un simple opérateur. Cela montre également que la virgule, lorsqu'elle est utilisée de façon idiomatique pour séparer les expressions, exactement comme vous le feriez avec un point-virgule. (La différence est que la virgule garde les deux expressions comme faisant partie de la même instruction, sous le contrôle de l'unique modificateur d'instruction.)

- Utilisez le contrôle de flux à votre avantage :

```
while (<>) {
    /^#/       and $commentaires++;
    /^s*$/     and $blancs++;
    /^__END__/ and last;
    $code++;
}
print "commentaires = $commentaires\nblancs = $blancs\ncode = $code\n";
```

  Il s'agit exactement de la même boucle, seulement cette fois-ci, les motifs sont au début. Le programmeur Perl perspicace comprend que la compilation donne exactement le même code interne que l'exemple précédent. Le modificateur `if` n'est qu'une conjonction `and` (ou `&&`) inversée et le modificateur `unless` est une conjonction `or` (ou `||`) inversée.

- Utilisez les boucles implicites fournies par les options **`-n`** et **`-p`**.
- Ne mettez pas de point-virgule à la fin d'un bloc d'une seule ligne :

```
#!/usr/bin/perl -n
$commentaires++, next LINE if /^#/;
$blancs++, next LINE       if /^\s*$/;
last LINE                  if /^__END__/;
$code++;

END {print "commentaires = $commentaires\nblancs = $blancs\ncode =
$code\n" }
```

Il s'agit essentiellement du même programme que précédemment. Nous avons mis une étiquette *LINE* (*LIGNE* en anglais) explicite dans les opérateurs de contrôle de boucle parce que nous en avions envie mais nous n'y étions pas obligés, puisque la boucle `LINE` implicite, fournie par **`-n`** est la boucle encadrante la plus intérieure. Nous avons employé `END` pour avoir l'instruction `print` finale à l'extérieur de la boucle principale implicite, exactement comme dans *awk*.

- Utilisez des documents « ici-même » (*here documents*) lorsque les affichages deviennent monstrueux.
- Utilisez un délimiteur significatif dans les documents « ici-même » :

```
END { print <<"COMPTEURS" } commentaires = $commentaires
blancs = $blancs
code = $code
COMPTEURS
```

Au lieu d'utiliser plusieurs `print`, le programmeur parlant Perl couramment emploie une chaîne multiligne avec interpolation. Bien que nous ayons appelé ceci une « étourderie courante » auparavant, nous avons effrontément enlevé le point-virgule final car il n'est pas nécessaire à la fin du bloc `END`. (Si nous le changeons un jour en bloc multiligne, nous remettrons le point-virgule.)

- Faites des substitutions et des traductions *en passant*[4] sur un scalaire :

```
($nouveau = $ancien) =~ s/mauvais/bon/g;
```

Puisque les lvalues sont lvaluables[5], pour ainsi dire, vous verrez souvent les gens changer une valeur *en passant* alors qu'elle est en train d'être affectée. Ceci peut en fait éviter une copie de chaîne en interne (si nous arrivons jamais à implémenter l'optimisation) :

```
chomp($reponse = <STDIN>);
```

Toute fonction qui modifie un argument sur place peut utiliser l'astuce *en passant*. Mais attendez, il y a mieux !

- Ne vous limitez pas à modifier les scalaires *en passant* :

```
for (@nouveau = @ancien) { s/mauvais/bon/g }
```

Ici, nous copions `@ancien` dans `@nouveau`, en modifiant tout *en passant* (pas tout d'un coup, bien entendu — le bloc est exécuté de manière répétitive, un « il » (c'est-à-dire `$_`) à la fois).

- Passez les paramètres nommés en utilisant l'opérateur virgule de fantaisie, `=>`.
- Fiez-vous à l'affectation à un hachage pour traiter un argument pair/impair :

```
sub aboyer {
    my Chien $medor = shift;
    my %param = @_
```

4. *N.d.T.* : en français dans le texte.

5. *N.d.T.* : les lvalues sont des valeurs (*values*, en anglais) que l'on peut mettre à gauche (l*eft*, en anglais) d'une affectation. Il y a donc ici un jeu de mots que l'on peut traduire littéralement par « puisque les valeurs que l'on peut mettre à gauche d'une affectation peuvent être mises à gauche d'une affectation. »

```
    my $qualite  = $param{QUALITE}  || "jappant";
    my $quantite = $param{QUANTITE} || "sans_arrêt";
    ...
}

$fido->aboyer( QUANTITE => "une_fois",
               QUALITE => "ouaf" );
```

Les paramètres nommés sont souvent un luxe abordable. Et avec Perl, vous les avez pour rien, si vous ne comptez pas le coût de l'affectation de hachage.

- Répétez les expressions booléennes jusqu'à obtenir une valeur fausse.
- Utilisez les recherches de correspondances minimales lorsqu'elles sont appropriées.
- Utilisez le modificateur `/e` pour évaluer une expression de remplacement :

```
#!/usr/bin/perl -p
1 while s/^(.*)?(\t+)/$1 . ' ' x length($2) *4 - length($1) %4)/e;
```

Ce programme corrige tout fichier que vous recevez de la part de quelqu'un pensant par erreur qu'il peut redéfinir les tabulations matérielles pour qu'elles occupent 4 espaces au lieu de 8. Il utilise plusieurs idiomes importants. Tout d'abord, l'idiome `1 while` est commode lorsque tout le travail que vous voulez accomplir dans la boucle est en fait déjà réalisé par la condition. (Perl est assez malin pour ne pas vous avertir que vous êtes en train d'utiliser `1` dans un contexte vide.) Nous devons répéter cette substitution car chaque fois que nous remplaçons un certain nombre d'espaces par des tabulations, nous devons recalculer depuis le début la colonne où doit se trouver la prochaine tabulation.

Le `(.*?)` correspond à la plus petite chaîne possible jusqu'à la première tabulation, en utilisant le modificateur de correspondance minimal (le point d'interrogation). Dans ce cas, nous aurions pu utiliser un * gourmand comme d'ordinaire, ainsi : `([^\t]*)`. Mais cela ne marche que parce qu'une tabulation est un seul caractère, nous avons donc employé une classe de caractères avec une négation pour éviter de dépasser la première tabulation. En général, la correspondance minimale est plus élégante et elle n'échoue pas si la prochaine chose qui doit correspondre fait plus d'un caractère.

Le modificateur `/e` réalise une substitution en utilisant une expression plutôt qu'une chaîne. Cela nous permet d'effectuer les calculs dont nous avons besoin au moment exact où nous en avons besoin.

- Utilisez une mise en page et des commentaires créatifs pour les substitutions complexes :

```
#!/usr/bin/perl -p
1 while s{
    ^               # attachement au début
    (               # commence le premier regroupement
        .*?         # correspond au nombre de caractères minimal
    )               # termine le premier regroupement
    (               # commence le second regroupement
        \t+         # correspond à une ou plusieurs tabulations
    )               # termine le second regroupement
}
```

```
{
    my $lg_espaces = length($2) * 4;     # comptage des tabulations
                                         # complètes
    $lg_espaces -= length($1) % 4;       # longueur de la tabulation
                                         # incomplète
    $1 . ' ' x $lg_espaces;              # calcule le nombre d'espaces
                                         # correct
}ex;
```

Nous en faisons probablement un peu trop mais certaines personnes trouvent cet exemple plus impressionnant que le précédent qui tenait sur une seule ligne. À vous de voir.

- Allez de l'avant et utilisez `$‘` si vous le sentez :

  ```
  1 while s/(\t+)/' ' x (length($1) * 4 - length($‘) % 4)/e;
  ```

  Il s'agit ici d'une version plus courte, qui utilise `$‘`, bien connu pour son impact sur les performances. Sauf que nous n'utilisons que sa longueur, alors ça ne compte pas vraiment.

- Utilisez les décalages directement à partir des tableaux `@-` (`@LAST_MATCH_START`) et `@+` (`@LAST_MATCH_END`) :

  ```
  1 while s/\t+/' ' x (($+[0] - $-[0]) * 4 - $-[0] % 4)/e;
  ```

  Cet exemple-ci est encore plus court. (Si vous ne voyez pas de tableaux ici, essayez à la place de rechercher des éléments de tableaux.) Voir `@-` et `@+` au chapitre 28.

- Utilisez eval avec une valeur de retour constante :

  ```
  sub est_un_motif_valide {
      my $motif = shift;
      return eval { "" =~ /$motif/; 1 } || 0;
  }
  ```

  Vous n'avez pas à utiliser l'opérateur `eval {}` pour renvoyer un valeur réelle. Ici, nous renvoyons toujours `1` si la correspondance parvient à se terminer. Toutefois, si le motif contenu dans `$motif` explose, l'`eval` intercepte ceci et renvoie `undef` à la condition booléenne de l'opérateur `||`, qui transforme cela vers la valeur définie `0` (simplement pour être poli, puisque `undef` est également faux mais pourrait conduire quelqu'un à penser que la routine `est_un_motif_valide` s'est mal comportée et nous ne voudrions pas que cela se produise, n'est-ce pas ?).

- Utilisez des modules pour faire le sale boulot.
- Utilisez des usines d'objets.
- Utilisez des fonctions de rappel (*callbacks*).
- Utilisez des piles pour garder une trace du contexte.
- Utilisez des indices négatifs pour accéder à la fin d'un tableau ou d'une chaîne :

  ```
  use XML::Parser;

  $p = new XML::Parser Style => 'subs';
  setHandlers $p Char => sub { $sorties[-1] .= $_[1] };

  push @sorties, "";
  ```

```
sub literal {
    $sorties[-1] .= "C<";
    push @sorties, "";
}

sub literal_ {
    my $texte = pop @sorties;
    $sorties[-1] .= $texte . ">";
}
...
```

Il s'agit d'un court extrait d'un programme de 250 lignes que nous avons utilisé pour retraduire la version XML de l'ancienne édition de ce livre[6] en format pod afin que nous puissions l'ouvrir pour la présente édition dans un Véritable Éditeur de Texte.

La première chose que vous remarquerez est que nous nous appuyons sur le module `XML::Parser` (de CPAN) pour analyser notre XML correctement, nous n'avons donc pas à comprendre comment le faire. Cela soulage déjà notre programme de quelque milliers de lignes (en supposant que nous réimplémentions en Perl tout ce que `XML::Parser` accomplit pour nous,[7] y compris la traduction de presque tous les ensembles de caractères vers de l'UTF-8).

`XML::Parser` utilise un idiome de haut niveau appelé une *usine d'objets*. Dans ce cas, il s'agit d'une usine d'analyseur. Lorsque nous créons un objet `XML::Parser`, nous lui disons quel style d'interface d'analyseur nous voulons et il en crée un pour nous. C'est un excellent moyen de construire une plate-forme de test pour une application lorsque vous n'êtes pas sûr de savoir quelle interface sera la meilleure à long terme. Le style `subs` ne représente qu'une des interfaces de `XML::Parser`. En fait, il s'agit de l'une des plus anciennes interface et probablement pas la plus populaire de nos jours.

La ligne `setHandlers` montre un appel de méthode sur l'analyseur, pas en notation fléchée, mais dans la notation « d'objet indirect », qui vous permet, entre autres, d'omettre les parenthèses autour des arguments. La ligne utilise également l'idiome des paramètres nommés que nous avons vu plus tôt.

La ligne montre également un autre concept très puissant, la notion de fonction de rappel (*callback*). Plutôt que d'appeler l'analyseur pour obtenir l'élément suivant, nous lui demandons de nous appeler. Pour les balises XML nommées, comme `<literal>`, ce style d'interface appellera automatiquement un sous-programme de ce nom (ou le nom avec un souligné à la fin pour la balise fermante correspondante). Mais les données entre les balises n'ont pas de nom, nous configurons donc

6. *N.d.T.* : le « Camel book » (littéralement le livre au chameau) est le surnom couramment employé par les fans de Perl (les *Perl Mongers*) pour désigner le livre que vous avez entre les mains ainsi que ses illustres ancêtres. Bien entendu, ceci est dû au dessin de la couverture (heu... c'est en fait un dromadaire, voir le Colophon). Selon les recherches approfondies des Perl Mongers, le chameau de cette troisième édition porte le doux nom d'Amélia, alors que celui de la précédente édition se prénommait Fido.

7. En fait, `XML::Parser` n'est qu'un enrobage pratique autour de l'analyseur XML *expat* de James Clark.

une fonction de rappel, `Char`, avec la méthode `setHandlers`.

Ensuite, nous initialisons le tableau `@sorties`, qui est une pile des affichages. Nous y mettons une chaîne vide pour représenter le fait que nous n'avons collecté aucun texte à ce niveau d'imbrication des balises (0, initialement).

C'est maintenant que réapparaît la fonction de rappel. Dès que nous voyons un texte, il est automatiquement ajouté après l'élément final du tableau, *via* l'idiome `$sortie[-1]` dans la fonction de rappel. Au niveau de balise le plus extérieur, `$sorties[-1]` est équivalent à `$sortie[0]`, `$sorties[0]` termine donc notre affichage complet. (Éventuellement. Mais nous devons d'abord traiter les balises.)

Supposez que nous voyions une balise `<literal>`. Alors le sous-programme `literal` est appelé, ajoute du texte à l'affichage courant, puis pousse un nouveau contexte dans la pile `@sorties`. Dès lors, tout texte rencontré jusqu'à la balise fermante se voit ajouté à cette nouvelle fin de la pile. Lorsque nous rencontrons la balise fermante, nous dépilons (avec `pop`) le `$texte` que nous avons collecté hors de la pile `@sorties` et nous ajoutons le reste des données métamorphosées à la nouvelle (c'est-à-dire, l'ancienne) fin de la pile, dont le résultat est la traduction de la chaîne XML, `<literal>`*texte*`</literal>`, vers la chaîne pod correspondante, `C<`*texte*`>`.

Les sous-programmes pour les autres programmes sont exactement les mêmes, à part qu'ils sont différents.

- Utilisez `my` sans affectation pour créer un tableau ou un hachage vide.
- Éclatez la chaîne par défaut selon les espaces qu'elle contient.
- Affectez aux listes de variables pour en collecter autant que vous voulez.
- Utilisez l'autovivification des références indéfinies pour les créer.
- Autoincrémentez les élements indéfinis de tableau ou de hachage pour les créer.
- Utilisez l'autoincrémentation d'un hachage `%vus` pour déterminer l'unicité.
- Affectez à une variable temporaire `my` bien pratique dans la condition.
- Utilisez le comportement de protection automatique des accolades.
- Utilisez un mécanisme de protection alternatif pour interpoler les guillemets.
- Utilisez l'opérateur `?:` pour choisir entre deux arguments d'un `printf`.
- Alignez verticalement les arguments de `printf` avec leur champ `%` :

```
my %vus;
while (<>) {
    my ($a, $b, $c) = split;
    print unless $vus{$a}{$b}{$c}++;
}
if (my $tmp = $vus{nifnif}{nafnaf}{noufnouf}) {
    printf qq("nifnif nafnaf noufnouf" vu %d coup%s\n),
                                          $tmp,  $tmp == 1 ? "" : "s";
}
```

Ces neuf lignes débordent d'idiomes. La première ligne donne un hachage vide car nous n'y affectons rien. Nous parcourons itérativement les lignes entrées en positionnant « le », c'est-à-dire `$_`, implicitement, puis nous utilisons un `split` sans arguments qui « l' » éclate selon les espaces qu'« il » contient. Ensuite nous pre-

nons les trois premiers mots avec une affectation de liste, en rejetant les mots suivants. Puis nous mémorisons les trois premiers mots dans un hachage à trois dimensions, qui crée automatiquement (si nécessaire) les deux premiers éléments de référence et le dernier élément de comptage pour l'autoincrémentation à incrémenter. (Avec `use warnings`, l'autoincrémentation ne vous avertira jamais que vous utilisez des valeurs indéfinies car l'autoincrémentation est un moyen accepté pour définir des valeurs indéfinies.) Nous imprimons alors la ligne si nous n'en avons jamais vue avant commençant par ces trois mots, car l'autoincrémentation est une postincrémentation, qui, en plus d'incrémenter la valeur du hachage, renverra l'ancienne valeur vraie s'il y en avait une.

Après la boucle, nous testons `%vus` à nouveau pour voir si une combinaison particulière de trois mots a été vue. Nous utilisons le fait que nous pouvons mettre un identificateur littéral entre accolades pour qu'il soit automatiquement protégé, comme avec des guillemets. Sinon, nous aurions dû écrire `$vus{"nafnaf"}{"nifnif"}{"noufnouf"}`, ce qui est une corvée même lorsque vous n'êtes pas poursuivi par un grand méchant loup.

Nous affectons le résultat de `$vus{nifnif}{nafnaf}{noufnouf}` à une variable temporaire avant même de la tester dans le contexte booléen fourni par le `if`. Comme l'affectation renvoie sa valeur de gauche, nous pouvons toujours tester la valeur pour voir si elle était vraie. Le `my` vous fait sauter aux yeux qu'il s'agit d'une nouvelle variable et que nous ne vérifions pas une égalité mais réalisons une affectation. Cela aurait également parfaitement marché sans le `my` et un programmeur Perl expert aurait encore remarqué immédiatement que nous utilisions un `=` au lieu de deux `==`. (Un programmeur Perl intermédiaire aurait cependant pu être bluffé. Le programmeur Pascal de n'importe quel niveau aura de l'écume aux lèvres.)

Si l'on passe à l'instruction `printf`, vous pouvez voir la forme `qq()` des guillemets que nous avons employée afin de pouvoir interpoler les guillemets ordinaires ainsi que le retour à la ligne. Nous aurions aussi bien pu interpoler directement ici `$tmp`, puisqu'il s'agit effectivement d'une chaîne entre guillemets mais nous avons choisi de pousser plus loin l'interpolation *via* `printf`. Notre variable temporaire `$tmp` est maintenant bien pratique, particulièrement depuis que nous ne voulons plus seulement l'interpoler mais également la tester dans la condition d'un opérateur `?:` pour voir si nous devions mettre le mot « coup » au pluriel. Enfin, remarquez que nous avons aligné verticalement les deux champs avec leur marqueurs `%` correspondants dans le format de `printf`. Si un argument est trop long pour coïncider, vous pouvez toujours aller à la ligne suivante pour l'argument suivant bien que nous n'ayons pas eu à le faire dans ce cas.

Wouah ! Vous en avez assez ? Il existe encore beaucoup d'autres idiomes dont nous aurions pu débattre mais ce livre est déjà suffisamment lourd. Mais nous aimerions encore parler d'un usage idiomatique de Perl, l'écriture de générateurs de programmes.

Générer des programmes

Depuis le jour, ou presque, où les gens se sont aperçus pour la première fois qu'ils pouvaient écrire des programmes, ils ont commencé à écrire des programmes qui écrivaient d'autres programmes. Nous les appelons souvent des *générateurs de programmes*. (Les his-

toriens se souviennent que RPG signifiait *Report Program Generator* bien avant *Role Playing Game*.) De nos jours, on le appellerait probablement des « usines à programmes » mais les gens des des générateurs sont arrivés les premiers, c'est donc leur nom qui a prévalu.

Quiconque a déjà écrit un générateur de programmes sait que cela peut vous donner un strabisme divergent quand bien même vous portez des verres correcteurs. Le problème vient de ce qu'une grande partie des données du programme ressemble à du code, a la couleur du code, mais n'est pas du code (du moins pas encore). Le même fichier texte contient à la fois des choses qui font quelque chose et des choses qui y ressemblent mais qui ne font rien. Perl comporte de nombreuses fonctionnalités qui facilitent son intégration avec d'autres langages, textuellement parlant.

(Bien sûr, ces fonctionnalités facilitent l'écriture de Perl en Perl, mais c'est le moins auquel on puisse s'attendre dès à présent.)

Générer d'autres langages en Perl

Perl est (entre autres) un langage de traitement de texte et la plupart des langages informatiques sont textuels. De plus, l'absence de limites arbitraires et les divers mécanismes de protection et d'interpolation de Perl facilitent la séparation visuelle du code d'un autre langage que l'on génère. Par exemple, voici un petit morceau de *s2p*, le traducteur *sed*-vers-*perl* :

```
print &q(< <"FIN");
:       #!$bin/perl
:       eval 'exec $bin/perl -S \$0\${1+"\$@"}'
:               if \$lance_sous_un_shell;
:
FIN
```

Il se trouve que le texte encadré est légal en deux langages, Perl et *sh*. Nous avons utilisé dès le début un idiome qui préservera votre santé mentale lors de l'écriture d'un générateur de programme : l'astuce de mettre un caractère « de bruit » et une tabulation au début de chaque ligne protégée, ce qui isole visuellement le code inclus, vous pouvez ainsi dire en un clin d'œil qu'il ne s'agit pas du code qui est en train de s'exécuter. Une variable, `$bin`, est interpolée en deux endroits de la chaîne multiligne protégée, puis la chaîne est passée à travers une fonction qui élimine le deux-points et la tabulation.

Bien sûr, il est inutile d'utiliser des chaînes protégées multilignes. On voit souvent des scripts CGI contenant des millions d'instructions `print`, une par ligne. C'est un peu comme aller au supermarché en Mirage III, mais bon, tant que l'on arrive à destination... (Nous admettrons qu'une colonne d'instructions `print` possède ses propres propriétés pour ressortir visuellement.)

Lorsque vous encadrez une grosse chaîne protégée multiligne contenant un autre langage (comme du HTML), il est parfois utile de faire comme si vous programmiez à l'envers, en encadrant plutôt du Perl dans l'autre langage, exactement comme vous le feriez avec des langages ouvertement inversés comme PHP :

```
print <<"XML";
   <truc>
   <insense>
```

```
    bla bla bla @{[ scalar EXPR ]} bla bla bla
    bla bla bla @{[ LISTE ]} bla bla bla
    </insense>
    </truc>
XML
```

Vous pouvez utiliser l'une de ces deux astuces pour interpoler des valeurs d'expressions arbitrairement complexes dans une grosse chaîne.

Certains générateurs de programmes ne ressemblent pas beaucoup à des générateurs de programmes, selon la quantité d'opérations qu'ils vous cachent. Au chapitre 22, *CPAN*, nous avons vu comment un petit programme *Makefile.PL* pouvait être utilisé pour écrire un *Makefile*. *Makefile* peut facilement être 100 fois plus gros que le *Makefile.PL* qui l'a généré. Pensez à l'usure que cela épargne à vos doigts. Ou n'y pensez plus — après tout, c'est le but.

Générer du Perl dans d'autres langages

Il est facile de générer d'autres langages en Perl, mais l'inverse est également vrai. Perl peut facilement être généré par d'autres langages parce qu'il est à la fois concis et malléable. Vous pouvez choisir votre manière de protéger les chaînes entre guillemets afin de ne pas interférer avec les mécanismes de protection des autres langages. Vous n'avez pas à vous soucier de l'indentation, ni où placer vos ruptures de lignes, ni si vous devez (encore) mettre un antislash avant vos antislashs. Vous n'avez pas besoin de définir un paquetage à l'avance par une seule chaîne de caractères, puisque vous pouvez glisser dans l'espace de noms du paquetage de façon répétée à chaque fois que vous voulez évaluer plus de code dans ce paquetage.

Une autre chose qui facilite l'écriture de Perl dans d'autres langages (y compris Perl) est la directive `#line`. Perl sait comment la traiter comme une directive spéciale qui reconfigure l'idée qu'il se faisait du nom de fichier courant et du numéro de ligne. Ceci peut s'avérer utile dans les messages d'erreur ou d'avertissement, en particulier pour les chaînes traitées avec `eval` (ce qui, lorsqu'on y pense, n'est rien d'autre que du Perl écrivant du Perl). La syntaxe de ce mécanisme est celle utilisée par le préprocesseur C : lorsque Perl rencontre un symbole `#` et le mot `line`, suivi par un numéro et un nom de fichier, il positionne `__LINE__` à ce numéro et `__FILE__` à ce nom de fichier.[8]

Voici quelques exemples que vous pouvez tester en tapant dans *perl* directement. Nous avons employé un Ctrl-D pour indiquer la fin de fichier, ce qui est typique d'Unix. Les utilisateurs de DOS/Windows et de VMS peuvent taper Ctrl-Z. Si votre shell utilise autre chose, vous devrez l'employer pour dire à *perl* que vous avez terminé. Sinon, vous pouvez toujours taper `__END__` pour dire au compilateur qu'il n'y a plus rien à analyser.

Voici comment la fonction interne de Perl, `warn`, affiche le nouveau nom de fichier est le numéro de ligne :

8. Techniquement parlant, Perl fait une correspondance avec le motif `/^#\\s*line\\s+(\\d+)\\s*(?:\\s"([^"]+)")?\\s*$/`, avec `$1` fournissant le numéro de ligne pour la ligne suivante et `$2` le nom de fichier optionnel spécifié entre les guillemets. (Un nom de fichier vide laisse `__FILE__` inchangé.)

```
% perl
# line 2000 "Odyssée"
# le "#" sur la ligne précédente doit être le premier caractère de la ligne
warn "les portes du compartiment pod"; # or die
^D
les portes du compartiment pod at Odyssée line 2001.
```

Ici, l'exception levée par le `die` à l'intérieur de l'`eval` retrouve ses petits dans la variable `$@` (`$EVAL__ERROR`), accompagnée du nouveau nom de fichier temporaire et du numéro de ligne :

```
# line 1996 "Odyssée"
eval qq {
#line 2025 "Hal"
    die "portes du compartiment pod";
};
print "Problèmes avec $@";
warn "J'ai peur de ne pouvoir faire cela",
^D
Problème avec les portes du compartiment pod at Hal line 2025.
J'ai peur de ne pouvoir faire cela at Odyssée line 2001.
```

Ceci montre comment une directive `#line` n'affecte que l'unité de compilation courante (le fichier ou l'`eval` *`CHAINE`*) et que, lorsque la compilation de cette unité est terminée, la configuration précédente est automatiquement restaurée. De cette manière vous pouvez positionner vos propres messages à l'intérieur d'un `eval` *`CHAINE`* ou d'un `do` *`FICHIER`* sans affecter le reste de votre programme.

Perl possède une option **`-P`** qui invoque le préprocesseur C, qui émet les directives `#line`. Le préprocesseur C était l'élan initial pour implémenter `#line`, mais il est rarement utilisé de nos jours, depuis qu'il existe de meilleurs moyens pour faire ce pour quoi nous nous reposions sur lui. Perl possède toutefois de nombreux autre préprocesseurs, y compris le module `AutoSplit`. Le préprocesseur JPL (*Java Perl Lingo*) change les fichiers *.jpl* en fichiers *.java*, *.pl*, *.h* et *.c*. Il utilise `#line` pour conserver la précision des messages d'erreurs.

Un des tous premiers préprocesseurs de Perl était le traducteur *sed*-vers-*perl*, *s2p*. En fait Larry a retardé la première version de Perl pour terminer *s2p* et *awk*-vers-*perl* (*a2p*) car il pensait qu'ils amélioraient la manière dont Perl serait perçu. Hum, peut-être que cela a été le cas.

Voir les documentations en ligne pour plus d'information à ce sujet, ainsi que sur le traducteur *find2perl*.

Filtres de code source

Si vous savez écrire un programme pour traduire des choses quelconques en Perl, alors pourquoi ne pas disposer d'un moyen pour invoquer ce traducteur depuis Perl ?

La notion de filtre de code source est née avec l'idée qu'un script ou un module devrait être capable de se décrypter lui-même à la volée, comme ceci :

```
#!/usr/bin/perl
use MonFiltreDeDecryptage;
```

```
@*x$]‘OuN&k^ZxO2jZ^X{.?s!(f;9Q/^A^@~~8H]|,%@^P:q-=
...
```

Mais l'idée a évolué depuis et maintenant un filtre de code source peut être défini pour faire toutes les transformations que vous désirez sur un texte en entrée. Mettez ceci en rapport avec la notion d'option **-x** mentionnée au chapitre 19, *L'interface de la ligne de commande*, et vous obtiendrez un mécanisme général pour extraire d'un message n'importe quel morceau de programme et pour l'exécuter, indépendamment du fait qu'il soit écrit ou non en Perl.

En utilisant le module `Filter` de CPAN, on peut même maintenant faire des choses comme programmer du Perl en *awk* :

```
#!/usr/bin/perl
use Filter::exec "a2p";       # le traducteur awk-vers-perl
1,30 { print $1 }
```

Maintenant voila précisément ce que vous pourriez qualifier d'idiomatique. Mais nous ne prétendons pas une seule seconde qu'il s'agit d'une technique couramment employée.

25

Portabilité

Un monde ne comportant qu'un seul système d'exploitation rendrait la portabilité facile et la vie ennuyeuse. Nous préférons un large champ génétique de systèmes d'exploitations, tant que l'écosystème ne se divise pas trop clairement en prédateurs et en proies. Perl tourne sur des douzaines de systèmes d'exploitation et puisque les programmes Perl ne sont pas dépendants de la plate-forme, le même programme peut tourner sur tous ces systèmes sans modification.

Enfin, presque. Perl essaie d'offrir au programmeur autant de fonctionnalités que possible mais si vous utilisez des fonctionnalités spécifiques à un certain système d'exploitation, vous réduirez forcément la portabilité de votre programme sur d'autres systèmes. Dans ce paragraphe, nous donnerons quelques lignes de conduite pour écrire du code Perl portable. Une fois que vous aurez pris votre décision sur le degré de portabilité que vous souhaitez, vous saurez où tirer un trait et vous saurez quelle limites vous décidez de ne pas franchir.

Si l'on regarde sous un autre angle, l'écriture de code portable entraîne une limitation volontaire des choix disponibles. Il n'existe aucune raison pour ne pas utiliser Perl pour lier ensemble des outils Unix, ou pour prototyper une application Macintosh, ou pour gérer la base de registre de Windows. Si cela a un sens de sacrifier la portabilité, allez-y.[1]

En général, remarquez que les notions de UID, de répertoire « maison » et même l'état d'être « logué » n'existeront que sur des plates-formes multiutilisateur.

La variable spéciale `$^O` vous indique sur quel système d'exploitation Perl a été compilé. Ceci permet d'accélérer le code qui autrement aurait dû faire `use Config` pour obtenir la même information *via* `$Config{osname}`. (Même si vous avez chargé Config pour d'autres raisons, cela vous économise une recherche dans un hachage lié.)

1. Toute conversation ne se doit pas d'être multiculturellement exacte. Perl essaie de vous donner au moins une manière de faire la Chose À Faire mais n'essaie pas de vous l'imposer strictement. À cet égard, Perl se rapproche plus de votre langue maternelle que de la langue de votre nourrice.

Pour obtenir des informations plus détaillées sur la plate-forme, vous pouvez rechercher le reste de l'information dans le hachage `%Config`, qui est disponible grâce au module standard `Config`. Par exemple, pour vérifier si la plate-forme dispose de l'appel `lstat`, vous pouvez tester `$Config{d_lstat}`. Voir la documentation en ligne de `Config` pour une description complète des variables accessibles et la page de manuel *perlport* pour une liste des comportements des fonctions internes de Perl sur différentes plates-formes. Voici les fonctions Perl dont le comportement varie le plus entre les plates-formes.

`-X` (tests de fichiers), `accept`, `alarm`, `bind`, `binmode`, `chmod`, `chown`, `chroot`, `connect`, `crypt`, `dbmclose`, `dbmopen`, `dump`, `endgrent`, `endhostent`, `endnetent`, `endprotoent`, `endpwent`, `endservent`, `exec`, `fcntl`, `fileno`, `flock`, `fork`, `getgrent`, `getgrgid`, `getgrnam`, `gethostbyaddr`, `gethostbyname`, `gethostent`, `getlogin`, `getnetbyaddr`, `getnetbyname`, `getnetent`, `getpeername`, `getpgrp`, `getppid`, `getpriority`, `getprotobyname`, `getprotobynumber`, `getprotoent`, `getpwent`, `getpwnam`, `getpwuid`, `getservbyport`, `getservent`, `getservbyname`, `getsockname`, `getsockopt`, `glob`, `ioctl`, `kill`, `link`, `listen`, `lstat`, `msgctl`, `msgget`, `msgrcv`, `msgsnd`, `open`, `pipe`, `qx`, `readlink`, `readpipe`, `recv`, `select`, `semctl`, `semget`, `semop`, `send`, `sethostent`, `setgrent`, `setnetent`, `setpgrp`, `setpriority`, `setprotoent`, `setpwent`, `setservent`, `setsockopt`, `shmctl`, `shmget`, `shmread`, `shmwrite`, `shutdown`, `socket`, `socketpair`, `stat`, `symlink`, `syscall`, `sysopen`, `system`, `times`, `truncate`, `umask`, `utime`, `wait`, `waitpid`

Sauts de ligne

Sur la plupart des systèmes d'exploitations, les lignes dans les fichiers sont terminées par un ou deux caractères qui signalent la fin de ligne. Les caractères varient d'un système à l'autre. Unix utilise traditionnellement `\012` (c'est-à-dire le caractère octal 12 en ASCII), un type d'entrées/sorties à la DOS utilise `\015\012` et les Mac utilisent `\015`. Perl emploie `\n` pour représenter un saut de ligne « logique », indépendamment de la plate-forme. Sur MacPerl, `\n` signifie toujours `\015`. Sur les Perl à la DOS, `\n` signifie généralement `\012`, mais lorsqu'on accède à un fichier en « mode texte », il est traduit vers (ou depuis) `\015\012`, selon que vous lisez ou que vous écrivez. Unix fait la même chose sur les terminaux en mode canonique. On se réfère généralement à `\015\012` en l'appelant CRLF.

Comme DOS fait une distinction entre les fichiers texte et les fichiers binaires, les Perl à la DOS ont des restrictions sur l'utilisation de `seek` et `tell` sur un fichier en « mode texte ». Pour obtenir les meilleurs résultats possibles, ne faites `seek` que sur des emplacements obtenus depuis `tell`. Si toutefois vous utilisez la fonction interne de Perl `binmode` sur le handle de fichier,vous pouvez généralement faire `seek` et `tell` en toute impunité.

Un malentendu fréquent dans la programmation de sockets est que `\n` vaudra `\012` partout. Dans beaucoup de protocoles Internet habituels, `\012` et `\015` sont spécifiés et les valeurs de `\n` et `\r` de Perl ne sont pas fiables puisqu'elles varient d'un système à l'autre :

```
print SOCKET "Salut, le client !\015\012";     # correct
print SOCKET "Salut le client !\r\n";          # FAUX
```

Toutefois, l'utilisation de `\015\012` (ou `\cM\cJ`, ou encore `\x0D\x0A`, ou même `v13.10`) peut devenir pénible et disgracieux, et semer la confusion chez ceux qui maintiennent

le code. Le module `Socket` fournit quelques Choses Correctes pour ceux qui en veulent :

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Salut le client !$CRLF"          # correct
```

Lors d'une lecture sur une socket, souvenez-vous que le séparateur d'enregistrement d'entrée par défaut `$/` est `\n`, ce qui signifie que vous avez encore du travail à faire si vous n'êtes pas sûrs que de ce que vous verrez à travers la socket. Un code de socket robuste devrait reconnaître soit `\012` ou `\015\012` comme étant la fin de ligne :

```
use Socket qw(:DEFAULT :crlf);
local ($/) = LF         # pas nécessaire si $/ vaut déjà /012

while (<SOCKET>) {
    s/$CR?$LF/\n/;      # remplace LF ou CRLF par un saut de ligne logique
}
```

De même, un code renvoyant des données textuelles — comme un sous-programmes qui va chercher une page web — devrait souvent traduire les sauts de ligne. Souvent, une seule ligne de code suffit :

```
$donnees =~ s/\015?\012/\n/g;
return $donnees;
```

Boutisme et taille des nombres

Les ordinateurs stockent les entiers et les nombres à virgule flottante dans des ordres différents (*gros-boutiste*, *big-endian* ou *petit-boutiste*, *little-endian*) et dans des tailles différentes (32 bits et 64 bits étant les tailles les plus courantes de nos jours). Normalement, vous n'aurez pas à y penser. Mais si votre programme envoie des données binaires à travers une connexion réseau ou s'il écrit sur le disque des données destinées à être lues par un ordinateur différent, il se peut que vous deviez prendre des précautions

Des ordres rentrant en conflit peuvent semer une pagaille absolue dans les nombres. Si un hôte petit-boutiste (comme une CPU Intel) stocke 0x12345678 (305 419 896 en décimal), un hôte grand-boutiste (comme une CPU Motorola) le lira comme 0x78563412 (2 018 915 346 en décimal). Pour éviter ce problème dans les connexions réseau (sur une socket), utilisez `pack` et `unpack` avec les formats `n` et `N`, qui écrivent des nombres entiers courts et longs dans l'ordre gros-boutiste (également appelé ordre « réseau ») quelle que soit la plate-forme.

Vous pouvez explorer le boutisme de votre plate-forme en dépaquetant une structure de données empaquetée en format natif, comme ceci :

```
print unpack("h*", pack("s2", 1, 2)), "\n";
# '10002000' sur par ex. Intel x86 ou Alpha 21064 en mode petit-boutiste
# '00100020' sur par ex. Motorola 68040
```

Pour déterminer votre boutisme, vous pourriez utiliser l'une de ces deux instructions :

```
$est_gros_boutiste  = unpack("h*", pack("s", 1)) =~ /01/;
$est_petit_boutiste = unpack("h*", pack("s", 1)) =~ /^1/;
```

Même si deux systèmes ont le même boutisme, il peut encore y avoir des problèmes lors du transfert de données entre des plates-formes 32-bits et 64-bits. Il n'existe pas de bonne

solution autre que d'éviter de transférer ou de stocker des nombres binaires bruts. Soit vous transférez et vous stockez les nombres en texte plutôt qu'en binaire, soit vous utilisez des modules comme `Data::Dumper` ou `Storable` pour le faire à votre place. Vous avez tout intérêt à utiliser des protocoles orientés texte en toutes circonstantes — ils sont plus robustes, plus maintenables et encore plus extensibles que les protocoles binaires.

Bien entendu, avec l'avènement de XML et d'Unicode, notre définition d'un protocole orienté texte devient encore plus souple. Par exemple, vous pouvez transférer, entre deux systèmes où Perl 5.6.0 (ou une version ultérieure) est installé, une séquence d'entiers encodés en tant que caractères en `utf8` (la version UTF-8 de Perl). Si les deux extrémités tournent sur une architecture avec des entiers 64-bits, vous pouvez échanger des entiers 64-bits. Sinon, vous êtes limités à des entiers 32-bits. Utilisez `pack` avec un canevas `U*` pour envoyer et `unpack` avec un canevas `U*` pour recevoir.

Fichiers et systèmes de fichiers

Les composants des chemins de fichiers sont séparés par `/` sur Unix, `\` sur Windows et `:` sur Mac. Certains systèmes n'implémentent ni les liens en dur (`link`), ni les liens symboliques (`symlink`, `readlink`, `lstat`). Certains systèmes attachent une importance à la casse (majuscules/minuscules) des noms de fichiers, d'autres non et d'autres encore n'y font attention que lors de la création de fichiers mais pas lors de leur lecture.

Il existe des modules qui peuvent vous aider. Les modules standards `File::Spec` fournissent des fonctions qui Font La Bonne Chose :

```
use File::Spec::Functions;
chdir( updir() );         # monter dans le répertoire supérieur
$fichier = catfile( curdir(), 'temp', 'fichier.txt');
```

La dernière ligne lit dans *./temp/fichier.txt* sur Unix et Windows ou *:temp:fichier.txt* sur Mac ou *[.temp]fichier.txt* sur VMS et stocke le contenu du fichier dans `$fichier`.

Le module `File::Basename`, un autre module livré avec Perl qui est également indépendant de la plate-forme, éclate un nom de chemin en ses composants : le nom de fichier de base, le chemin complet vers le répertoire et le suffixe du fichier.

Voici quelques astuces pour écrire des programmes Perl manipulant des fichiers qui soient portables :

- N'utilisez pas deux fichiers du même nom avec des casses différentes, comme *test.pl* et *Test.pl*, puisque certaines plates-formes ignorent la casse.
- Limitez vos noms de fichiers à la convention 8.3 (noms de huit lettres et extensions de trois lettres) partout où c'est possible. Vous pouvez souvent vous en sortir avec des noms de fichiers plus longs pour autant que vous soyez sûrs qu'ils resteront uniques lorsqu'on les aura poussés à travers un trou dans le mur de taille 8.3. (Hé, cela s'avère plus facile que de pousser un chameau à travers le chas d'une aiguille.)
- Réduire l'emploi de caractères non alphanumériques dans les noms de fichiers. L'utilisation de soulignés est souvent correcte mais elle gaspille un caractère qui pourrait être mieux employé pour l'unicité sur les systèmes 8.3. (Souvenez-vous que c'est la raison pour laquelle nous ne mettons habituellement pas de souligné dans les noms de modules.)

- De même, lorsque vous utiliser le module `AutoSplit`, essayez de limiter vos noms de sous-programmes à huit caractères au plus et ne donnez pas le même nom avec une casse différente à deux sous-programmes. Si vous avez besoin de plus de huit caractères, arrangez-vous pour qu'il y ait unicité sur les huit premiers.
- Utilisez toujours `<` explicitement pour ouvrir un fichier en lecture ; sinon, sur les systèmes qui autorisent la ponctuation dans les noms de fichiers, un fichier préfixé par un caractère `>` pourrait être effacé et un fichier préfixé par `|` pourrait engendrer une ouverture de pipe. Car la forme à deux arguments d'`open` est magique et elle interprétera les caractères comme `>`, `<` et `|`, ce qui peut être la mauvaise chose à faire. (Sauf si c'est la bonne).

```
open(FICHIER,      $fichier_existant) or die $!;   # MAUVAIS
open(FICHIER,    "<$fichier_existant) or die $!;   # mieux
open(FICHIER, "<", $fichier_existant) or die $!;   # encore mieux
```

- Ne présupposez pas que les fichiers texte finiront par un saut de ligne. Ils le devraient, mais les gens oublient quelquefois, particulièrement lorsque leur éditeur de texte les y aide.

Interactions avec le système

Les plates-formes reposant sur une interface graphique sont parfois démunis de toute espèce de ligne de commande, ainsi les programmes exigeant une interface en ligne de commande pourraient ne pas fonctionner partout. Vous n'y pouvez pas grand chose, à part mettre à jour.

Quelques autres astuces :

- Certaines plates-formes ne peuvent pas supprimer ou renommer des fichiers en train d'être utilisés, souvenez-vous donc de fermer avec `close` les fichiers lorsque vous en avez fini avec eux. Ne faites pas `unlink` ou `rename` sur un fichier ouvert. Ne faites pas `tie` ou `open` sur un fichier déjà lié avec `tie` ou déjà ouvert ; faites d'abord `untie` ou `close` sur ce fichier.
- N'ouvrez pas le même fichier plusieurs fois simultanément, car certains systèmes posent d'office des verrous sur les fichiers ouverts.
- Ne dépendez pas d'une variable d'environnement spécifique existant dans `%ENV` et ne présupposez pas que quoi que ce soit dans `%ENV` soit insensible à la casse ou préserve la casse. Ne présumez pas de la sémantique de l'héritage d'Unix pour les variables d'environnement ; sur certains systèmes, elles peuvent être visibles par tous les processus.
- N'utilisez pas les signaux, ni `%SIG`.
- Tâchez d'éviter les globs de noms de fichiers. Utilisez `opendir`, `readdir` et `closedir` à la place. (Depuis la version 5.6.0 de Perl, le glob basique de noms de fichiers est bien plus portable qu'il ne l'était mais certains systèmes peuvent encore se frotter aux « Unixismes » de l'interface par défaut si vous essayez d'être original.)
- Ne présumez pas des valeurs spécifiques pour les numéros d'erreur ou les chaînes stockées dans `$!`.

Communication interprocessus (IPC)

Pour maximiser la portabilité, n'essayez pas de lancer de nouveaux processus. Cela signifie que vous devriez éviter `system`, `exec`, `fork`, `pipe`, ``` `` ```, `qx//` ou `open` avec un `|`.

Le problème principal ne réside pas dans les opérateurs eux-mêmes : les commandes lançant des processus externes sont généralement implémentées sur la plupart des plates-formes (bien que certaines n'implémentent aucun type de `fork`). Les problèmes sont d'avantage susceptibles de se produire lorsque vous invoquez des programmes externes qui ont des noms, des emplacements, des sorties ou des sémantiques pour les arguments qui varient entre les plates-formes.

Un morceau de code Perl particulièrement populaire est l'ouverture d'un pipe vers *sendmail* pour que votre programme puisse envoyer un mail :

```
open(MAIL, '|/usr/lib/sendmail -t') or die "fork sendmail impossible: $!";
```

Ceci ne marchera pas sur les plates-formes sans *sendmail*. Pour une solution portable, utilisez l'un des modules de CPAN pour envoyer votre mail, comme `Mail::Mailer` et `Mail::Send` dans la distribution MailTools ou `Mail::Sendmail`.

Les fonctions des IPC System V d'Unix (`msg*()`, `sem*()`, `shm*()`) ne sont pas toujours disponibles, même sur certaines plates-formes Unix.

Sous-programmes externes (XS)

Il est généralement possible d'élaborer un code XS pour qu'il fonctionne avec n'importe quelle plate-forme mais les bibliothèques et les fichiers d'en-tête pourraient ne pas être disponibles partout. Si les bibliothèques et les fichiers d'en-tête sont portables, alors il y a une chance raisonnable pour que le code XS le soit également.

Un problème de portabilité différent se produit lors de l'écriture de code XS : la disponibilité d'un compilateur C sur la plate-forme de l'utilisateur final. C apporte avec lui ses propres problèmes de portabilité et l'écriture de code XS vous exposera à certains d'entre eux. L'écriture en Perl pur est un moyen facile d'atteindre la portabilité car le processus de configuration de Perl traverse des supplices extrêmes pour vous cacher les tares de portabilité du C.[2]

Modules standard

En général, les modules standard (ceux qui sont livrés avec Perl) fonctionnent sur toutes les plates-formes. Les exceptions notables sont le module `CPAN.pm` (qui réalise en fait des connexions vers des programmes externes qui peuvent ne pas être disponibles), les modules spécifiques à une plate-forme (comme `ExtUtils::MM_VMS`) et les modules `DBM`.

Il n'existe pas un seul module DBM qui soit disponible sur toutes les plates-formes.

2. Certaines personnes en marge de la société lancent le script Configure de Perl comme une forme d'amusement bon marché. Certaines personnes sont même connues pour organiser des « courses de Configure » entre des systèmes concurrents et parient dessus de grandes sommes d'argent. Cette pratique est désormais hors-la-loi dans la majeure partie du monde civilisé.

`SDBM_File` et les autres sont généralement disponibles sur tous les portages Unix et à la DOS mais pas dans MacPerl, où seuls `NBDM_File` et `DB_File` sont disponibles.

La bonne nouvelle est qu'au moins un module DBM devrait être disponible et que `AnyDBM_File` utilisera n'importe quel module qu'il sera capable de trouver. Avec une telle incertitude, vous ne devriez utiliser que les fonctionnalités communes à toutes les implémentations DBM. Par exemple, limitez vos enregistrements à 1 K octet, pas plus. Voir la documentation du module `AnyDBM_File` pour plus de détails.

Dates et heures

Là où c'est possible, utilisez le standard ISO-8601 (« *AAAA-MM-JJ* ») pour représenter les dates. Les chaînes comme « `1971-06-21` » peuvent facilement être converties vers une valeur spécifique au système avec un module comme `Date::Parse`. Une liste de valeurs de date et d'heure (comme celle renvoyée par la fonction interne `localtime`) peut être convertie vers une représentation spécifique au système en utilisant `Time::Local`.

La fonction interne `time` renverra toujours le nombre de secondes depuis le commencement de l'« epoch » (l'origine, voir la fonction `time` au chapitre 29) mais les systèmes d'exploitation ont des opinions divergentes concernant la date à laquelle cela s'est produit. Sur la plupart des systèmes, l'origine est fixée au 1er janvier 1970 à 00:00:00 UTC mais elle a commencé 66 ans plus tôt sur Mac et sur VMS elle a démarré le 17 novembre 1858 à 00:00:00. Ainsi, pour obtenir des dates portables, vous voudrez peut-être calculer un décalage pour l'origine :

```
require Time::Local;
$decalage = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

La valeur de `$decalage` sur Unix et Windows vaudra toujours `0` mais sur Macs et VMS, elle pourra valoir un nombre plus grand. Il est alors possible d'ajouter `$decalage` à une valeur de date et d'heure Unix pour obtenir ce qui devrait être la même valeur sur tous les systèmes.

La représentation par le système de l'heure du jour et de la date calendaire peut être contrôlée par des moyens largement différents. Ne présupposez pas que le fuseau horaire soit stocké dans `$ENV{TZ}`. Et même s'il l'est, ne présupposez pas que vous pouvez le contrôler *via* cette variable.

Internationalisation

Utilisez de l'Unicode dans vos programmes. Faites toutes les conversions depuis et vers d'autres jeux de caractères dans vos interfaces vers le monde extérieur. Voir le chapitre 15, *Unicode*.

En dehors du monde d'Unicode, vous devriez présumez de peu en ce qui concerne les jeux de caractères et de rien du tout à propos des valeurs de `ord` pour les caractères. Ne présupposez pas que les caractères alphabétiques aient des valeurs de `ord` séquentielles. Les lettres minuscules peuvent venir avant ou après les majuscules ; les majuscules et les minuscules peuvent être mêlées de manière à ce qu'à la fois `a` et `A` viennent avant `b` ; les caractères accentués et les autres caractères internationaux peuvent être mêlés pour que `ä` vienne avant `b`.

Si votre programme opère sur un système POSIX (une hypothèse plutôt large), consultez la page de manuel de *perllocale* pour plus d'informations sur les locales de POSIX. Les locales affectent les jeux de caractères et les encodages ainsi que la date et l'heure, entre autres choses. Une utilisation correcte des locales rendra votre programme un peu plus portable ou au moins, plus pratique et nativement amical pour les utilisateurs non anglo-saxons. Mais soyez conscients que les locales et Unicode ne se mélangent pas encore très bien.

Style

Lorsqu'il est nécessaire d'avoir du code spécifique à la plate-forme, pensez à le garder à un endroit qui facilite le portage vers d'autres plates-formes. Utiliser le module `Config` et la variable spéciale `$^O` pour marquer les différences entre les plates-formes.

Faites attention aux tests que vous fournissez avec votre module ou vos programmes. Le code d'un module peut être totalement portable mais ses tests peuvent ne pas l'être tout à fait. Ceci arrive souvent lorsque les tests engendrent d'autres processus ou appellent des programmes externes pour faciliter les contrôles ou lorsque (comme on l'a remarqué ci-dessus) les tests présument de certaines choses concernant le système de fichiers et les chemins. Prenez garde à ne pas dépendre d'un style d'affichage spécifique pour les erreurs, même en vérifiant `$!` pour les erreurs « standards » après un appel système. Utilisez plutôt le module `Errno`.

Souvenez-vous qu'un bon style transcende à la fois le temps et les cultures, ainsi, pour une portabilité maximale, vous devez chercher à comprendre ce qui est universel parmis les exigences de votre existence. Les personnes les plus équilibrées ne sont pas prisonnières et n'ont pas à l'être car elles ne se soucient pas d'être à la mode en égard à leur propre culture, de programmation ou autre. La mode est variable mais le style est constant.

26

POD

Un des principes sous-jacents dans la conception de Perl est que les choses simples devraient rester simples et que les choses difficiles devraient être possibles. La documentation devrait rester simple.

Perl implémente un format de balises simple appelé *pod* qui peut avoir son existence propre ou être librement entremêlé avec votre code source pour créer une documentation insérée. Pod peut être converti vers différents formats pour l'impression ou la consultation ou vous pouvez simplement le lire directement car il est à plat (*plain* en anglais).

Pod n'est pas aussi expressif que des langages comme XML, LATEX, *troff*(1) ou même HTML. C'est une volonté de notre part : nous avons sacrifié l'expressivité au profit de la simplicité et de la commodité. Certains langages à balises font écrire aux auteurs plus de balises que de texte, ce qui rend l'écriture plus difficile qu'elle ne l'était et la lecture proche de l'impossible. Un bon format, comme une bonne musique de film, reste en arrière-plan sans distraire le spectateur.

Amener les programmeurs à écrire de la documentation est presque aussi difficile que de les amener à porter des cravates. Pod a été conçu pour être si simple à écrire que même un programmeur puisse le faire — et il devrait le faire. Nous ne prétendons pas que pod est suffisant pour écrire un livre, bien qu'il fût suffisant pour écrire celui-ci.

Pod en quelques mots

La plupart des formats de document exigent que le document entier soit dans ce format. Pod est plus indulgent : vous pouvez insérer du pod dans tout type de fichier, en vous reposant sur les *traducteurs de pod* pour l'extraire. Certains fichiers sont entièrement constitués de pod pur à 100 %. Mais d'autres, notamment les programmes Perl et les modules, peuvent contenir une belle quantité de pod dispersée de-ci de-là au bon vouloir de l'auteur. Il suffit à Perl de sauter le texte en pod lorsqu'il analyse le fichier en vue de l'exécuter.

L'analyseur lexical de Perl sait où commencer à sauter le pod lorsqu'à un certain endroit

où il trouverait d'ordinaire une instruction, il rencontre à la place une ligne commençant par un signe égal et un identificateur, comme ceci :

```
=head1 Ici se trouvent les pods !
```

Ce texte, ainsi que tout ce qui se trouve jusqu'à et y compris une ligne commençant par `=cut`, sera ignoré. Cela vous permet d'entremêler votre code source et votre documentation en toute liberté, comme dans :

```
=item guinde

La fonction guinde() se comportera de la façon la plus spectaculairement
coincée que vous puissiez imaginer, avec aussi peu de naturel qu'une
pyrotechnie cybernétique.

=cut

sub guinde() {
    my $arg = shift;
    ...
}

=item guincher

La fonction guincher() permet la génération autodidact  d'épistémologie.

=cut

sub guincher {
    print "La génération d'épistémologie n'est pas implémentée sur
           cette plate-forme.\n";
}
```

Pour d'autres exemples, regardez dans n'importe quel module standard ou venant de CPAN. Ils sont tous supposés venir avec du pod, et pratiquement tous le font, excepté ceux qui ne le font pas.

Puisque le pod est reconnu par l'analyseur lexical de Perl et rejeté, vous pouvez également utiliser une directive pod appropriée pour inhiber rapidement une section de code arbitrairement longue. Utilisez un bloc pod `=for` pour inhiber un paragraphe ou une paire `=begin`/`=end` pour un section plus grosse. Nous détaillerons la syntaxe de ces directives pod plus tard. Souvenez-vous cependant que dans les deux cas, vous êtes toujours dans le mode pod après ceci, vous avez donc besoin de faire un `=cut` pour revenir au compilateur.

```
print "reçu 1\n";

=for comment

Ce paragraphe seul est ignoré par tout le monde sauf
le mythique traducteur « comment » (N.d.t : commentaire).
Lorsque c'est fait, vous êtes toujours en mode pod
et non en mode programme.
```

```
print "reçu 2\n";
=cut

# ok, à nouveau dans le programme réel
print "reçu 3\n";

=begin commentaire

print "reçu 4\n";

tout ce qui se trouve ici
sera ignoré
par tout le monde

print "reçu 5\n";

=end commentaire

=cut

print "reçu 6\n";
```

Ceci affichera qu'on a reçu 1, 3 et 6. Souvenez-vous que ces directives pod ne peuvent pas aller partout. Vous devez les mettre seulement là où l'analyseur s'attend à trouver une nouvelle instruction, pas juste au milieu d'une expression, ni à d'autres endroits arbitraires.

Du point de vue de Perl, tout ce qui est balisé par pod est rejeté mais du point de vue des traducteurs de pod, c'est le code qui est rejeté. Les traducteurs de pod voient le texte qui reste comme une séquence de paragraphes séparés par des lignes blanches. Tous les traducteurs de pods modernes analysent le pod de la même manière, en utilisant le module standard `Pod::Parser`. Ils ne se différencient que par leur sortie, puisque chaque traducteur se spécialise dans un format de sortie.

Il existe trois sortes de paragraphes : les paragraphes « tels quels » (*verbatim*), les paragraphes de commandes et les paragraphes de prose.

Paragraphes « tels quels »

Les paragraphes « tels quels » sont utilisés pour du texte littéral que vous voulez voir apparaître tel qu'il est, comme des extraits de code. Un paragraphe « tel quel » doit être indenté ; c'est-à-dire qu'il doit commencer par un caractère espace ou de tabulation. Le traducteur devrait le reproduire exactement, généralement avec une fonte de largeur fixe, et en alignant les tabulations sur des colonnes de huit caractères. Il n'y a pas de formatage spécial des séquences d'échappement, vous ne pouvez donc pas jouer sur les fontes pour mettre en italique ou en relief. Un caractère < signifie un < littéral et rien d'autre.

Directives pod

Toutes les directives pod commence par = suivi d'un identificateur. Ce dernier peut être suivi par autant de texte arbitraire que la directive peut utiliser de quelque manière qui lui plaise. La seule exigence syntaxique est que le texte doit être tout entier dans un paragraphe. Les directives actuellement reconnues (appelées parfois *commandes pod*) sont :

`=head1`
`=head2`
`...`

Les directives `=head1`, `=head2`,... produisent des en-têtes du niveau spécifié. Le reste du texte dans le paragraphe est traité comme étant la description de l'en-tête. Elle sont identiques aux en-têtes de section et de sous-section `.SH` et `.SS` dans *man*(7) ou aux balises `<h1>...</h1>` et `<h2>...</h2>` en HTML. En fait, ces traducteurs convertissent ces directives exactement comme cela.

`=cut`

La directive `=cut` indique la fin du mode pod. (Il peut y avoir encore du pod plus loin dans le document mais, si c'est le cas, il sera introduit par une autre directive pod.)

`=pod`

La directive `=pod` ne fait rien d'autre que dire au compilateur d'arrêter d'analyser le code jusqu'au prochain `=cut`. Elle est utile pour ajouter un autre paragraphe au document si vous mélangez beaucoup du code et du pod.

`=over` *`NOMBRE`*
`=item` *`SYMBOLE`*
`=back`

La directive `=over` commence un paragraphe spécifiquement pour la génération d'une liste utilisant la directive `=item`. À la fin de votre liste, utilisez `=back` pour la terminer. Le *`NOMBRE`*, s'il est fourni, indique au formateur de combien d'espaces indenter. Certains formateurs ne sont pas assez riches pour respecter l'indication, alors que d'autres le sont *trop* pour la respecter, tant il est difficile, lorsqu'on travaille avec des fontes proportionnelles, d'aligner le texte simplement en comptant les espaces. (Toutefois, on présume en général qu'avec quatre espaces il y a assez de place pour une puce ou un numéro.)

Le véritable type de la liste est indiqué par le *`SYMBOLE`* sur chaque élément. Voici une liste à puces :

```
=over 4

=item *

Armure de mithril

=item *

Manteau elfique

=back
```

Une liste numérotée :

```
=over 4

=item 1.

Premièrement, dites "ami".

=item 2.

Deuxièmement, entrez dans la Moria.

=back
```

Une liste de définitions :

```
=over 4

=item armure()

Description de la fonction armure()

=item chant()

Description de la fonction chant()
```

Vous pouvez imbriquer des listes de types identiques ou différents mais il faut appliquer certaines règles de base : n'utilisez pas `=item` en dehors d'un bloc `=over`/`=back` ; utilisez au moins un `=item` à l'intérieur d'un bloc `=over`/`=back` ; et peut-être le plus important, conservez une cohérence dans le type des items dans une liste donnée. Utilisez soit `=item *` avec chaque élément pour produire une liste à puces, soit `=item 1.`, `=item 2.`, et ainsi de suite pour produire une liste numérotée, soit utilisez `=item truc`, `=item machin`, et ainsi de suite pour produire une liste de définitions. Si vous commencez avec des puces ou des numéros, gardez-les, puisque les formateurs sont autorisés à utiliser le type du premier `=item` pour décider de comment formater la liste.

Comme pour tout ce qui concerne pod, le résultat n'est pas meilleur que le traducteur. Certains traducteurs accordent de l'attention aux nombres particuliers (ou aux lettres, ou aux chiffres romains) qui suivent l'`=item` et d'autres non. Le traducteur *pod2html* actuel, par exemple, est quelque peu cavalier : il vide entièrement la séquence d'indicateurs sans les regarder pour en déduire quelle séquence vous utilisez, puis il enveloppe la liste complète à l'intérieur de balises `<ol>` et `</ol>` pour que le navigateur puisse afficher une liste ordonnée en HTML. Ce ne doit pas être interprété comme une fonctionnalité, cela pourra éventuellement être corrigé.

```
=for TRADUCTEUR
=begin TRADUCTEUR
=end TRADUCTEUR
```

`=for`, `=begin` et `=end` vous permettent d'inclure des paragraphes spéciaux à passer inchangés aux formateurs, mais seulement à des formateurs particuliers. Les formateurs reconnaissant leur propre nom ou des alias pour leur nom, dans *TRADUCTEUR* font attention à ces directives ; tous les autres les ignorent complètement. La direc-

tive `=for` spécifie que le reste de ce paragraphe n'est prévu que pour (*for*, en anglais) un traducteur particulier.

```
=for html
<p> Voici un paragraphe en <small>HTML</small> <flash>brut</flash> </p>
```

Les directives appariées `=begin` et `=end` fonctionnent de manière identique à `=for` mais, au lieu de n'accepter qu'un seul paragraphe, elles traitent tout le texte compris entre le `=begin` et le `=end` correspondant comme étant destiné à un traducteur particulier. Quelques exemples :

```
=begin html

<br>Figure 1.<img src="figure.png"><br>

=end html

=begin text

  ------------------
  |  truc          |
  |        machin  |
  ------------------

^^^^ Figure 1. ^^^^

=end text
```

Les valeurs de *TRADUCTEUR* couramment acceptées par les formateurs comprennent `roff`, `man`, `troff`, `nroff`, `tbl`, `eqn`, `latex`, `tex`, `html` et `text`. Certains formateurs accepteront certains d'entre eux comme synonymes. Aucun traducteur n'accepte `comment` — il s'agit juste du mot habituel pour quelque chose devant être ignoré par tout le monde. N'importe quel mot non reconnu pourrait jouer le même rôle. En écrivant ce livre, nous avons souvent laissé des remarques pour nous mêmes sous la directive `=for later` (pour plus tard).

Remarquez que `=begin` et `=end` s'emboîtent mais seulement dans le sens où la paire la plus à l'extérieur fait que tout ce qui se trouve au milieu est traité comme n'étant pas du pod, même s'il y a là-dedans d'autres directives `=mot`. C'est-à dire que dès qu'un traducteur voit `=begin truc`, soit il ignorera, soit il traitera *tout* jusqu'au `=end truc` correspondant.

Séquences pod

Le troisième type de paragraphe est simplement du texte au kilomètre. C'est à dire que si un paragraphe ne commence ni par un espace, ni par un signe égal, il est pris comme un paragraphe à plat : du texte ordinaire tapé avec aussi peu de décorations que possible. Les sauts de ligne sont traités comme des espaces. C'est en grande partie au traducteur de le rendre agréable car les programmeurs ont des choses plus importantes à faire. On présume que les traducteurs appliqueront certaines heuristiques courantes — voir la section *Traducteurs et modules pod* plus loin dans ce chapitre.

Vous pouvez toutefois faire certaines choses explicitement. À l'intérieur des paragraphes ordinaires ou des directives d'en-têtes ou d'items (mais pas dans les paragraphes

« tels quels »), vous pouvez utiliser des séquences spéciales pour ajuster le formatage. Ces séquences commencent toujours par une seule lettre majuscule suivie par un signe inférieur et elles se prolongent jusqu'au signe supérieur correspondant (qui n'est pas forcément le suivant). Les séquences peuvent contenir d'autres séquences.

Voici les séquences définies par pod :

`I<`*`texte`*`>`

Texte en italique, utilisé pour mettre en relief, pour les titres de livres, les noms de bateaux et les références de pages de manuel comme « *perlpod*(1) ».

`B<`*`texte`*`>`

Texte en gras, utilisé presqu'exclusivement pour les options de la ligne de commande et parfois pour les noms de programmes.

`C<`*`texte`*`>`

Code littéral, probablement dans une fonte de largeur fixe, comme du Courrier. Ce n'est pas nécessaire pour les items dont le traducteur devrait être capable d'en déduire qu'il s'agit de code mais vous devriez mettre cette séquence de toute façon.

`S<`*`texte`*`>`

Texte avec des espaces insécables. Entoure souvent d'autres séquences.

`L<`*`nom`*`>`

Une référence croisée (un lien) vers un nom :

`L<`*`nom`*`>`

Page de manuel.

`L<`*`nom/ident`*`>`

Article dans une page de manuel.

`L<nom /"`*`sec`*`">`

Section dans une autre page de manuel.

`L<"`*`sec`*`">`

Section dans cette page de manuel (les guillemets sont optionnels).

`L</"`*`sec`*`">`

Idem.

Les cinq séquences suivantes sont les mêmes que celles ci-dessus, mais la sortie sera seulement *`texte`*, en cachant l'information sur le lien, comme en HTML :

```
L<texte |nom>
L<texte |nom/ident>
L<texte |nom/"sec">
L<texte |"sec">
L<texte |/"sec">
```

Le texte ne peut pas contenir les caractères `/` et `|` et il ne devrait contenir `<` ou `>` que s'il s'agit d'une paire.

`F<chemin>`

Utilisé pour les noms de fichiers. Ceci donne traditionnellement la même présentation que `I`.

`X<entrée>`

Une entrée d'index quelconque. Comme toujours, c'est au traducteur de décider quoi faire. La spécification de pod n'impose rien à ce sujet.

`E<seq_échappement>`

Un caractère nommé, comme les séquences d'échappement en HTML :

`E<lt>`

Un < littéral (optionnel, sauf à l'intérieur d'autres séquences et lorsqu'il est précédé d'une lettre en majuscule).

`E<gt>`

Un > littéral (optionnel, sauf à l'intérieur d'autres séquences).

`E<sol>`

Un / littéral (nécessaire seulement dans `L<>`).

`E<verbar>`

Un | littéral (nécessaire seulement dans `L<>`).

`E<NNN>`

Le caractère numéro *NNN*, probablement en ISO-8859-1, mais peut-être en Unicode. Cela ne devrait rien changer, dans l'abstrait...

`E<entité>`

Une entité HTML non numérique, comme `E<Agrave>`.

`Z<>`

Un caractère de taille nulle. C'est sympathique pour préfixer des séquences qui pourrait rendre quelque chose confus. Par exemple, si vous aviez une ligne en prose ordinaire que devait commencé par un signe égal, vous devriez l'écrire ainsi :

```
Z<>=vous voyez ?
```

ou pour quelque chose contenant un « From », pour que le logiciel de mail ne mette pas un > devant :

```
Z<>From veut dire "de la part de" en anglais...
```

La plupart du temps, vous n'aurez besoin que d'un seul ensemble de signes supérieur/inférieur pour délimiter l'une de ces séquence pod. Cependant, vous voudrez parfois mettre un < ou un > à l'intérieur d'une séquence. (C'est particulièrement courant lorsqu'on utilise une séquence `C<>` pour donner une fonte de largeur fixe à un extrait de code.) Comme toujours en Perl, il y a plus d'une manière de faire. Une méthode est de simplement représenter les signes inférieur/supérieur avec une séquence `E` :

```
C<$a E<lt>=E<gt> $b>
```

Ce qui donne « `$a <=> $b` ».

Une méthode plus lisible et peut-être plus « à plat », est d'utiliser un autre ensem-

ble de délimiteurs qui ne nécessite pas que les signes inférieur/supérieur soient protégés. Des doubles chevrons (`C<< truc >>`) peuvent être employés, pourvu qu'il y ait un espace suivant immédiatement le délimiteur ouvrant et précédent immédiatement le délimiteur fermant. Par exemple, ce qui suit fonctionnera :

```
C<< $a <=> $b >>
```

Vous pouvez utiliser autant de chevrons que vous désirez pour autant que vous en avez le même nombre des deux côtés et que vous vous êtes assuré qu'un espace suive immédiatement le dernier `<` du côté gauche et qu'un espace précède immédiatement le premier `>` du côté droit. Ainsi, ce qui suit fonctionnera également :

```
C<<< $a <=> $b >>>
C<<<< $a <=> $b >>>>
```

Tous ces exemples finissent par donner `$a <=> $b` dans une fonte de largeur fixe.

L'espace supplémentaire à l'intérieur de chaque extrémité disparaît, vous devriez donc laisser un espace à l'extérieur si vous en voulez un. De même, les deux caractères espaces supplémentaires à l'intérieur ne se chevauchent pas, donc, si la première chose citée est `>>`, ce n'est pas pris en tant que délimiteur fermant :

```
L'opérateur C<< >> >> de décalage vers la droite.
```

Ce qui donne « L'opérateur >> de décalage vers la droite ».

Remarquez que les séquences pod s'imbriquent *bien*. Cela signifie que vous pouvez écrire « `La I<Santa MarE<iacute>a> a pris le large depuis longtemps` » pour produire « La *Santa María* a pris le large depuis longtemps » ou « `B<touch> S<B<t> I<time> > I<fichier>` » pour produire « **touch -t *time* *fichier*** », et vous attendre à ce que ça marche correctement.

Traducteurs et modules pod

Perl est livré avec plusieurs traducteurs de pod qui convertissent les documents en pod (ou le pod inclus dans d'autres types de documents) vers divers formats. Tous devraient travailler purement sur 8 bits.

pod2text

Convertit du pod en texte. Normalement, ce texte sera en ASCII sur 7 bits mais il sera sur 8 bits s'il a reçu une entrée sur 8 bits, ou spécifiquement de l'ISO-8859-1 (ou de l'Unicode) si vous utilisez des séquence telles que `LE<uacute>thien` pour *Lúthien* ou `EE<auml>rendil` pour *Eärendil*.

Si vous avez un fichier contenant du pod, la manière la plus facile (mais peut-être pas la plus jolie) de ne visualiser que le pod formaté serait :

```
% pod2text Fichier.pm | more
```

Mais, encore une fois, le pod est supposé être lisible par un humain sans formatage.

pod2man

Convertit du pod vers le format des pages de manuel d'Unix, qui convient pour être visualisé *via nroff* (1) ou pour créer des copies composées *via troff* (1). Par exemple :

```
% pod2man Fichier.pm | nroff -man | more
```

ou

```
% pod2man Fichier.pm | troff -man -Tps -t > page_temp.ps
% ghostview page_temp.ps
```

et pour imprimer :

```
% lpr -Ppostscript page_temp.ps
```

pod2html

Convertit le pod en HTML pour aller avec votre navigateur préféré :

```
% pod2html Fichier.pm > page_temp.html
% lynx page_temp.html
% netscape -remote "openURL(file:‘pwd‘/page_temp.html)"
```

Le dernier exemple est une astuce de *Netscape* qui ne fonctionne que si vous avez déjà un navigateur de lancé quelque part pour dire à cette incarnation de charger la page. Sinon, appelez-le exactement comme vous l'avez fait pour *lynx*.

pod2latex

Convertit du pod en LATEX.

Des traducteurs supplémentaires pour d'autres formats sont disponibles sur CPAN.

Les traducteurs font preuve de comportements par défaut différents selon le format de sortie. Par exemple, si votre pod contient un paragraphe en prose disant :

```
Il s'agit ici d'une $variable
```

alors *pod2html* changera ça en :

```
Il s'agit ici d'une <STRONG>$variable</STRONG>
```

alors que *pod2text* le laissera sans décoration, puisque le dollar devrait suffire pour que cela soit lisible.

Vous devriez écrire votre pod en restant le plus proche possible du texte « à plat », en vous passant le plus possible de balises explicites. C'est à chaque traducteur de décider comment créer des signes de protection appariés, comment remplir et ajuster le texte, comment trouver une fonte plus petite pour les mots entièrement en majuscules, etc. Puisqu'ils ont été écrits pour traiter la documentation en Perl, la plupart des traducteurs[1] devraient également reconnaître des éléments sans décoration comme ceux-ci et les présenter de manière appropriée :

- `HANDLE_FICHIER`
- `$scalaire`
- `@tableau`
- `fonction()`
- `page_man(3r)`
- `quelquun@quelquepart.com`
- `http://truc.com/`

Perl apporte également plusieurs modules standard pour analyser et convertir du pod,

1. Si vous concevez un traducteur de pod générique, et non pour du code Perl, vos critères peuvent différer.

comprenant `Pod::Checker` (et l'utilitaire associé *podchecker*) pour vérifier la syntaxe des documents pod, `Pod::Find` pour trouver des documents pod dans les arborescences de répertoires et `Pod::Parser` pour créer vos propres utilitaires pod.

Remarquez que les traducteurs de pod devraient seulement regarder les paragraphes commençant par une directive pod (cela facilite l'analyse), puisqu'en fait le compilateur sait rechercher les séquences d'échappement pod même au milieu d'un paragraphe. Cela signifie que le secret suivant sera ignoré à la fois par le compilateur et par les traducteurs.

```
$a=3;
=secret
warn "Ni du POD, ni du CODE !?"
=cut back
print "reçu $a\n";
```

Vous ne devriez probablement pas vous reposer sur le fait que le `warn` restera à jamais sorti du pod. Les traducteurs de pod ne sont pas tous sages à cet égard et le compilateur pourrait devenir moins sélectif un jour ou l'autre.

Écrire vos propres outils pod

Pod a été conçu tout d'abord et surtout pour être facile à écrire. Un corollaire de la simplicité de pod est qu'elle permet d'écrire soi-même des outils simples pour traiter du pod. Si vous recherchez des directives pod, vous n'avez qu'à positionner le séparateur d'enregistrements en entrée en mode paragraphe (peut-être avec l'option `-00`) et n'accorder de l'attention qu'à ce qui ressemble à du pod.

Voici par exemple un simple programme afficheur de plan en pod :

```
#!/usr/bin/perl -l00n
# planpod - affiche le plan en pod
next unless /^=head/;
s/^=head(\d)\s+/ ” x ($1 * 4 - 4)/e;
print $_, "\n";
```

Si vous le lancez sur le présent chapitre de ce livre, vous obtiendrez quelque chose comme cela :

```
Plain Old Documentation
    Pod en quelques mots
        Paragraphes « tels quels »
        Directives pod
        Séquences pod
    Traducteurs et modules pod
    Écrire vos propres outils pod
    Pièges du pod
    Documenter vos programmes perl
```

Cet afficheur de plan en pod n'a pas vraiment fait attention au fait que le bloc pod était valide ou non. Puisque ce qui est en pod et ce qui ne l'est pas peuvent s'entremêler dans le même fichier, cela n'a pas toujours de sens de lancer des outils génériques pour rechercher et analyser le fichier complet. Mais ce n'est pas un problème, étant donné qu'il est si facile d'écrire des outils pour le pod. Voici un outil qui *est* attentif aux différences

entre le pod et ce qui n'est pas du pod, puis n'affiche que le pod :

```
#!/usr/bin/perl -00
# catpod - n'affiche que ce qui est en pod
while (<>) {
    if (!$enpod) { $enpod = /^=/;           }
    if ($enpod)  { $enpod = !/=cut/; print; }
} continue {
    if (eof)     { close ARGV; $enpod = ''; }
}
```

Vous pourriez utiliser ce programme sur un autre programme ou un module en Perl, puis mettre un pipe sur la sortie vers un autre outil. Par exemple, si vous posséder le programme *wc*(1)[2] pour compter le nombre de lignes, de mots et de caractères, vous pourriez l'approvisionner avec la sortie de *catpod* pour qu'il ne considère dans son comptage que le pod :

```
% catpod MonModule.pm | wc
```

Il existe beaucoup d'endroits où le pod vous permet d'écrire des outils rudimentaires de façon simplissime en utilisant directement du Perl brut. Maintenant que vous avez *catpod* à utiliser comme composant, voici un autre outil pour ne montrer que le code indenté :

```
#!/usr/bin/perl -n00
# podlit - affiche les blocs littéraux indentés
# à partir de pod en entrée
print if /^\s/;
```

Que pourriez-vous faire avec ça ? Et bien, vous pourriez premièrement vouloir vérifier le code dans le document avec *perl -wc*. Ou peut-être voudriez-vous une fonctionnalité de *grep*(1)[3] qui ne regarde que les exemples de code :

```
% catpod MonModule.pm | podlit | grep nom_fonc
```

Cette philosophie, selon laquelle les outils et les filtres sont des parties interchangeables (et testables unitairement), est une approche tout bonnement simple et puissante pour concevoir des composants logiciels réutilisables. C'est une forme de paresse de ne construire qu'une solution minimale pour accomplir le travail aujourd'hui — du moins, pour certains types de travaux.

Cependant, pour d'autres tâches, ceci peut même s'avérer contre-productif. Parfois, il y a plus de travail à écrire un outil à partir de rien et parfois moins. Pour ceux que nous vous avons montrés précédemment, les prouesses naturelles de Perl en ce qui concerne le traitement du texte sont tout indiquées pour appliquer « la force brute ». Mais tout ne fonctionne pas de cette façon. En jouant avec le pod, vous pourriez remarquer que bien que ses directives soient simples à analyser, ces séquences peuvent s'avérer plus risquées. Bien que des traducteurs, hum, pas très corrects, ne s'adaptent pas à ceci, les séquences peuvent s'imbriquer dans d'autres séquences et peuvent avoir des délimiteurs de taille variable.

---

2. Et si ce n'est pas le cas, allez cherchez la version des Perl Power Tools dans le répertoire *scripts* de CPAN.

3. Et si vous ne possédez pas grep, voir la précédente note de bas de page.

Au lieu de vous mettre à coder vous-même ce code d'analyse, les paresseux chercheront une autre solution. Le module standard `Pod::Parser` remplit ce contrat. Il est tout particulièrement utile pour les tâches difficiles, comme celles qui nécessitent une réelle analyse des morceaux internes des paragraphes, une conversion vers d'autres formats de sortie, et ainsi de suite. Il est plus facile d'utiliser le module pour les cas compliqués car la quantité de code à laquelle vous aboutirez est plus petite. C'est également mieux car l'analyse épineuse est déjà réalisée pour vous. Il s'agit tout à fait du même principe que l'emploi de *catpod* dans une chaîne de pipes.

Le module `Pod::Parser` adopte une approche intéressante dans son travail. C'est un module orienté objet d'une facture différente de la plupart de ceux que vous avez vus dans ce livre. Son objectif premier n'est pas tant de fournir des objets à manipuler directement que d'apporter une classe de base à partir de laquelle d'autres classes peuvent être construites.

Vous créez votre propre classe et héritez de `Pod::Parser`. Puis vous déclarez des sous-programmes servant de méthodes de rappel (*callbacks*) que l'analyseur de votre classe parente invoquera. Il s'agit d'une façon de programmer différente des programmes procéduraux donnés précédemment. Dans un sens, il s'agit plus d'un style de programmation déclaratif car, pour accomplir le travail, vous enregistrez simplement des fonctions et laissez d'autres entités les invoquer pour vous. La logique ennuyeuse du programme est gérée ailleurs. Vous ne faites que donner des morceaux « plug-and-play ».

Voici une réécriture du programme original *catpod* donné auparavant mais, cette fois-ci, elle utilise le module `Pod::Parser` pour créer votre propre sous-classe :

```
#!/usr/bin/perl
# catpod2, classe et programme

package catpod_analyseur;
use Pod::Parser;
@ISA = qw(Pod::Parser);
sub command {
    my ($analyseur, $commande, $paragraphe, $num_ligne) = @_;
    my $hf_sortie = $analyseur->output_handle();
    $paragraphe .= "\n" unless substr($paragraphe, -1) eq "\n";
    $paragraphe .= "\n" unless substr($paragraphe, -2) eq "\n\n";
    print $hf_sortie "=$commande $paragraphe";
}

sub verbatim {
    my ($analyseur, $paragraphe, $num_ligne) = @_;
    my $hf_sortie = $analyseur->output_handle();
    print $hf_sortie $paragraphe;
}

sub textblock {
    my ($analyseur, $paragraphe, $num_ligne) = @_;
    my $hf_sortie = $analyseur->output_handle();
    print $hf_sortie $paragraphe;
}
```

```
sub interior_sequence {
    my ($analyseur, $commande_seq, $argument_seq) = @_;
    return "$commande_seq<$argument_seq>";
}

if (!caller) {
    package main;
    my $analyseur = catpod_analyseur->new();
    unshift @ARGV, '-' unless @ARGV;
    for (@ARGV) { $analyseur->parse_from_file($_); }
}
1;
__END__

=head1 NAME
documentation décrivant le nouveau programme catpod, ici
```

Comme vous le voyez, c'est un peu plus long et plus compliqué. C'est également plus évolutif car tout ce que vous avez à faire est de brancher vos propres méthodes lorsque vous voulez que votre sous-classe agisse différemment que sa classe de base.

Le dernier morceau à la fin, où il est écrit `!caller`, vérifie si le fichier est utilisé en tant que module ou en tant que programme. S'il est employé comme programme, il n'y aura alors pas d'appelant (`caller` en anglais). Il lancera donc son propre analyseur (en utilisant la méthode `new` dont il a hérité) sur les arguments de la ligne de commande. Si aucun nom de fichier n'a été passé, il prend l'entrée standard, tout comme le faisait la version précédente.

À la suite du code du module, il y a un marqueur `__END__`, une ligne vide sans espace, puis la propre documentation en pod du programme/module. Il s'agit d'un exemple d'un fichier qui est à la fois un programme *et* un module *et* sa propre documentation. Il est probablement plusieurs autres choses également.

Pièges du pod

Pod est assez direct à apprendre et à écrire mais il est toujours possible de rater quelques petites choses :

- Il est vraiment facile d'oublier le signe supérieur final.
- Il est vraiment facile d'oublier la directive `=back` finale.
- Il est facile de mettre accidentellement une ligne vide au milieux d'une longue directive `=for` *`comment`*. Songez à utiliser `=begin`/`=end` à la place.
- Si vous faites une faute de frappe sur l'une des deux balises d'une paire `=begin`/`=end`, cela avalera tout le reste de votre fichier (en ce qui concerne le pod). Songez à utiliser `=for` à la place.
- Les traducteurs de pod exigent que les paragraphes soient séparés par des lignes complètement vides ; c'est-à-dire, par au moins deux caractères de saut de ligne consécutifs (`\n`). Si vous avez une ligne avec des espaces ou des tabulations, elle ne sera pas prise comme une ligne blanche. Ceci peut causer le traitement de deux paragraphes ou plus comme s'il n'y en avait qu'un seul.

- La signification d'un « lien » n'est pas définie par pod et c'est à chaque traducteur de décider quoi en faire. (Si vous commencez à penser que la plupart des décisions ont été reportées sur les traducteurs et pas dans pod, vous avez raison.) Les traducteurs ajouteront souvent des mots autour d'un lien `L<>`, ainsi, « `L<truc(1)>` » deviendra « la page de manuel *truc*(1) », par exemple. Vous ne devriez donc pas écrire des choses comme « `la page de manuel L<truc>` » si vous voulez que le document traduit soit raisonnablement lisible : cela aboutirait à « la page de manuel la page de manuel *truc*(1) ».

  Si vous avez besoin d'un contrôle total sur le texte utilisé dans un lien, utilisez plutôt la forme `L<montre ce texte|truc>`.

Le programme standard *podchecker* vérifie la syntaxe pod pour trouver les erreurs et les avertissements. Par exemple, il recherche les séquences pod inconnues et les lignes blanches trompeuses contenant des espaces. Il est toujours recommandé de passer votre document à travers différents traducteurs de pod et de corriger les résultats. Certains des problèmes que vous trouverez peuvent être des idiosyncrasies des traducteurs particuliers, ce sur quoi vous voudrez ou non travailler.

Et, comme toujours, *tout est sujet à modification selon les caprices d'un hacker quelconque*.

Documenter vos programmes Perl

Nous espérons que vous documentez votre code, que vous soyez ou non un *hacker quelconque*. Si c'est le cas, vous pouvez souhaiter inclure les sections suivantes dans votre pod :

`=head1 NOM`

Le nom de votre programme ou de votre module.

`=head1 SYNOPSIS`

Une description d'une ligne sur ce que fait (soi-disant) votre programme ou votre module.

`=head1 DESCRIPTION`

La masse de votre documentation. (Masse est le terme approprié dans ce contexte.)

`=head1 AUTEUR`

Qui vous êtes. (Ou un pseudonyme, si vous avez honte de votre programme.)

`=head1 BOGUES`

Ce que vous avez mal fait (et pourquoi ce n'était pas vraiment votre faute).

`=head1 VOIR AUSSI`

Où les gens peuvent trouver des informations en rapport (pour qu'ils puissent travailler sur vos bogues).

`=head1 COPYRIGHT`

Les instructions légales. Si vous voulez revendiquer un copyright explicite, vous devriez dire quelque chose comme :

```
Copyright 2013, Randy Waterhouse. Tous droits réservés?
```

Beaucoup de modules ajoutent également :

```
Ce programme est un logiciel libre. Vous pouvez le copier ou le
redistribuer sous les mêmes termes que Perl lui-même.
```

Un avertissement : si vous êtes sur le point de mettre votre pod à la fin du fichier et que vous utilisez les tokens `__END__` ou `__DATA__`, assurez-vous de mettre une ligne vide avant la première directive pod :

```
__END__

=head1 NOM

Moderne - Je suis vraiment un modèle d'un module essentiel moderne
```

Sans la ligne vide avant le `=head1`, les traducteurs de pod ignoreront le début de votre documentation (vaste, précise, cultivée).

27

Culture Perl

Ce livre fait partie de la culture Perl, nous ne pouvons donc pas espérer mettre ici tout ce que nous connaissons à propos de la culture Perl. Nous pouvons seulement attiser votre appétit avec un peu d'histoire et un peu d'art — certains diraient « de l'art mineur ». Pour une dose beaucoup plus grosse de culture Perl, voir *www.perl.org* et *www.perl.com*. (Larry garde des copies de tous ses délires (officiels) à *www.wall.org/~larry*.) Ou acoquinez-vous à d'autres programmeurs Perl. Nous ne pouvons pas vous dire quel genre de personnes ils seront — le seul trait de caractère que les programmeurs Perl ont en commun est qu'ils sont pathologiquement serviables.

L'histoire en pratique

Pour comprendre pourquoi Perl est ainsi défini (ou indéfini), il faut d'abord comprendre la raison même pour laquelle Perl existe. Sortons donc le grimoire poussiéreux...

En 1986, Larry était un programmeur système sur un projet de réseaux à grande échelle et à multiples niveaux de sécurité. Il s'occupait d'une installation composée de trois Vax et de trois Sun sur la Côte Ouest, connectés par une ligne série chiffrée à 1200 bauds à une configuration similaire sur la Côte Est. Comme son travail consistait surtout en du support (il n'était pas programmeur sur le projet, seulement le gourou système), il put exploiter ses trois vertus (paresse, impatience et orgueil) pour développer et améliorer toutes sortes d'outils intéressants — comme *rn*, *patch* et *warp*.[1]

1. C'est à peu près à ce moment que Larry revendiqua l'expression « feeping creaturism » (*N.d.T.* : jeu de mot délibéré avec « creeping featurism », ajouter des fonctionnalités rampantes, signifiant que le système ou le programme en question est devenu une créature difforme à cause des bidouilles qu'on lui a successivement ajoutées). Ceci dans une tentative désespérée de justifier, sur la base d'une nécessité biologique, l'habitude d'ajouter « rien qu'une dernière fonctionnalité ». Après tout, si la vie est simplement trop compliquée, pourquoi pas les programmes ? Surtout les programmes comme *rn* qui devraient vraiment être traités comme des projets avancés d'Intelligence Artificielle afin qu'ils puissent lire les news pour vous. Évidemment, certains trouvent le programme *patch* déjà *trop* intelligent.

Un jour, alors que Larry finissait de découper *rn* en petits morceaux, le laissant éparpillé sur le sol de son répertoire, le Grand Patron vint à lui et dit : « Larry, il nous faut un système de gestion de configurations et de contrôle pour les six Vax et les six Sun. Dans un mois. Exécution ! »

Larry, qui n'a jamais été un fainéant, se demanda quel était le meilleur moyen d'obtenir un système de gestion de configurations transcontinental, sans devoir partir de zéro, qui permettrait de visualiser les rapports d'incidents sur les deux côtes, avec validation et contrôle. La réponse tenait en un mot : B-news.[2]

Larry commença par installer les news sur ces machines et ajouta deux commandes de contrôle : une commande « append » pour enrichir un article existant et une commande « synchronize » pour que les numéros d'articles restent les mêmes sur les deux côtes. La gestion du contrôle passerait par RCS (Revision Control System) et les validations et les soumissions se feraient par les news et *rn*. Jusqu'ici, tout allait bien.

Puis, le Grand Patron lui demanda de fournir des rapports. Les news étaient maintenues dans des fichiers séparés sur une machine maître, avec un grand nombre de références croisées entre les fichiers. La première pensée de Larry fut : « Utilisons *awk*. » Malheureusement, le *awk* d'alors ne savait pas gérer l'ouverture et la fermeture de plusieurs fichiers en fonction des informations de ces fichiers. Larry ne voulait pas programmer un outil spécifique. C'est ainsi qu'un nouveau langage naquit.

À l'origine, il ne s'appelait pas Perl. Larry envisagea un certain nombre de noms avec ses collègues et son entourage (Dan Faigin, qui écrivit cette histoire et Mark Biggar, son beau-frère, qui l'assista énormément lors de la conception initiale). Larry considéra et rejeta en fait tous les mots de 3 ou 4 lettres du dictionnaire. L'un des premiers avait été « Gloria », inspiré de sa chère et tendre (et épouse). Il se dit rapidement que cela provoquerait trop de confusions d'ordre domestique.

Le nom devint alors « Pearl », qui muta en notre « Perl » actuel, en partie parce que Larry trouva une référence à un autre langage nommé « PEARL » mais surtout parce qu'il est trop paresseux pour taper cinq lettres d'affilée à chaque fois. Évidemment, afin que Perl puisse être utilisé comme l'un de ces mots de quatre lettres chers aux anglophones. (Notez cependant les vestiges du nom initial dans le développement de l'acronyme : « Practical Extraction *And* Report Language »).

Le Perl des origines avait beaucoup moins de fonctionnalités que celui d'aujourd'hui. La recherche de correspondances et les handles de fichiers étaient là, les scalaires aussi, de même que les formats, mais il ne comportait que très peu de fonctions, pas de tableaux associatifs et juste une implémentation restreinte des expressions régulières, empruntée à *rn*. Le manuel ne contenait que 15 pages. Mais Perl était plus rapide que *sed* et *awk* et commença à être utilisé pour d'autres applications du projet.

Mais Larry était demandé par ailleurs. Un autre Grand Patron arriva un jour et dit : « Larry, il faut assister la R&D » . Et Larry fut d'accord. Il prit Perl avec lui et découvrit qu'il se transformait en un bon outil d'administration système. Il emprunta à Henry Spencer son superbe paquetage d'expressions régulières et le charcuta en quelque chose auquel Henry préférerait ne pas penser à table. Puis Larry ajouta la plupart des fonctionnalités qui lui semblaient bonnes et quelques autres que d'autres personnes dési-

2. C'est-à-dire la seconde implémentation du logiciel de transport de Usenet.

raient. Il le diffusa sur le réseau.[3] Le reste, comme on dit, appartient à l'histoire.[4]

L'histoire s'est déroulée comme ceci : Perl 1.0 sort le 18 décembre 1987 ; certains prennent toujours au sérieux l'anniversaire de Perl. Perl 2.0 suit en juin 1988 et Randal Schwartz crée la signature légendaire « Just Another Perl Hacker » (juste un autre bidouilleur Perl). En 1989, Tom Christiansen présente le premier tutoriel public au Usenix de Baltimore. Avec Perl 3.0 en octobre 1989, le langage sort et distribué pour la première fois dans les termes de la Licence Publique Générale de GNU.

En mars 1990, Larry écrit le premier poème en Perl (voir la section suivante). Puis lui et Randal écrivent la première édition de ce livre, le « Pink Camel » (le chameau rose ou fushia) ; il est publié en début d'année 1991. Perl 4.0 sort en même temps ; il comprend la Licence Artistique ainsi que la LPG.

L'inauguration du très attendu Perl 5 se déroule en octobre 1994. C'est une réécriture complète de Perl, comprenant des objets et des modules. L'avènement de Perl 5 fait même la couverture du journal *The Economist*. En 1995, CPAN est officiellement présenté à la communauté Perl. Jon Orwant commence à publier *The Perl Journal* en 1996. Après une longue gestation, la seconde édition de ce livre, le « Blue Camel » (le chameau bleu) paraît cet automne. La première conférence Perl d'O'Reilly (TPC) se déroule à San José, en Californie, durant l'été 1997. Des événements marquants arrivent maintenant presque quotidiennement, donc, pour la suite de l'histoire, regardez la Perl Timeline sur CPAST, le Comprehensive Perl Arcana Society Tapestry (*N.d.T.* : la ligne du temps sur la tapisserie de la société des arcanes détaillées de Perl) sur *history.perl.org*.

Poésie en Perl

La contrefaçon ci-contre apparut sur Usenet le 1er avril 1990. Elle est présentée sans commentaires, essentiellement afin de montrer quel degré d'abjection peuvent atteindre les métaphores d'un langage de programmation typique. Larry est particulièrement soulagé que « *Black Perl* », écrit à l'origine pour Perl 3, ne compile plus sous Perl 5.

Le, euh, corpus de Larry a été heureusement supplanté par la Poétesse Perl régnante, Sharon Hopkins. Elle a écrit un certain nombre de poèmes Perl, ainsi qu'un article sur la poésie en Perl, qu'elle a présenté à la Conférence Technique Usenix de l'hiver 1992, intitulé « Camel and Needles: Computer Poetry Meets the Perl Programming Language » (Le chameau et les aiguilles : quand la poésie informatique rencontre le langage de programmation Perl). (L'article est disponible sous *misc/poetry.ps* sur CPAN.) Non contente d'être la plus prolifique des poètesses Perl, Sharon est également la plus largement publiée, le poème suivant étant sorti dans l'*Economist* et le *Guardian* :

---

3. Ce qui est encore plus extraordinaire, c'est qu'il continua à le diffuser alors qu'il était embauché chez Jet Propulsion Lab, puis NetLabs, puis Seagate. Maintenant, d'autres personnes font la plupart du vrai boulot et Larry prétend travailler pour O'Reilly & Associates (une petite société qui publie des opuscules sur les ordinateurs et autres balivernes).

4. Et voici, pour ainsi dire, un addendum à l'histoire. Aux débuts de Perl, *rn* venait d'être mis en pièces avant une réécriture complète. Depuis qu'il a commencé à travailler sur Perl, Larry n'a plus touché à *rn*. Il reste en pièces détachées. Parfois, Larry menace de réécrire *rn* en Perl mais jamais sérieusement.

```
#!/usr/bin/perl

APPEAL:

listen (please, please);

open yourself, wide;
    join (you, me),
connect (us,together),

tell me.

do something if distressed;

    @dawn, dance;
    @evening, sing;
    read (books,$poems,stories) until peaceful;
    study if able;

    write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

        do*not*die (like this)
        if sin abounds;

keys (hidden), open (locks, doors), tell secrets;
do not, I-beg-you, close them, yet.

                        accept (yourself, changes),
                        bind (grief, despair);

require truth, goodness if-you-will, each moment;

select (always), length(of-days)

# listen (a perl poem)
# Sharon Hopkins
# rev. June 19, 1995
```

Poésie en Perl

Article 970 of comp.lang.perl:
Path: jpl-devvax!jpl-dexxav!lwall
From: lwall@jpl-dexxav.JPL.NASA.GOV (Larry Wall)
Newsgroups: news.groups,rec.arts.poems,comp.lang.perl
Subject: CALL FOR DISCUSSION: comp.lang.perl.poems
Message-ID: <0401@jpl-devvax.JPL.NASA.GOV>
Date: 1 Apr 90 00:00:00 GMT
Reply-To: lwall@jpl-devvax.JPL.NASA.GOV (Larry Wall)
Organization: Jet Prepulsion Laboratory, Pasadena, CA
Lines: 61

Mon attention a été attirée par l'urgente nécessité qu'éprouvent les gens à exprimer simultanément leurs natures émotionnelle et technique. Plusieurs personnes m'ont envoyé des messages qui ne correspondent à aucun groupe de news. C'est peut-être parce que j'ai récemment posté simultanément dans comp.lang.perl et dans rec.arts.poems que des gens écrivent des poèmes en Perl, et qu'ils me demandent où les poster. En voici un échantillon :

D'un étudiant en faculté (dans ses dernières semaines), le haiku suivant :

```
study, write, study,
do review (each word) if time.
close book. sleep? what's that?
```

Et quelqu'un de Fort Lauerdale écrit :

```
sleep, close together,
sort of sin each spring & wait;
50% die
```

Une personne désirant rester anonyme a écrit l'exemple suivant de « Black Perl » (le poète de Pearl en aurait été choqué, sans aucun doute.)

```
BEFOREHAND: close door, each window & exit; wait until time.
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
    kill spiders, pop them, chop, split, kill them.
        unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
    values aside, each one;
        die sheep! die to reverse the system
        you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
    wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
    select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
    wait until next year, next decade;
        sleep, sleep, die yourself,
        die at last
```

Je l'ai essayé et il compile effectivement en Perl. Il ne semble cependant pas faire grand chose. Ce qui à mon avis est probablement heureux... Je propose donc la création de comp.lang.perl.poems pour servir de lieu d'accueil pour ces articles, afin que nous ne polluions pas les groupes de news sur perl ou les poèmes avec des choses qui ne seraient d'intérêt à aucun d'entre eux. Ou bien, nous pouvons créer rec.arts.poems.perl pour ceux qui se contentent de compiler, sans rien faire d'utile (il existe des précédents dans rec.arts.poems, après tout). Ensuite, créons également comp.lang.perl.poems pour ceux qui font réellement quelque chose, comme ce haiku personnel :

```
print STDOUT q Just another Perl hacker,
unless $spring
```

Larry Wall lwall@jpl-devvax.jpl.nasa.gov

V

Références

28

Noms spéciaux

Ce chapitre traite des variables qui ont une signification spéciale pour Perl. La plupart des noms à ponctuation ont des mnémoniques raisonnables ou des homologues dans les shells (ou les deux). Néanmoins, s'il faut utiliser les noms longs de variables, il suffit d'écrire :

```
use English;
```

au début du programme. Ceci crée des alias longs pour les noms courts dans le paquetage courant. Certain d'entre eux ont même des noms moyens, généralement empruntés à *awk*. La plupart des gens continuent d'utiliser les noms courts, du moins pour les variables les plus courantes. Au long de cet ouvrage, nous employons uniformément les noms courts, mais nous mentionnerons aussi les noms longs (entre parenthèses) pour que vous puissiez facilement les retrouver dans ce chapitre.

La sémantique de ces variables est quelque peu magique (pour créer vos propres sorts, voir le chapitre 24, *Variables liées*). Certaines de ces variables sont en lecture seule. Si l'on essaie d'y assigner une valeur, une exception sera lancée.

Dans ce qui suit, nous donnerons tout d'abord une liste des variables et fonctions auxquelles Perl attribue une signification spéciale, classées par type ; ainsi vous pourrez retrouver les variables dont vous n'êtes plus tout à fait sûrs du nom exact. Ensuite nous détaillerons toutes les variables par ordre alphabétique, avec leur nom approprié (ou leur nom le moins inapproprié).

Noms spéciaux classés par type

Nous utilisons improprement le mot « type », les sections suivantes regroupent davantage les variables selon leur contexte, c'est-à-dire depuis l'endroit où elles sont visibles.

Variables spéciales des expressions régulières

Les variables spéciales suivantes sont associées à la recherche de motifs. Elles sont visibles tout au long du contexte dans lequel la recherche a lieu (hormis $*, qui est dépré-

ciée). Autrement dit, elles se comportent comme si elles étaient déclarées avec `local`, vous n'avez donc pas besoin de les déclarer ainsi vous-même. Voir le chapitre 5, *Reconnaissance de motifs*.

```
$*
$chiffres
@+ (@LAST_MATCH_END)
@- (@LAST_MATCH_START)
$+ ($LAST_PAREN_MATCH)
$^R ($LAST_REGEXP_CODE_RESULT)
$& ($MATCH)
$' ($POSTMATCH)
$` ($PREMATCH)
```

Variables spéciales des handles de fichiers

Ces variables spéciales n'ont jamais besoin d'être mentionnées dans un `local` car elles se réfèrent toujours à des valeurs appartenant au handle de sortie actuellement sélectionné — chaque handle de fichier garde son propre jeu de valeurs. Quand on sélectionne un autre handle de fichier, l'ancien garde ses valeurs et les variables reflètent alors les valeurs du nouveau handle de fichier. Voir aussi le module `FileHandle` dans le chapitre 32, *Modules standards*.

```
$| ($AUTOFLUSH, $OUTPUT_AUTOFLUSH)
$- ($FORMAT_LINES_LEFT)
$= ($FORMAT_LINES_PER_PAGE)
$~ ($FORMAT_NAME)
$% ($FORMAT_PAGE_NUMBER)
$^ ($FORMAT_TO_NAME)
```

Variables spéciales par paquetage

Ces variables spéciales existent séparément dans chaque paquetage. Il n'est pas besoin de les localiser, puisque `sort` le fait automatiquement pour `$a` et `$b`, et qu'il est probablement préférable de laisser les autres se débrouiller toutes seules (cependant si vous employez `use strict`, vous devrez les déclarer avec `our`).

```
$a
$b
@EXPORT
@EXPORT_OK
%EXPORT_TAGS
%FIELDS
@ISA
%OVERLOAD
$VERSION
```

Variables spéciales pour le programme entier

Ces variables sont vraiment globales au sens exact du terme — elles signifient la même chose dans chaque paquetage, car elles sont toutes localisées de force dans le paquetage

`main` lorsqu'elles sont non qualifiées (sauf pour `@F`, qui est spéciale dans `main`, mais pas forcée). Si vous voulez une copie temporaire de l'une d'entre elles, vous devez la localiser dans le contexte dynamique courant.

```
%ENV
%INC
%SIG
%!
%^H

@_
@ARGV
@F
@INC

$_ ($ARG)
$0 ($PROGRAM_NAME)
$ARGV

$! ($ERRNO, $OS_ERROR)
$" ($LIST_SEPARATOR)
$#
$$ ($PID, $PROCESS_ID)
$( ($GID, $REAL_GROUP_ID)
$) ($EGID, $EFFECTIVE_GROUP_ID)
$, ($OFS, $OUTPUT_FIELD_SEPARATOR)
$. ($NR, $INPUT_LINE_NUMBER)
$/ ($RS, $INPUT_RECORD_SEPARATOR)
$: ($FORMAT_LINE_BREAK_CHARACTERS)
$; ($SUBSEP, $SUBSCRIPT_SEPARATOR)
$< (UID, $REAL_USER_ID)
$> (EUID, $EFFECTIVE_USER_ID)
$? ($CHILD_ERROR)
$@ ($EVAL_ERROR)
$[
$\ ($ORS, $OUTPUT_RECORD_SEPARATOR)
$] ($OLD_PERL_VERSION)
$^A ($ACCUMULATOR)
$^C ($COMPILING)
$^D ($DEBUGGING)
$^E ($EXTENDED_OS_ERROR)
$^F ($SYSTEM_FD_MAX)
$^H
$^I ($INPLACE_EDIT)
$^L ($FORMA_FORMFEED)
$^M
$^O ($OSNAME)
$^P ($PERLDB)
$^R ($LAST_REGEXP_DOC_RESULT)
$^S ($EXCEPTION_BEING_CAUGHT)
$^T ($BASETIME)
$^V ($PERL_VERSION)
$^W ($WARNING)
${^WARNING_BITS}
${^WIDE_SYSTEM_CALLS}
$^X ($EXECUTABLE_NAME)
```

Handles de fichiers spéciaux par paquetage

Sauf pour `DATA`, qui est toujours particulier à un paquetage, les handles de fichiers suivants sont toujours considérés comme étant dans `main` lorsqu'ils ne sont pas pleinement qualifiés dans un autre paquetage.

```
_ (souligné)
ARGV
ARGVOUT
DATA
STDIN
STDOUT
STDERR
```

Fonctions spéciales par paquetage

Les noms des routines suivantes ont une signification spéciale pour Perl. Elles sont toujours appelées implicitement suite à un événement, comme l'accès à une variable liée ou l'appel une fonction indéfinie. Nous ne les décrirons pas dans ce chapitre puisqu'elles sont largement couvertes par ailleurs dans cet ouvrage.

Fonction indéfinie appelant une interception (voir le chapitre 10, *Paquetages*) :

```
AUTOLOAD
```

Achèvement des objets moribonds (voir le chapitre 12, *Objets*) :

```
DESTROY
```

Objets d'exception (voir `die` dans le chapitre suivant) :

```
PROPAGATE
```

Fonctions d'auto-initialisation et d'autonettoyage (voir le chapitre 18, *Compilation*) :

```
BEGIN, CHECK, INIT, END
```

Méthodes de liaison (voir le chapitre 14) :

```
BINMODE, CLEAR, CLOSE, DELETE, EOF, EXISTS, EXTEND, FETCH, FETCHIZE, FILENO,
FIRSTKEY, GETC, NEXTKEY, OPEN, POP, PRINT, PRINTF, PUSH, READ, READLINE,
SEEK, SHIFT, SPLICE, STORE, STORESIZE, TELL, TIEARRAY, TIEHANDLE, TIEHASH,
TIESCALAR, UNSHIFT, WRITE
```

Variables spéciales dans l'ordre alphabétique

Nous avons classé alphabétiquement ces entrées selon le nom long des variables. Si vous ne connaissez pas le nom long d'une variable, vous pouvez le trouver dans la section précédente. (Les variables sans nom alphabétique sont classées au début.)

Pour éviter les répétitions, chaque description de variable débute avec une, ou plusieurs, des annotations suivantes.

Annotation	Signification
XXX	Dépréciée, *ne pas utiliser* pour quelque chose de nouveau.
NON	Non Officiellement Nommée (pour usage interne seulement).
GLO	Réellement globale, partagée par tous les paquetages.
PKG	Globale au paquetage ; chaque paquetage peut redéfinir sa propre variable.
AHF	Attribut de handle de fichier ; un attribut par objet d'entrée/sortie.
DYN	Automatiquement dans un contexte dynamique (entraîne GLO).
LEX	Dans un contexte lexicographique à la compilation.
LEC	En lecture seule ; lève une exception si vous tentez de modifier.

Quand plus d'un symbole ou plus d'un nom de variable est mentionné, seul le nom court est disponible par défaut. En utilisant le module `English`, les synonymes plus longs sont disponibles dans le paquetage courant et uniquement dans le paquetage cou-

rant, même si la variable est marquée avec [GLO].

Les entrées de la forme *méthode HANDLE EXPR* montrent l'interface orientée objet des variables pourvues par `FileHandle` et les différents module `IO::`. (Si vous préférez vous pouvez aussi utiliser la notation *HANDLE->méthode(EXPR)*). Ceci vous permet d'éviter d'appeler `select` pour changer le handle de sortie par défaut avant d'examiner ou de modifier la variable en question. Chacune de ces méthodes renvoie l'ancienne valeur de l'attribut du `FileHandle` ; l'attribut prend une nouvelle valeur si l'on positionne l'argument *EXPR*. Si cet argument n'est pas fourni, la plupart des méthodes ne modifient pas la valeur courante, excepté pour `autoflush`, qui suppose que l'argument vaut 1, uniquement pour se distinguer.

`_` *(souligné)*

[GLO] Il s'agit du handle de fichier spécial utilisé pour garder en cache les informations obtenues lors du dernier `stat`, `lstat` ou du dernier opérateur de test de fichier (comme `-w $file` ou `-d $file`) ayant réussi.

`$`*chiffres*

[DYN, LEC] Les variables numérotées `$1`, `$2`, etc. (aussi élevé que vous le désirez)[1] contiennent le texte trouvé par le jeu de parenthèses correspondant dans la dernière recherche de motifs du contexte dynamique actif à cet instant. (Mnémonique : comme `\chiffres`).

`$[`

[XXX, LEX] L'index du premier élément d'un tableau et du premier caractère d'une sous-chaîne. 0 par défaut, mais nous utilisons cette variable positionnée à 1 pour que Perl se comporte mieux comme *awk* (ou FORTRAN) au moment d'indicer et d'évaluer les fonctions `index` et `substr`. Parce qu'on a trouvé cette variable dangereuse, l'affectation de $[est maintenant traité comme une directive de compilation dans un contexte lexical et ne peut pas influencer le comportement de tout autre fichier. (Mnémonique : `[` précède les indices).

`$#`

[XXX, GLO] N'utilisez pas cette variable ; préférez `printf`. `$#` contient le format de sortie des nombres imprimés, dans une tentative peu convaincante d'émulation de la variable `OFMT` de *awk*. (Mnémonique : # en anglais signifie numéro, mais si vous êtes pointilleux, mieux vaut oublier tout ceci afin de ne pas trop parasiter votre programme et éviter ainsi un sermon).

`$*`

[XXX, GLO] Et bien, trois variables dépréciées à la suite ! Celle-ci peut (mais ne doit pas) être positionnée à vrai pour que Perl assume un `/m` sur chaque recherche de motif qui n'a pas explicitement de `/s`. (Mnémonique : * correspond à de multiples quantités).

1. Bien que beaucoup de moteur d'expressions régulières ne supportent que jusqu'à neuf références arrières, Perl ne possède pas de telle limite ; ainsi si vous écrivez `$768`, Perl n'en aura cure, bien que ceux qui maintiennent votre code puissent tiquer si vous employez autant de parenthèses dans vos expressions régulières.

`$a`

[PKG] Cette variable est utilisée par la fonction `sort` pour contenir le premier élément de chaque paire de valeurs à comparer (`$b` est le second élément de chaque paire). Le paquetage pour `$a` est le même que celui où l'opérateur `sort` a été compilé, qui n'est pas nécessairement celui dans lequel la fonction de comparaison a été compilée. Cette variable est implicitement localisée dans le bloc de comparaison de `sort`. Comme elle est globale, elle est exempte des messages de protestation de `use strict`. Puisque c'est un alias pour la valeur du tableau concerné, on pourrait penser qu'on peut la modifier, n'en faites rien. Voir la fonction `sort`.

`$ACCUMULATOR`
`$^A`

[GLO] La valeur courante de l'accumulateur de `write` pour les lignes de `format`. Un format contient contient des commandes `formline` qui émettent leur résultat dans `$^A`. Après avoir appelé son format, `write` affiche le contenu de `$^A` et le vide. On ne voit donc jamais le contenu de `$^A` à moins d'appeler `formline` soi-même pour l'examiner. Voir la fonction `formline`.

`$ARG`
`$_`

[GLO] L'espace d'entrée et de recherche de motifs par défaut. Ces paires sont équivalentes :

```
while (<>) {...} # n'est équivalent qu'avec un while sans décorations
while (defined($_ = <>)) {...}

chomp
chomp($_)

/^Objet:/
$_ =~ /^Objet:/

tr/a-z/A-Z/
$_ = ~ tr/a-z/A-Z/
```

Voici les endroits où Perl prend `$_` si l'on ne spécifie pas d'opérande aux fonctions et opérateurs :

- Des fonctions de listes comme `print` et `unlink` et des fonctions unaires comme `ord`, `pos` et `int`, ainsi que tous les tests de fichier (sauf pour `-t`, qui prend `STDIN` par défaut). Toutes les fonctions qui prennent par défaut `$_` sont repérées en tant que telles dans le chapitre 29, *Fonctions*.
- Les opérations de recherche de motifs `m//` et `s///` et les opérations de transformation `y///` et `tr///`, lorsqu'on les utilise sans l'opérateur `=~`.
- La variable d'itération dans une boucle `foreach` (même si on l'épèle `for` ou lorsqu'on l'utilise avec un modificateur d'instruction) si aucune autre variable n'est fournie.
- La variable d'itération implicite dans les fonctions `grep` et `map`. (Il n'y a aucun moyen de spécifier une variable différente pour ces fonctions.)

- L'endroit par défaut où est placé un enregistrement en entrée quand le résultat d'une opération `<HF>`, `readline` ou `glob` est testé en tant qu'unique critère d'un `while`. Cette affectation ne se produit pas en dehors d'un `while` ou si un quelconque élément additionnel est inclus dans l'expression `while`.

(Mnémonique : le souligné est l'opérande sous-jacente de certaines opérations).

`@ARG`
`@_`

[GLO] Dans un sous-programme, ce tableau contient la liste d'arguments passée à ce sous-programme. Voir le chapitre 6, *Sous-programmes*. Un `split` dans un contexte scalaire s'applique sur ce tableau, mais cet usage est déprécié.

`ARGV`

[GLO] Le handle de fichier spécial qui parcourt les noms de fichiers de la ligne de commande et contenus dans `@ARGV`. Habituellement, on l'écrit comme le handle de fichier nul dans l'opérateur d'angle `<>`.

`$ARGV`

[GLO] Contient le nom du fichier courant lorsqu'on lit depuis le handle de fichier `ARGV` en utilisant les opérateurs `<>` ou `readline`.

`@ARGV`

[GLO] Le tableau contenant les arguments de la ligne de commande prévus pour le script. Remarquez que `$#ARGV` est généralement le nombre d'arguments moins un, puisque `$ARGV[0]` est le premier argument et non le nom de la commande ; utilisez `scalar @ARGV` pour le nombre d'arguments du programme. Voir `$0` pour le nom du programme.

`ARGVOUT`

[GLO] Ce handle de fichier spécial est utilisé lorsqu'on traite le handle de fichier `ARGV` avec l'option `-i` ou la variable `$^I`. Voir l'option `-i` dans le chapitre 19, *L'interface de ligne de commande*.

`$b`

[PKG] Cette variable, sœur de `$a`, est utilisée dans les comparaisons avec `sort`. Voir `$a` et la fonction `sort` pour les détails.

`$BASETIME`
`$^T`

[GLO] Le moment où le script a commencé à tourner, en secondes depuis l'origine (le début de 1970, pour les systèmes Unix). Les valeurs renvoyées par les tests de fichier `-M`, `-A` et `-C` sont basées sur ce moment.

`$CHILD_ERROR`
`$?`

[GLO] Le statut renvoyé par la dernière fermeture de tube, la dernière commande en apostrophe inverse (`‘‘`) ou les fonctions `wait`, `waitpid` ou `system`. Remarquez qu'il ne s'agit pas simplement du code retour, mais le mot entier de 16-bits du statut renvoyé par les appels système sous-jacents *wait(2)* ou *waitpid(2)* (ou leurs équivalents). Ainsi, la valeur de retour du sous-programme se trouve dans l'octet de

poids fort, c'est-à-dire, `$? >> 8` ; dans l'octet de poids faible, `$? & 127` donne quel signal (s'il y en a un) a tué le processus, alors que `$? & 128` rapporte si sa disparition a produit un *core dump*. (Mnémonique : similaire à `$?` dans *sh* et sa progéniture).

À l'intérieur d'un bloc `END`, `$?` contient la valeur qui va être donnée à `exit`. Vous pouvez modifier `$?` dans un `END` pour changer le statut retour du script.

Sous VMS, le pragma `use vmsish 'status'` fait que `$?` reflète le véritable statut de retour de VMS, au lieu de l'émulation par défaut du statut POSIX.

Si la variable `h_errno` est supportée en C, sa valeur numérique est retournée *via* `$?` si l'une des fonctions `gethost*()` échoue.

`$COMPILING`
`$^C`

[GLO] La valeur courante de l'argument interne associé à l'option **`-c`**, principalement utilisé avec **`-MO`** et l'outil *perlcc(1)* pour laisser le code modifier son comportement lorsqu'il est compilé pour la génération de code. Par exemple, vous pouvez avoir envie d'exécuter `AUTOLOAD` pendant la phase de compilation, au lieu d'utiliser le chargement différé habituel, de sorte que le code sera généré immédiatement. Voir le chapitre 18.

`DATA`

[PKG] Ce handle de fichier spécial se réfère à tout ce qui suit soit le token `__END__` ou le token `__DATA__` dans le fichier courant. Le token `__END__` ouvre toujours le handle de fichier `main::DATA`, ainsi il est utilisé dans le programme principal. Le token `__DATA__` ouvre le handle de fichier `DATA` dans n'importe quel paquetage effectif à ce moment. Ainsi différents modules peuvent posséder leur propre handle de fichier `DATA`, puisqu'ils ont (on le présume) des noms de paquetage différents.

`$DEBUGGING`
`$^D`

[GLO] La valeur courante des arguments internes, positionnés par l'option **`-D`** de la ligne de commande ; voir le chapitre 19 pour les valeurs de bit. (Mnémonique : valeur de l'option **`-D`**).

`$EFFECTIVE_GROUP_ID`
`$EGID`
`$)`

[GLO] Le gid (group ID) effectif de ce processus. Si vous vous trouvez sur une machine qui accepte que l'on soit membre de plusieurs groupes simultanément, `$)` donne une liste des groupes concernés, séparés par des espaces. Le premier nombre est celui qui est renvoyé par *getgid(2)*, et les suivants par *getgroups(2)*, l'un d'entre eux pouvant être identique au premier nombre.

De même, une valeur assignée à `$)` doit également être une liste de nombres séparés par des espaces. Le premier nombre est utilisé pour positionner le GID effectif, et les autres (s'il y en a) sont passés à l'appel système *setgroups(2)*. Pour obtenir l'effet d'une liste vide pour `setgroups`, il suffit de répéter le nouveau GID effectif ; par exemple, pour forcer un GID effectif de 5 et une liste vide effective pour

`setgroups` :

```
$) = "5 5";
```

(Mnémonique : les parenthèses sont employées pour *regrouper* les choses. Le GID effectif est le groupe qui est *fermement* le bon, si votre script est lancé avec setgid. Il faut donc une parenthèse fermante). Remarque : `$<`, `$>`, `$(` et `$)` ne peuvent être modifiés que sur des machines qui supportent la routine système set-id correspondante. `$(` et `$)` ne peuvent être échangés que sur des machines supportant *setregid(2)*.

`$EFFECTIVE_USER_ID`
`$EUID`
`$>`

[GLO] L'UID effectif de ce processus tel que renvoyé par l'appel système *geteuid(2)*.

Exemple :

```
$< = $>;              # mettre l'uid réel à l'uid effectif
($<, $>) = ($>, $<);  # échanger l'uid réel et l'uid effectif
```

(Mnémonique : il s'agit de l'UID *vers* lequel vous êtes parvenus si votre script est lancé avec setuid). Remarque : `$<` et `$>` ne peuvent être échangés que sur des machines qui supportent *setreuid(2)*. Parfois même, cela ne suffit pas.

`%ENV`

[GLO] Le hachage contenant les variables de l'environnement courant. Modifier une valeur dans `%ENV` change l'environnement à la fois pour votre processus et pour les processus fils lancés après l'affectation. (On ne peut modifier l'environnement d'un processus parent sur aucun système ressemblant à Unix).

```
$ENV{PATH}  = "/bin:/usr/bin;"
$ENV{PAGER} = "less";
$ENV{LESS}  = "MQeicsnf";     # nos options favorites pour less(1)
system "man perl";            # récupère les nouveaux paramètres
```

Pour supprimer quelque chose dans votre environnement, soyez sûr d'utiliser la fonction `delete` au lieu d'un `undef` sur la valeur du hachage.

Remarquez que les processus qui sont lancés par une entrée de *crontab(5)* héritent d'un jeu de variables d'environnement particulièrement appauvri. (Si votre programme fonctionne parfaitement depuis la ligne de commande mais pas avec *cron*, c'est probablement la cause). Remarquez également que vous devez modifier `$ENV{PATH}`, `$ENV{SHELL}`, `$ENV{BASH_ENV}`, et `$ENV{IFS}` si votre script est lancé avec setuid. Voir le chapitre 23, *Sécurité*.

`$EVAL_ERROR`
`$@`

[GLO] L'exception courante levée ou le message d'erreur de syntaxe Perl de la dernière opération `eval`. (Mnémonique : quelle est « l'adresse » de la dernière erreur de syntaxe ?) Contrairement à `$!` (`$OS_ERROR`), qui est positionné sur un échec mais pas nettoyé lors d'un succès, on a la garantie que `$@` est bien positionné (à une valeur vraie) si le dernier `eval` a engendré une erreur de compilation ou une exception à l'exécution, mais aussi la garantie que `$@` est nettoyé (à une valeur fausse) si aucun de ces problèmes ne survient.

Les messages d'avertissement ne sont pas collectés dans cette variable. Cependant, vous pouvez installer une routine destinée à traiter les avertissements en positionnant `$SIG{__WARN__}` comme décrit plus loin dans cette section.

Remarquez que la valeur de `$@` peut être un objet exception plutôt qu'une chaîne. Dans ce cas, vous pouvez toujours le traiter comme une chaîne pourvu que l'objet exception ait défini une surcharge de la transformation en chaîne (*N.d.T.* : *stringification overload*) pour sa classe. Si vous propagez une exception en écrivant :

```
die if $@;
```

alors un objet exception appellera `$@->PROPAGATE` pour savoir que faire. (Une chaîne exception ajoutera une ligne « propagée à » à la chaîne.)

`$EXCEPTIONS_BEING_CAUGHT`
`$^S`

[GLO] Cette variable reflète la situation courante de l'interpréteur, en retournant vrai à l'intérieur d'un `eval`, faux sinon. Elle est indéfinie si l'analyse de l'unité courante de compilation n'est pas encore finie, ce qui peut être le cas dans les gestionnaires de signaux `$SIG{__DIE__}` et `$SIG{__WARN__}`. (Mnémonique : situation de `eval`.)

`$EXECUTABLE_NAME`
`$^X`

[GLO] Le nom avec lequel le binaire perl lui-même a été exécuté, provient de `argv[0]` en C.

`@EXPORT`

[PKG] Ce tableau est consulté par la méthode `import` du module `Exporter` pour trouver la liste des variables et sous-programmes d'autres paquetages qui doivent être exportés par défaut quand le module est utilisé (avec `use`) ou lorsqu'on emploie l'option d'importation `:DEFAULT`. Il n'est pas exempt des complaintes de `use strict`, ainsi il doit être déclaré avec our ou pleinement qualifié avec le nom du paquetage si vous avez activé ce pragma. De toute façon, toutes les variables qui commencent par « `EXPORT` » sont exemptes des avertissements comme quoi elles ne sont utilisées qu'une seule fois. Voir le chapitre 11, *Modules*.

`@EXPORT_OK`

[PKG] Ce tableau est consulté par la méthode `import` du module `Exporter` pour déterminer si une requête d'importation est légale. Il n'est pas exempt des complaintes de `use strict`. Voir le chapitre 11.

`%EXPORT_TAGS`

[PKG] Ce hachage est consulté par la méthode `import` du module `Exporter` lorsqu'on requiert un symbole d'importation débutant par deux points, comme dans `use POSIX ":queue"`. Les clés sont les tags commençant par deux points mais sans les deux points (comme `sys_wait_h`). Les valeurs doivent être des références à des tableaux contenant les symboles à importer lorsqu'on requiert ce tag ; toutes les valeurs du hachage doivent également apparaître soit dans `@EXPORT`, soit dans `@EXPORT_OK`. Ce hachage n'est pas exempt des complaintes de `use strict`. Voir le chapitre 11.

`$EXTENDED_OS_ERROR`
`$^E`

[GLO] Information d'erreur spécifique au système d'exploitation courant. Sous Unix, `$^E` est identique à `$!` ($OS_ERROR), mais elle diffère sous OS/2, VMS, les systèmes Microsoft et MacPerl. Consultez les informations sur votre portage pour les spécificités. Les mises en garde mentionnées dans la description de `$!` s'appliquent aussi à `$^E` en général . (Mnémonique : explication étendue d'erreur.)

`@F`

[PKG] Le tableau dans lequel les lignes d'entrée sont éclatées quand la ligne de commande comporte l'option **`-a`** est donnée. Si l'option **`-a`** n'est pas utilisée, ce tableau n'a aucune signification particulière (ce tableau n'est en fait que `@main::F` et non dans tous les paquetages à la fois).

`%FIELDS`

[NON, PKG] Ce hachage est utilisé en interne par le pragma `use fields` pour déterminer les champs légaux courants dans un objet hachage. Voir `use fields`, `use base` et *Déclarations de champ avec use fields* au chapitre 12.

`format_formfeed` *`HANDLE EXPR`*
`$FORMAT_FORMFEED`
`$^L`

[GLO] Ce qu'une fonction `write` génère implicitement pour effectuer un saut de page avant d'émettre un en-tête de page. `"\f"` par défaut.

`format_lines_left` *`HANDLE EXPR`*
`$FORMAT_LINES_LEFT`
`$-`

[AHF] Le nombre de lignes restantes dans la page du canal de sortie actuellement sélectionné, pour une utilisation avec la déclaration `format` et la fonction `write`. (Mnémonique : `lines_on_pages lines_printed`.)

`format_lines_per_page` *`HANDLE EXPR`*
`$FORMAT_LINES_PER_PAGE`
`$=`

[AHF] La longueur de la page courante (lignes imprimables) du canal de sortie actuellement sélectionné, pour une utilisation avec `format` et `write`. 60 par défaut. (Mnémonique : est composé des lignes horizontales.)

`format_line_break_characters` *`HANDLE EXPR`*
`$FORMAT_LINE_BREAK_CHARACTERS`
`$:`

[GLO] Le jeu de caractères courant d'après lequel une chaîne peut être scindée pour remplir les champs de continuation (commençant par ^) dans un format. `" \n-"` par défaut, pour couper sur l'espace ou le tiret. (Mnémonique : un « deux-points » en poésie fait partie d'une ligne. Maintenant il ne vous reste plus qu'à vous rappeler du procédé mnémonique. . .)

`format_name` *`HANDLE EXPR`*
`$FORMAT_NAME`
`$~`

[AHF] Le nom du format de rapport courant pour le canal de sortie actuellement sélectionné. Le nom du handle du fichier par défaut. (Mnémonique : cette variable tourne après $^.)

`format_page_number` *`HANDLE EXPR`*
`$FORMAT_PAGE_NUMBER`
`$%`

[AHF] Le numéro de page courant du canal de sortie actuellement sélectionné, pour une utilisation avec `format` et `write`. (Mnémonique : % est le registre de numéro de page dans *troff(1)*. Quoi, vous ne savez pas ce qu'est *troff* ?)

`format_top_name` *`HANDLE EXPR`*
`$FORMAT_TOP_NAME`
`$^`

[AHF] Le nom du format d'en-tête de page courant pour le canal de sortie actuellement sélectionné. Le nom du handle de fichier auquel est ajouté *_TOP* par défaut. (Mnémonique : pointe vers le haut de la page.)

`$^H`

[NON, LEX] Cette variable contient les bits de statut du contexte lexical (autrement dit, des indications) pour le parseur Perl. Cette variable est strictement réservée à une utilisation interne. Sa disponibilité, son comportement et son contenu sont sujets à modifications sans préavis. Si vous y touchez, vous périrez indubitablement d'une mort horrible causée par une détestable maladie tropicale inconnue de la science. (Mnémonique : nous ne vous donnerons aucune indication.)

`%^H`

[NON, LEX] Le hachage %^H fournit la même sémantique de contexte lexical que $^H, le rendant utile pour l'implémentation des pragmas de contexte lexical. Lisez les affreuses mises en garde énoncées pour $^H, et ajoutez-y le fait que cette variable est encore expérimentale.

`%INC`

[GLO] Le hachage contenant les entrées de noms de fichiers pour chaque fichier Perl chargé *via* `do FICHIER`, `require` ou `use`. La clé est le nom du fichier spécifié, et la valeur est l'endroit où le fichier a été véritablement trouvé. L'opérateur `require` utilise ce tableau pour déterminer si un fichier donné a déjà été chargé. Par exemple :

```
% perl -MLWP::Simple -le 'print $INC{LWP/Simple.pm}'
/usr/local/lib/perl/5.6.0/lib/site_perl/LWP/Simple.pm
```

`@INC`

[GLO] Le tableau contenant la liste des répertoires où les modules Perl peuvent être trouvés par `do FICHIER`, `require` ou `use`. Initialement, il se compose des arguments de n'importe quelle option **`-I`** de la ligne de commande et des répertoires de la variable d'environnement `PERL5LIB`, suivis par les bibliothèques par défaut de

Perl, comme :

```
/usr/local/lib/perl5/5.6.0/i386-linux
/usr/local/lib/perl5/5.6.0
/usr/local/lib/perl5/site_perl/5.6.0/i386-linux
/usr/local/lib/perl5/site_perl/5.6.0
/usr/local/lib/perl5/site_perl/5.00552/i386-linux
/usr/local/lib/perl5/site_perl/5.00552
/usr/local/lib/perl5/site_perl/5.005/i386-linux
/usr/local/lib/perl5/site_perl/5.005
/usr/local/lib/perl5/site_perl
```

suivies par « `.` », qui représente le répertoire courant. Si vous avez besoin de modifier cette liste depuis votre programme, essayer le pragma `use lib`, qui non seulement modifie la variable durant la compilation, mais qui fait également des ajouts dans tous les répertoires dépendant de l'architecture (comme ceux qui contiennent les bibliothèques partagées utilisées par les modules XS) :

```
use lib "/monchemin/lib/"; use UnModule;
```

`$INPLACE_EDIT`
`$^I`

[GLO] La valeur courante de l'extension d'édition sur place. Employez `undef` pour désactiver l'édition sur place. Vous pouvez utilisez ceci à l'intérieur de votre programme pour obtenir le même comportement qu'avec l'option **`-i`**. Par exemple, pour produire l'équivalent de cette commande :

```
% perl -i.orig -pe 's/machin/truc/g' *.c
```

vous pouvez utiliser le code suivant dans votre programme :

```
local $^I = '.orig';
local @ARGV = glob("*.c");
while (<>) {
    s/machin/truc/g;
    print;
}
```

(Mnémonique : la valeur de l'option `-i`.)

`$INPUT_LINE_NUMBER`
`$NR`
`$.`

[GLO] Le numéro de l'enregistrement courant (généralement le numéro de ligne) pour le dernier handle de fichier dans lequel vous avez lu (ou sur lequel vous avez appelé `seek` ou `tell`). La valeur peut être différente de la ligne physique actuelle, selon la définition à ce moment de ce qu'est une « ligne » — voir `$/` (`$INPUT_RECORD_NUMBER`) pour la manière de choisir cette définition). Un `close` explicite sur un handle de fichier réinitialise à zéro le numéro de ligne. Comme `<>` n'effectue jamais de `close` explicite, les numéros de ligne augmentent en parcourant les fichiers `ARGV` (mais voyez les exemples de `eof`). La mise en `local` de `$.` a également pour effet de mettre en local la notion que Perl a du « dernier handle de fichier lu ». (Mnémonique : de nombreux programmes utilise « `.` » pour le numéro de ligne courant.)

```
$INPUT_RECORD_SEPARATOR
$RS
$/
```

[GLO] Le séparateur d'enregistrement en entrée, le caractère « saut de ligne » par défaut, celui qui est consulté par la fonction `readline`, l'opérateur `<HF>`, et la fonction `chomp`. Il se comporte comme la variable `RS` de *awk*, et, s'il contient la chaîne nulle, traite une ou plusieurs lignes blanches comme des délimiteurs d'enregistrements. (Mais une ligne vide ne doit contenir aucun espace caché ou aucune tabulation). Vous pouvez affecter à cette variable une chaîne de plusieurs caractères pour correspondre à un séparateur sur plusieurs caractères, mais vous ne pouvez pas lui affecter un motif — il faut bien que *awk* soit meilleur dans un domaine.

Remarquez que le fait d'affecter `"\n\n"` à `$/` signifie quelque chose de sensiblement différent que de lui affecter `""`, si le fichier contient des lignes blanches consécutives. `""` traitera deux lignes blanches consécutives ou plus comme une seule ligne blanche. `"\n\n"` signifie que Perl suppose aveuglément qu'une troisième ligne appartient au paragraphe suivant.

Si l'on rend carrément `$/` indéfini, la prochaine opération de lecture de ligne avalera le reste du fichier dans une seule valeur scalaire :

```
undef $/;         # enclenche le mode "cul-sec"
$_ = <HF>;        # maintenant le fichier entier est ici
s/\n[ \t]+/ /g;   # aplatit les lignes indentées
```

Si vous utilisez la construction `while (<>)` pour accéder au handle de fichier `ARGV` alors que `$/` vaut `undef`, chaque lecture prend le fichier suivant :

```
undef $/;
while (<>) {    # $_ contient le prochain fichier dans sa totalité
    ...
}
```

Bien que nous ayons employé `undef` ci-dessus, il est plus sûr de rendre `$/` indéfini en utilisant `local` :

```
{
    local $/;
    $_ = <HF>;
}
```

Affecter à `$/` une référence soit à un entier, soit à un scalaire contenant un entier, soit à un scalaire convertible en entier, fera que les opérations `readline` et `<HF>` liront des enregistrements de taille fixe (avec la taille maximale d'enregistrement valant l'entier référencé) au lieu d'un enregistrement de taille variable terminé par une chaîne particulière. Ainsi, ceci :

```
$/ = 32768; # ou \"32768" ou \$var_scalaire_contenant_32768
open(FICHIER, $monfichier);
$enregistrement = <FICHIER>;
```

lira un enregistrement d'au plus 32768 octets depuis le handle de `FICHIER`. Si vous ne lisez pas un fichier structuré par enregistrement (*N.d.T.* : *record-oriented file*) (ou si votre système d'exploitation ne connaît pas les fichiers structurés par enregistrement), alors vous aurez vraisemblablement un morceau de données à chaque lecture. Si un enregistrement est plus volumineux que la taille que vous avez fixée,

vous recevrez l'enregistrement découpé en plusieurs tranches. Le mode par « enregistrement » ne se marie bien avec le mode par ligne que sur les systèmes où les entrées/sorties standards fournissent une fonction *read(3)* ; VMS est une exception notoire.

Un appel à `chomp` lorsque `$/` est positionnée sur le mode par enregistrement — ou lorsqu'elle n'est pas définie — n'a aucun effet. Voir aussi les options de la ligne de commande **`-0`** (le chiffre) et **`-l`** (la lettre) dans le chapitre 19. (Mnémonique : / est utilisé pour séparer les lignes lorsque l'on cite de la poésie.)

`@ISA`

[PKG] Ce tableau contient les noms des autres paquetages dans lesquels il faut rechercher une méthode qui n'a pu être trouvée dans le paquetage courant. Autrement dit, il contient les classes de base du paquetage. Le pragma `use base` positionne ceci implicitement. Il n'est pas exempt des complaintes de `use strict`. Voir le chapitre 12.

`@LAST_MATCH_END`
`@+`

[DYN,LEC] Ce tableau contient les positions des fins des sous-éléments de la dernière recherche de correspondance ayant réussi dans le contexte dynamique actuellement actif. `$-[0]` est la position à la fin de la correspondance entière. Il s'agit de la même valeur que celle retournée par la fonction `pos` lorsqu'on l'appelle avec comme paramètre la variable sur laquelle s'est faite la recherche de correspondance. Lorsque nous parlons de « la position de la fin », nous voulons réellement dire la position du premier caractère *qui suit* la fin de ce qui a correspondu, ainsi nous pouvons soustraire les positions de début aux positions de fin pour obtenir la longueur. Le *n*ième élément de ce tableau contient la position du *n*ième élément de la correspondance, ainsi `$+[1]` est la position où finit `$1`, `$+[2]` la position où `$2` finit, etc. Vous pouvez utiliser `$#+` pour déterminer combien d'éléments ont réussi à correspondre dans la dernière recherche. Voir aussi `@-` (`@LAST_MATCH_START`).

Après une recherche de correspondance réussi sur une variable `$var` :

- `$'` est identique à `substr($var, 0, $-[0])`
- `$&` est identique à `substr($var, $-[0], $+[0] - $-[0])`
- `$'` est identique à `substr($var, $+[0])`
- `$1` est identique à `substr($var, $-[1], $+[1] - $-[1])`
- `$2` est identique à `substr($var, $-[2], $+[2] - $-[2])`
- `$3` est identique à `substr($var, $-[3], $+[3] - $-[3])`, etc.

`@LAST_MATCH_START`
`@-`

[DYN,LEC] Ce tableau contient les positions des débuts des sous-éléments de la dernière recherche de correspondance ayant réussi dans le contexte dynamique actif. `$-[0]` est la position au début de la correspondance entière. Le *n*ième élément de ce tableau contient la position du *n*ième élément de la correspondance, ainsi `$-[1]` est la position où débute `$1`, `$-[2]` la position où `$2` débute, etc. Vous pouvez utiliser `$#-` pour déterminer combien d'éléments ont réussi à correspondre dans la dernière recherche. Voir aussi `@+` (`@LAST_MATCH_END`).

`$LAST_PAREN_MATCH`
`$+`

[DYN, LEC] Cette variable renvoie ce qu'a trouvé la dernière recherche de parenthèse dans le contexte dynamique actuellement actif. C'est utile si l'on ne sait pas (ou si l'on ne s'en préoccupe pas) lequel des motifs parmi une alternative a été trouvée. (Mnémonique : Soyez positifs et regardez droit devant.) Exemple :

```
$rev = $+ if /Version: (.*)|Revision: (.*)/;
```

`$LAST_REGEXP_CODE_RESULT`
`$^R`

[DYN] Cette variable contient le résultat du dernier morceau de code exécuté au sein d'une recherche de motifs réussie construite avec `(?{ CODE })`. `$^R` vous donne un moyen d'exécuter du code et d'en retenir le résultat pour l'utiliser plus tard dans le motif, ou même encore après.

Comme le moteur d'expressions régulières de Perl se déplace tout au long du motif, il peut rencontrer de multiples expressions `(?{ CODE })`. Lorsque cela arrive, il retient chaque valeur de `$^R` de manière à ce qu'il puisse restaurer l'ancienne valeur de `$^R` si plus tard il doit reculer avant l'expression. En d'autres termes, `$^R` possède un contexte dynamique au sein du motif, un peu comme `$1` et ses amis.

Ainsi, `$^R` n'est pas juste le résultat du dernier morceau de code exécuté au sein d'un motif. C'est le résultat du dernier morceau de code *qui a conduit au succès de la correspondance*. Un corollaire est que si la correspondance a échoué, `$^R` reprendra la valeur qu'elle avait avant la correspondance.

Si le motif `(?{ CODE })` est employé directement comme la condition d'un sous-motif `(?(CONDITION)SI_VRAI|SI_FAUX)`, `$^R` n'est pas valorisée.

`$LIST_SEPARATOR`
`$"`

[GLO] Lorsqu'un tableau ou une tranche est interpolé dans une chaîne entre guillemets (ou un équivalent), cette variable indique la chaîne à mettre entre les éléments individuels. Un espace par défaut. (Mnémonique : on peut espérer que c'est évident.)

`$^M`

[GLO] Par défaut, on ne peut pas intercepter un dépassement de mémoire. Toutefois, si votre *perl* a été compilé pour bénéficier de `$^M`, vous pouvez l'utiliser comme un espace mémoire en cas d'urgence. Si votre Perl est compilé avec **`-DPERL_EMERGENCY_SBRK`** et utilise le `malloc` de Perl, alors :

```
$^M = 'a' x (1 < < 16);
```

allouera un tampon de 64K en cas d'urgence. Voir le fichier *INSTALL* dans le répertoire du code source de la distribution Perl pour les informations sur la manière d'activer cette option.

En guise d'effet dissuasif pour l'emploi de cette fonctionnalité avancée, il n'y a pas de nom long avec `use English` pour cette variable (et nous ne vous dirons pas quel est le procédé mnémonique).

`$MATCH`
`$&`

[DYN, LEC] La chaîne trouvée par la dernière recherche de motif dans le contexte dynamique actuellement actif. (Mnémonique : comme & dans certains éditeurs.)

`$OLD_PERL_VERSION`
`$]`

[GLO] Renvoie la version + niveau_de_patch/1000. On peut l'utiliser pour déterminer au début du script si l'interpréteur Perl qui exécute le script est dans une plage de versions correcte. (Mnémonique : cette version de Perl est-elle dans le bon intervalle ? L'intervalle mathématique étant représenté par un crochet). Exemple :

```
warn "Pas de somme de contrôle !\n" if $] < 3.019;
die "Exige la disponibilité du prototypage\n" if $] < 5.003;
```

Voir aussi la documentation de `use` *`VERSION`* et `require` *`VERSION`* pour un moyen convenable d'échouer si l'interpréteur Perl est trop ancien. Voir `$^V` pour une représentation UTF-8 plus flexible de la version de Perl.

`$OSNAME`
`$^O`

[GLO] Cette variable contient le nom de la plate-forme (généralement du système d'exploitation) pour laquelle le binaire Perl courant a été compilé. Elle sert surtout à ne pas appeler le module `Config`.

`$OS_ERROR`
`$ERRNO`
`$!`

[GLO] Dans un contexte numérique, donne la valeur courante de la dernière erreur d'appel système, avec toutes les restrictions d'usage. (Ce qui signifie qu'il vaut mieux ne pas attendre grand chose de la valeur de `$!` à moins qu'une erreur renvoyée n'indique spécifiquement une erreur système). Dans un contexte de chaîne, `$!` donne la chaîne du message d'erreur correspondant. Vous pouvez assigner un numéro d'erreur à `$!` si, par exemple, vous voulez que `$!` renvoie la chaîne correspondant à cette erreur particulière, ou si vous voulez positionner la valeur de sortie de `die`. Voir aussi le module `Errno` au chapitre 32. (Mnémonique : j'ai entendu qu'on venait de s'exclamer ?)

`%OS_ERROR`
`%ERRNO`
`%!`

[GLO] Ce hachage n'est défini que si vous avez chargé le module standard `Errno` décrit au chapitre 32. Une fois ceci fait, vous pouvez accéder à `%!` en utilisant une chaîne d'erreur particulière, et sa valeur n'est vraie que s'il s'agit de l'erreur courante. Par exemple, `$!{ENOENT}` n'est vraie que si la variable C `errno` vaut à ce moment la valeur de `ENOENT` dans le `#define` en C. C'est un moyen pratique d'accéder aux symboles spécifiques d'un fabriquant.

autoflush *HANDLE EXPR*
$OUTPUT_AUTOFLUSH
$AUTOFLUSH
$|

[AHF] Si elle est positionnée à vrai, cette variable force un tampon à être vidé après chaque print, printf et write sur le handle de sortie actuellement sélectionné. (C'est ce que nous appelons *vidage du tampon de commande*. Contrairement à la croyance populaire, positionner cette variable ne désactive pas le vidage du tampon.) Elle vaut faux, par défaut, ce qui signifie sur de nombreux systèmes que STDOUT est vidé ligne par ligne si la sortie s'effectue sur terminal, et vidé bloc par bloc si ce n'est pas le cas, même dans les tubes et les sockets. Positionner cette variable est utile lorsque vous envoyez la sortie vers un tube, comme lorsque vous lancez un script Perl sous *rsh(1)* et que vous voulez en voir la sortie en temps réel. Si vous avez des données en attente d'être vidées dans le tampon du handle de fichier actuellement sélectionné lorsque cette variable est positionnée à vrai, ce tampon sera immédiatement vidé par effet de bord de l'affectation. Voir la forme à un seul argument de select pour des exemples de contrôle du tampon sur les handles de fichier autres que STDOUT. (Mnémonique : quand vous voulez que vos tubes soient fumants.)

Cette variable n'a aucun effet sur le vidage du tampon d'entrée ; pour ceci, voir getc dans le chapitre 29 ou l'exemple dans le module POSIX au chapitre 32.

$OUTPUT_FIELD_SEPARATOR
$OFS
$,

[GLO] Le séparateur de champs en sortie (le limitateur, en fait) pour print. Normalement, print affiche simplement la liste d'éléments que vous spécifiez sans rien entre eux. Affecter cette variable fera spécifier à la variable OFS de *awk* ce qui doit être affiché entre les champs. (Mnémonique : ce qui est affiché quand il y a une « , » dans l'instruction print.)

$OUTPUT_RECORD_SEPARATOR
$ORS
$\

[GLO] Le séparateur d'enregistrement en sortie (le délimiteur, en fait) pour print. Normalement, print affiche simplement les champs séparés par des virgules que vous avez spécifié, sans saut de ligne final, ni séparateur d'enregistrements. Affectez cette variable, comme vous auriez affecté la variable ORS de *awk*, pour spécifier ce qui est affiché à la fin du print. (Mnémonique : vous assignez $\ au lieu d'ajouter "\n" à la fin du print. De même, cette variable ressemble à /, mais pour ce que Perl « retourne ».) Voir aussi l'option de la ligne de commande **-l** (pour « ligne ») dans le chapitre 19

%OVERLOAD

[NON, PKG] Ces entrées de hachage sont positionnées en interne grâce au pragma use overload pour implémenter la surcharge d'opérateur pour les objets de la classe du paquetage courant. Voir le chapitre 13, *Surcharge*.

`$PERLDB`
`$^P`

[NON, GLO] La variable interne pour activer le débogueur Perl (*perl -d*).

`$PERL_VERSION`
`$^V`

[GLO] La révision, la version, et la sous-version de l'interpréteur Perl, représentés comme une « chaîne de version » binaire. Les v-chaînes n'ont généralement pas de valeur numérique, mais cette variable est doublement valuée, et elle a une valeur numérique équivalente à l'ancienne variable `$]` ; c'est-à-dire un nombre à virgule valant révision + version/1000 + sous-version/1000000. La valeur de la chaîne est composée des caractères UTF8 : `chr($revision) . chr($version) . chr($sous_version)`. Cela signifie que `$^V` n'est *pas* affichable. Pour l'afficher, vous devez écrire :

```
printf "%vd", $^V;
```

Dans les bons côtés, cela veut également dire que la comparaison normale de chaînes peut être utilisée pour déterminer si l'interpréteur Perl qui exécute votre script est dans une plage de versions correcte. (Ceci s'applique à n'importe quelle version numérique représentée avec des v-chaînes, pas seulement celles de Perl). Exemple :

```
warn "Pas de déclarations avec 'our'\n" if $^V lt v5.6;
```

Voir la documentation de `use` *`VERSION`* et `require` *`VERSION`* pour un moyen convenable d'échouer si l'interpréteur Perl est plus ancien que ce que vous espériez. Voir aussi `$]` pour la représentation originelle de la version de Perl.

`$POSTMATCH`
`$'`

[DYN, LEC] La chaîne qui suit tout ce qui a été trouvé par la dernière recherche de motif réussie dans le contexte dynamique actuellement actif. (Mnémonique : `'` suit souvent une chaîne protégée.) Exemple :

```
$_ = 'abcdefghi';
/def/;
print "$`:$&:$'\n";   # affiche abc:def:ghi
```

À cause du contexte dynamique, Perl ne peut pas savoir quel motif aura besoin de sauvegarder ses résultats dans ces variables, ainsi mentionner `$`` ou `$'` n'importe où dans le programme entraîne une pénalisation dans les performances pour toutes les recherches de motif tout au long du programme. Ce n'est pas un gros problème dans les programmes courts, mais vous voudrez probablement éviter cette paire de variables si vous écrivez le code d'un module réutilisable. L'exemple ci-dessus peut être réécrit de manière équivalente, mais sans le problème de performances globales, ainsi :

```
$_ = 'abcdefghi';
/(.*?)(def)(.*)/s    # /s au cas où $1 contiendrait des sauts de ligne
print "$1:$2:$3\n";  # affiche abc:def:ghi
```

`$PREMATCH`
`$``

[DYN, LEC] La chaîne qui précède tout ce qui a été trouvé par la dernière recherche de motif réussie dans le contexte dynamique actuellement actif.

(Mnémonique : ‘ précède souvent une chaîne protégée.) Voir la remarque sur les performances de $’ ci-dessus.

$PROCESS_ID
$PID
$$

[GLO] Le numéro de processus (PID) du Perl en train d'exécuter ce script. Cette variable est automatiquement mise à jour dans un fork. En fait, vous pouvez même modifier $$ vous-même ; ceci ne changera pas votre PID de toute façon. Ce serait un joli coup. (Mnémonique : comme pour les shells divers et variés).

Vous devez faire attention de ne pas utiliser la variable $$ n'importe où elle pourrait être malencontreusement interprétée comme une déréférence : $$alphanum. Dans ce cas, écrivez ${$}alphanum pour distinguer de ${$alphanum}.

$PROGRAM_NAME
$0

[GLO] Contient le nom du fichier contenant le script Perl en train d'être exécuté. L'affectation d'une valeur à $0 est magique : elle tente de modifier la zone d'arguments que voit le programme *ps(1)*. Ceci est plus utile pour indiquer l'état courant du programme que pour cacher le programme en train de tourner. Mais cela ne fonctionne pas sur tous les systèmes. (Mnémonique : pareil que dans *sh*, *ksh*, *bash*, etc.)

$REAL_GROUP_ID
$GID
$(

[GLO] Le GID (groupe ID) réel de ce processus. Si vous vous trouvez sur une machine qui accepte que l'on soit membre de plusieurs groupes simultanément, $(donne une liste des groupes concernés, séparés par des espaces. Le premier nombre est celui qui est renvoyé par *getgid(2)*, et les suivants par *getgroups(2)*, l'un d'entre eux pouvant être identique au premier nombre.

De toute façon, une valeur assignée à $(doit être un seul et unique nombre utilisé pour positionner le GID. Ainsi la valeur donnée par $(ne doit *pas* être réaffectée à $(sans la forcer à être numérique, comme en lui ajoutant zéro. Ceci parce que vous ne pouvez avoir qu'un seul groupe réel. Voir plutôt $) ($EFFECTIVE_GROUP_ID), qui vous permet d'affecter plusieurs groupes effectifs.

(Mnémonique : les parenthèses sont employées pour *regrouper* les choses. Le GID réel est le groupe que vous laissez *ouvert* derrière vous, si votre script est lancé avec setgid. Il faut donc une parenthèse ouvrante.)

$REAL_USER_ID
$UID
$<

[GLO] L'UID réel de ce processus tel que renvoyé par l'appel système *geteuid(2)*. Savoir si vous pouvez le changer et comment le faire est dépendant de la bonne volonté de l'implémentation sur votre système — voir les exemples dans $> ($EFFECTIVE_USER_ID). (Mnémonique : il s'agit de l'UID *d'où* vous provenez si votre script est lancé avec setuid.)

`%SIG`

[GLO] Le hachage utilisé pour installer les gestionnaires de divers signaux. (Voir la section *Signaux* dans le chapitre 16, *Communication inter-processus*). Par exemple :

```
sub gestionnaire {
    my $sig = shift;  # le 1er argument est le nom du signal
    syswrite STDERR, "SIG$sig détecté--on quitte\n";
                      # Évitez les entrées/sorties standards dans les
                      # gestionnaires asynchrones pour supprimer le
                      # core dump). (Même cette concaténation de chaîne
                      # est risquée).
    close LOG;        # Ceci appelle une entrée/sortie standard, ainsi
                      # il peut y avoir quand même un core dump !
    exit 1;           # Mais puisque l'on quitte le programme,
                      # il n'y a pas de mal à essayer.
}

$SIG{INT} = \&gestionnaire;
$SIG{QUIT) = \&gestionnaire;
...
$SIG{INT} = 'DEFAULT';  # rétablir l'action par défaut
$SIG{QUIT} = 'IGNORE';  # ignorer SIGQUIT
```

Le hachage `%SIG` contient des valeurs indéfinies correspondant aux signaux pour lesquels aucun gestionnaire n'a été affecté. Un gestionnaire peut être spécifié comme une référence de sous-programme ou comme une chaîne. Une chaîne dont la valeur n'est pas l'une des deux actions spéciales « `DEFAULT` » ou « `IGNORE` » est le nom d'une fonction, qui, si elle n'est pas qualifiée par un paquetage, est interprétée comme faisant partie du paquetage `main`. Voici quelques exemples :

```
$SIG{PIPE} = "Plombier";     # OK, on suppose qu'il s'agit de
                             # main::plombier
$SIG{PIPE} = \&Plombier;     # bien, on utilise Plombier dans
                             # le paquetage courant
```

Certaines routines internes peuvent également être installées avec le hachage `%SIG`. La routine indiquée par `$SIG{__WARN__}` est appelée quand un message d'avertissement va être affiché. Le message est passé en premier argument. La présence d'une routine `__WARN__` provoque la suppression de l'affichage normal des messages d'avertissements vers `STDERR`. Vous pouvez vous en servir pour sauvegarder ces messages dans une variable ou pour transformer les avertissements en erreurs fatales, comme ceci :

```
local $SIG{__WARN__} = sub { die $_[0] };
eval $programmette;
```

C'est équivalent à :

```
use warnings qw/FATAL all/;
eval $programmette
```

excepté le fait que le premier code a un contexte dynamique alors que le second a un contexte lexical.

La routine indiquée par `$SIG{__DIE__}` offre un moyen de changer une exception batracienne en exception princière avec un baiser magique, ce qui souvent ne fonc-

tionne pas. La meilleure utilisation est l'exécution de choses de dernière minute lorsqu'un programme est sur le point de mourir suite à une exception non interceptée. Vous ne sauverez pas votre peau ainsi, mais vous aurez le droit de formuler une dernière volonté.

Le message d'exception est passé en premier argument. Quand une routine `__DIE__` revient, le traitement de l'exception continue comme il l'aurait fait en l'absence de l'interception, à moins que la routine d'interception ne quitte elle-même par un `goto`, une sortie de boucle ou un `die`. Le gestionnaire de `__DIE__` est explicitement désactivé pendant l'appel, afin que vous puissiez vous-même appeler alors le véritable `die` depuis un gestionnaire de `__DIE__`. (Si ce n'était pas le cas, le gestionnaire s'appellerait lui-même indéfiniment.) Le gestionnaire pour `$SIG{__WARN__}` procède de la même façon.

Seul le programme principal doit positionner `$SIG{__DIE__}`, pas le module. Ceci car actuellement, même les exceptions qui sont interceptées continuent de déclencher un gestionnaire de `$SIG{__DIE__}`. Ceci est fortement déconseillé car vous pourriez briser d'innocents modules qui ne s'attendaient pas à ce que les exceptions qu'ils prévoyaient soient altérées. Utilisez cette fonctionnalité uniquement en dernier ressort, et si vous le devez, mettez toujours devant un `local` pour limiter la durée du danger.

N'essayez pas de construire un mécanisme de gestion de signal à partir de cette fonctionnalité. Utilisez plutôt `eval {}` pour intercepter les exceptions.

`STDERR`

[GLO] Le handle de fichier spécial pour la sortie d'erreur standard dans n'importe quel paquetage.

`STDIN`

[GLO] Le handle de fichier spécial pour l'entrée standard dans n'importe quel paquetage.

`STDOUT`

[GLO] Le handle de fichier spécial pour la sortie standard dans n'importe quel paquetage.

`$SUBSCRIPT_SEPARATOR`
`$SUBSEP`
`$;`

[GLO] Le séparateur d'indices pour l'émulation de hachage multidimensionnels. Si vous vous référez à un élément de hachage comme :

```
$machin{$a,$b,$c}
```

cela veut vraiment dire :

```
$machin{join($;, $a, $b, $c)}
```

Mais ne mettez pas :

```
@machin{$a,$b,$c}    # une tranche de tableau - remarquez le @
```

qui signifie :

```
($m{$a},$machin{$b},$machin{$c})
```

La valeur par défaut est `"\034"`, la même que SUBSEP dans *awk*. Remarquez que si vos clés contiennent des données binaires, il ne peut exister de valeur sûre pour `$;`.

(Mnémonique : virgule — le séparateur d'indices syntaxique — est un point-virgule. Oui, je sais, c'est un peu faible, mais `$,` est déjà prise pour quelque chose de plus important.)

Bien que nous n'ayons pas découragé cette fonctionnalité, vous devez plutôt envisager d'utiliser maintenant de « véritables » hachages multidimensionnels, comme `$machin{$a}{$b}{$c}` au lieu de `$machin{$a,$b,$c}`. Les simulations de hachages multidimensionnels peuvent cependant être plus faciles à trier et sont plus à même d'utiliser les fichiers `DBM`.

`$SYSTEM_FD_MAX`
`$^F`

[GLO] Le descripteur de fichier « système » maximum, généralement 2. Les descripteurs de fichiers systèmes sont passés aux nouveaux programmes durant un `exec`, alors que les descripteurs supérieurs ne le sont pas. De plus, pendant un `open`, les descripteurs de fichiers systèmes sont préservés même si le `open` échoue. (Les descripteurs de fichiers ordinaires sont fermés avant que l'ouverture ne soit tentée et le restent si elle échoue.) Remarquez que le statut de *close-on-exec* d'un descripteur de fichier se décide d'après la valeur de `$^F` au moment du `open`, et non au moment du `exec`. Évitez ceci en affectant à `$^F` une valeur qui explose les compteurs :

```
{
    local $^F = 10_000;
    pipe(SALUT, CAVA) or die "pipe a échoué : $!";
}
```

`$VERSION`

[PKG] Cette variable est accessible à tout moment où une version minimale acceptable d'un module est spécifiée, comme dans `use UnModule 2.5`. Si `$UNMODULE::VERSION` est inférieure, une exception est levée. Techniquement, c'est la méthode `UNIVERSAL->VERSION` qui inspecte cette variable, vous pouvez donc définir votre propre fonction `VERSION` dans le paquetage courant si vous voulez autre chose que le comportement par défaut. Voir le chapitre 12.

`$WARNING`
`$^W`

[GLO] La valeur booléenne courante de l'option globale d'avertissement (à ne pas confondre avec l'option globale d'asservissement, à propos de laquelle nous avons entendus de nombreux avertissements globaux). Voir aussi le pragma `use warnings` au chapitre 31, *Modules de pragma*, et les options de la ligne de commande **`-W`** et **`-X`** pour les avertissements dans un contexte lexical, qui ne sont pas affectés par cette variable. (Mnémonique : la valeur est liée à l'option **`-w`**.)

`${^WARNING_BITS}`

[NON, GLO] L'ensemble courant des tests d'avertissement activés par le pragma `use warnings`. Voir `use warnings` au chapitre 31 pour de plus amples détails.

`${^WIDE_SYSTEM_CALLS}`

[GLO} L'argument global qui permet à tous les appels systèmes effectués par Perl d'utiliser les *API wide-characters* natives du système, si elles sont disponibles. Il peut

également être activé depuis la ligne de commande en utilisant l'option **-C**. La valeur initiale est généralement 0 pour assurer la compatibilité avec les versions de Perl antérieures à la 5.6, mais il peut être automatiquement mis à 1 par Perl si le système fournit une valeur par défaut modifiable par l'utilisateur (comme *via* `$ENV{LC_TYPE}`). Le pragma `use bytes` annule les effets de cet argument dans le contexte lexical courant.

Et maintenant attachez vos ceintures pour un *grand* chapitre. . .

29

Fonctions

Ce chapitre décrit les fonctions intrinsèques de Perl par ordre alphabétique[1] par souci pratique de référence. Chaque description de fonction commence par un bref résumé de la syntaxe pour cette fonction. Les noms de paramètre en *`CAPITALES OBLIQUES`* indiquent en fait les emplacements pour des expressions ; le texte qui suit le résumé de la syntaxe décrit ensuite la sémantique de ces arguments lorsqu'on les fournit (ou qu'on les omet).

Vous pouvez considérer que les fonctions sont des termes dans une expression au même titre que les littéraux et les variables. Ou bien, vous pouvez considérer qu'il s'agit d'opérateurs préfixes, qui traitent les arguments qui les suivent. D'ailleurs, nous utilisons le mot *opérateur* la plupart du temps.

Certains de ces opérateurs, euh, fonctions, prennent comme argument une *`LISTE`*. Les éléments de la *`LISTE`* doivent être séparés par des virgules (ou par `=>`, qui est juste une drôle de variété de virgule). Les éléments de la *`LISTE`* sont évalués dans un contexte de liste, ainsi chaque élément renvoie, soit un scalaire, soit une valeur de liste, selon sa sensibilité au contexte de liste. Chaque valeur renvoyée, que ce soit un scalaire ou une liste, sera interpolée comme étant un composant de la séquence entière de valeurs scalaires. C'est-à-dire que les listes sont mises à plat dans une seule liste. Du point de vue de la fonction qui reçoit les arguments, l'argument global *`LISTE`* est toujours une valeur de liste à une dimension. (Pour interpoler un tableau comme un élément unique, vous devez explicitement créer et interpoler à la place une référence à ce tableau.)

Les fonction prédéfinies de Perl peuvent être utilisées avec ou sans parenthèses encadrant les arguments ; dans ce chapitre les résumés de syntaxe omettent les parenthèses. Si vous utilisez des parenthèses, la règle simple mais parfois surprenante est la suivante : si ça ressemble à une fonction, alors *c'est* une fonction, ainsi la précédence n'a pas d'importance. Sinon, il s'agit d'un opérateur de liste ou d'un opérateur unaire, et la précé-

1. Parfois, les fonctions étroitement liées sont regroupées ensemble dans les pages man du système, nous respectons donc ici cette organisation. Pour trouver par exemple la description de `endpwent`, vous devrez cherchez sous `getpwent`.

dence devient importante. Soyez vigilants, car même si vous mettez un espace entre le mot-clé et sa parenthèse ouvrante, cela ne l'empêche pas d'être une fonction :

```
print 1+2*4;      # Affiche 9.
print(1+2) * 4;   # Affiche 3 !
print (1+2)*4;    # Affiche aussi 3 !
print +(1+2)*4;   # Affiche 12.
print ((1+2)*4);  # Affiche 12.
```

Perl peut vous avertir de cette situation, si vous le lancez avec l'option **-w**. Par exemple, la deuxième et la troisième ligne ci-dessus donneront ce type de messages :

```
print () interpreted as function at - line 2.
Useless use of integer multiplication (*) in void context at - line 2.
```

Avec la simple définition de certaines fonctions, vous avez une latitude considérable dans la manière de passer des arguments. Par exemple, la façon la plus habituelle d'utiliser `chmod` est de passer les permissions du fichier (le mode) comme argument initial :

```
chmod 0644, @tableau;
```

mais la définition de `chmod` indique juste :

```
chmod LISTE
```

ainsi vous pouvez tout aussi bien écrire :

```
unshift @tableau, 0644;
chmod @tableau;
```

Si le premier argument de la liste n'est pas un mode valide, `chmod` échouera, mais c'est un problème sémantique à l'exécution sans rapport avec la syntaxe de l'appel. Si la sémantique réclame des arguments spéciaux à donner en premier, le texte décrira ces restrictions.

Au contraire des simples fonctions de *LISTE*, d'autres fonctions imposent des contraintes syntaxiques supplémentaires. Par exemple, `push` a un résumé de syntaxe comme ceci :

```
push TABLEAU, LISTE
```

Cela veut dire que `push` demande un tableau correct comme premier argument, mais ne tient pas compte des arguments suivants. C'est ce que signifie le *LISTE* à la fin. (Les *LISTE* viennent toujours à la fin, puisqu'elles englobent toutes les valeurs qui restent). Si le résumé de syntaxe contient des arguments avant la *LISTE*, ces arguments sont distingués syntaxiquement par le compilateur, et non seulement distingués sémantiquement par l'interpréteur lorsqu'on le lancera plus tard. De tels arguments ne sont jamais évalués dans un contexte de liste. Ils peuvent être évalués soit dans un contexte scalaire, soit ce sont des références spéciales comme le tableau dans `push`. (La description vous dira qui est qui.)

Pour les opérations qui sont directement basées sur les fonctions de la bibliothèque C, nous n'essayerons pas de dupliquer la documentation de votre système. Lorsque la description d'une `fonction` demande de voir *fonction*(2), cela signifie que vous devez vous référer à la version correspondante en C de cette fonction pour en savoir plus sur sa sémantique. Le nombre entre parenthèses indique la section du manuel de programmation système dans laquelle vous trouverez la page man, si vous avez installez les pages man. (Et dans laquelle vous ne trouverez rien, si vous n'avez rien installé.)

Ces pages de manuel peuvent documenter des comportements spécifiques de votre système, comme les mots de passe shadow, les listes de contrôle d'accès et ainsi de suite. De nombreuses fonctions Perl dérivent de la bibliothèque C sous Unix mais sont émulées même sur les autres plates-formes. Par exemple, même si votre système d'exploitation ne connaît pas les appels système *flock*(2) et *fork*(2), Perl fait de son mieux pour les émuler avec les ressources natives de votre plate-forme.

Parfois, vous constaterez que, dans la documentation, la fonction C reçoit plus d'arguments que la fonction Perl correspondante. En général, les arguments supplémentaires sont déjà connus par Perl, comme la longueur d'un autre argument. Les autres différences viennent de la façon dont Perl et C spécifient les handles de fichiers, ainsi que les valeurs de succès et d'échec.

En général, les fonctions en Perl qui servent de sur-couches aux appels système du même nom (comme *chown(2)*, *fork*(2), *closedir*(2), etc.) renvoient toutes une valeur vraie lorsqu'elle réussissent, sinon `undef` comme indiqué dans les descriptions qui suivent. Ceci est différent des interfaces C à ces opérations, lesquelles retournent toutes `-1` en cas d'échec. Les exceptions à cette règle sont `wait`, `waitpid` et `syscall`. Les appels système positionnent aussi la variable spéciale `$!` (`$OS_ERROR`). Les autres fonctions ne le font pas, excepté accidentellement.

Pour les fonctions qui peuvent être utilisées aussi bien dans un contexte scalaire que dans un contexte de liste, un échec est généralement indiqué dans un contexte scalaire en retournant une valeur fausse (en général `undef`) et dans un contexte de liste en retournant une liste vide. La réussite de l'opération est généralement indiquée en renvoyant une valeur qui sera évaluée comme vraie (dans le contexte).

Retenez la règle suivante : il *n'y a pas* de règle qui relie le comportement d'une fonction dans un contexte de liste à son comportement dans un contexte scalaire, ou vice versa. Il peut s'agir de deux choses totalement différentes.

Chaque fonction connaît le contexte dans lequel elle a été appelée. La même fonction, qui renvoie une liste lorsqu'on l'appelle dans un contexte de liste, retournera, lorsqu'on l'appelle dans un contexte scalaire, la valeur qui lui semble la plus appropriée. Certaines fonctions renvoient la longueur de la liste qui aurait été retournée dans un contexte de liste. D'autres fonctions retourne « l'autre » valeur, lorsque quelque chose peut être manipulé, soit par un nombre, soit par son nom. D'autres encore renvoient un compteur des opérations qui ont réussi. En général, les fonctions de Perl font exactement ce que vous voulez faire, à moins que vous ne vouliez de la cohérence.

Une dernière remarque : nous avons essayé d'être très cohérents dans notre emploi des termes *octets* et *caractères*. Historiquement, ces termes ont été confondus l'un avec l'autre (et avec eux-mêmes). Mais lorsque nous disons octet, nous désignons toujours un octet de 8 bits. Lorsque nous disons caractère, nous voulons dire un caractère dans son abstraction, généralement un caractère Unicode, lequel peut être représenté dans vos chaînes par un ou plusieurs octets.

Mais notez que nous avons dit « généralement ». Perl confond à dessein les octets avec les caractères dans le contexte d'une déclaration `use bytes`, ainsi à chaque fois que nous parlons de caractère, vous devez comprendre un octet dans un contexte `use bytes`, sinon un caractère Unicode. En d'autres termes, `use bytes` reprend la définition d'un caractère telle qu'on la concevait dans les précédentes versions de Perl. Ainsi, par exemple,

si nous disons qu'un `reverse` scalaire, inverse une chaîne caractère par caractère, ne nous demandez pas si cela veut *vraiment* dire caractères ou alors octets, car la réponse est : « Oui ».

Fonctions Perl par catégorie

Voici les fonctions de Perl et les mots-clés assimilés, classés par catégorie. Certaines fonctions apparaissent sous plusieurs en-têtes.

Manipulation de scalaires

`chomp`, `chop`, `chr`, `crypt`, `hex`, `index`, `lc`, `lcfirst`, `length`, `oct`, `ord`, `pack`, `q//`, `qq//`, `reverse`, `rindex`, `sprintf`, `substr`, `tr///`, `uc`, `ucfirst`, `y///`

Expressions rationnelles et recherche de motifs

`m//`, `pos`, `qr//`, `quotemeta`, `s///`, `split`, `study`

Fonctions numériques

`abs`, `atan2`, `cos`, `exp`, `hex`, `int`, `log`, `oct`, `rand`, `sin`, `sqrt`, `srand`

Traitement des tableaux

`pop`, `push`, `shift`, `splice`, `unshift`

Traitement des listes

`grep`, `join`, `map`, `qw//`, `reverse`, `sort`, `unpack`

Traitement des hachages

`delete`, `each`, `exists`, `keys`, `values`

Entrées et sorties

`binmode`, `close`, `closedir`, `dbmclose`, `dbmopen`, `die`, `eof`, `fileno`, `flock`, `format`, `getc`, `print`, `printf`, `read`, `readdir`, `readpipe`, `rewinddir`, `seek`, `seekdir`, `select` (descripteurs de fichiers prêts), `syscall`, `sysread`, `sysseek`, `syswrite`, `tell`, `telldir`, `truncate`, `warn`, `write`

Données et enregistrements de longueur fixe

`pack`, `read`, `syscall`, `sysread`, `sysseek`, `syswrite`, `unpack`, `vec`

Handles de fichiers, fichiers et répertoires

`chdir`, `chmod`, `chown`, `chroot`, `fcntl`, `glob`, `ioctl`, `link`, `lstat`, `mkdir`, `open`, `opendir`, `readlink`, `rename`, `rmdir`, `select` (descripteurs de fichiers prêts), `select` (handle du fichier de sortie), `stat`, `symlink`, `sysopen`, `umask`, `unlink`, `utime`

Contrôle de flux des programmes

`caller`, `continue`, `die`, `do`, `dump`, `eval`, `exit`, `goto`, `last`, `next`, `redo`, `return`, `sub`, `wantarray`

Portée

`caller`, `import`, `local`, `my`, `no`, `our`, `package`, `use`

Divers

`defined`, `dump`, `eval`, `formline`, `lock`, `prototype`, `reset`, `scalar`, `undef`, `wantarray`

Processus et groupes de processus

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe,qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid

Modules de bibliothèques

do, import, no, package, require, use

Classes et objets

bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use

Accès aux sockets de bas niveau

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

Communications inter-processus System V

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Recherches d'informations sur les groupes et les utilisateurs

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Recherches d'informations réseaux

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Date et heure

gmtime, localtime, time, times

Fonctions Perl par ordre alphabétique

La plupart des fonctions suivantes sont annotées avec, euh, des annotations. Voici leur signification :

$_ Utilise $_ ($ARG) comme variable par défaut.

$! Modifie $! ($OS_ERROR) pour les erreurs dans les appels système.

$@ Lève des exceptions ; utilisez eval pour intercepter $@ ($EVAL_ERROR).

$? Modifie $? ($CHILD_ERROR) lorsque le processus fils se termine.

T Marque la donnée retournée.

T Marque la donnée retournée dans certains systèmes, localement, ou configurée manuellement.

X ARG Lève une exception si un argument d'un type inapproprié est donné.

X RO Lève une exception si on modifie une cible en lecture seule.

[x T] Lève une exception si on l'alimente avec une donnée marquée.

[x U] Lève une exception s'il n'existe pas d'implémentation sur la plate-forme courante.

Les fonctions qui retournent une donnée marquée lorsqu'elles sont alimentées avec une donnée marquée ne sont elles-mêmes pas marquées, puisque c'est le cas de la plupart des fonctions. En particulier, si vous utilisez n'importe quelle fonction avec `%ENV` ou `@ARGV`, vous obtiendrez une donnée marquée.

Les fonctions annotées avec [x ARG] lèvent une exception lorsqu'elles attendent, mais ne reçoivent pas, un argument d'un type précis (comme des handles de fichiers pour les opérations d'entrée/sortie, des références pour `bless`, etc.).

Les fonctions annotées avec [x RO] ont parfois besoin de modifier leurs arguments. Si elles ne peuvent pas modifier l'argument car il est marqué en lecture seule, elles lèvent une exception. Des exemples de variables en lecture seule sont les variables spéciales contenant une donnée capturée durant une recherche de motif et les variables qui sont en fait des alias de constantes.

Les fonctions annotées avec [x U] peuvent ne pas être implémentées sur toutes les plates-formes. Bien que la plupart d'entre elles soient nommées d'après les fonctions de la bibliothèque C sous Unix, ne croyez pas que vous ne pouvez en appeler aucune, seulement parce que vous n'êtes pas sous Unix. Beaucoup sont émulées, même celles que vous n'espériez jamais voir — comme `fork` sur les systèmes Win32, qui est en place depuis la version 5.6 de Perl. Pour plus d'informations sur la portabilité et le comportement de fonctions spécifiques à un système, voir la page de man *perlport*, plus quelques documentations spécifiques à la plate-forme fournie par votre portage de Perl.

Les fonctions qui lèvent d'autres exceptions diverses et variées sont annotées avec [$@], y compris les fonctions mathématiques qui émettent des erreurs d'intervalle, comme `sqrt(-1)`.

abs [$_]

```
abs VALEUR
abs
```

Cette fonction renvoie la valeur absolue de son argument.

```
$diff = abs($premier - $second);
```

Remarque : ici et dans les exemples suivants, un bon style (et le pragma `use strict`) exigerait que vous ajoutiez un modificateur `my` pour déclarer une nouvelle variable de portée lexicale, comme ceci :

```
my $diff = abs($premier - $second);
```

Toutefois, nous avons omis `my` de la plupart de nos exemples pour plus de clarté. Vous n'avez qu'à supposer qu'une telle variable a été déclarée plus tôt, si cela vous démange.

accept [$!] [x ARG] [x U]

```
accept SOCKET, SOCKET_PROTO
```

Cette fonction est utilisée par les processus serveurs qui désirent se mettre en attente de

connexion de clients sur une socket. *SOCKET_PROTO* doit être un handle de fichier déjà ouvert *via* l'opérateur `socket` et attaché à l'une des adresses réseau du serveur ou à `INADDR_ANY`. L'exécution est suspendue jusqu'à ce qu'une connexion soit établie, un handle de fichier *SOCKET* étant alors ouvert et attaché à la nouvelle connexion. Le handle original *SOCKET_PROTO* reste inchangé ; son seul but est d'être clôné dans une véritable socket. La fonction renvoie l'adresse de la connexion si l'appel réussit, une valeur fausse sinon. Par exemple :

```
unless ($interlocuteur = accept(SOCK, SOCK_PROTO)) {
    die "Ne peut accepter la connexion : $!\n";
}
```

Sur les systèmes qui l'acceptent, le flag close-on-exec sera positionné pour le nouveau descripteur de fichier ouvert, comme le veut la valeur de `$^F` (`$SYSTEM_FD_MAX`).

Voir *accept*(2). Voir aussi l'exemple dans la section *Sockets* au chapitre 16, *Communication interprocessus.*

alarm

```
alarm EXPR
alarm
```

Cette fonction envoie un signal `SIGALRM` au processus courant après *EXPR* secondes.

Un seul minuteur (*timer*) peut être actif à un moment donné. Chaque appel désactive le précédent minuteur et un argument *EXPR* valant 0 peut être fourni pour annuler le minuteur précédent sans en lancer un de nouveau.

```
print "Répondez en une minute ou allez en enfer : ";
alarm(60);                  # tue le programme dans une minute
$reponse = <STDIN>;
$temps_restant = alarm(0)  # efface l'alarme
print "Il vous reste $temps_restant secondes\n";
```

Une erreur courante est de mélanger les appels à `alarm` et `sleep`, car nombre de systèmes utilisent le mécanisme de l'appel système à *alarm*(2) pour implémenter *sleep*(3). Sur des machines plus anciennes, le temps écoulé peut être jusqu'à une seconde inférieur à celui que vous avez spécifié, selon la manière dont sont comptées les secondes. De plus, un système surchargé peut ne pas être prêt à lancer votre processus immédiatement. Voir le chapitre 16 pour plus d'informations sur l'interception de signaux.

Pour des alarmes d'une granularité plus fine que la seconde, vous pouvez utiliser la fonction `syscall` pour accéder à *setitimer*(2) si votre système le supporte. Le module CPAN, `Timer::HiRes` offre aussi des fonctions dans cet objectif.

atan2

```
atan2 Y, X
```

Cette fonction renvoie la valeur principale de l'arctangente de *Y/X* dans l'intervalle de -π à π. Une moyen rapide d'obtenir une valeur approchée de π est d'écrire :

```
$pi = atan2(1, 1) * 4;
```

Pour la tangente, vous pouvez utiliser la fonction `tan` provenant soit du module `Math::Trig`, soit du module `POSIX`, ou employer la relation bien connue :

```
sub tan { sin($_[0] / cos($_[0]) }
```

bind

$! X ARG X U X T

```
bind SOCKET, NOM
```

Cette fonction attache une adresse (un nom) à une socket déjà ouverte, spécifiée par le handle de fichier *SOCKET*. La fonction renvoie vrai si elle a réussi, sinon faux. *NOM* doit être une adresse empaquetée de la socket avec le type approprié.

```
use Socket;
$numero_port = 80;          # prétend que nous voulons être un serveur web
$adresse_socket = sockaddr_in($numero_port, INADDR_ANY);
bind SOCK, $adresse_socket
or die "Impossible d'attacher $numero_port : $!\n";
```

Voir *bind*(2). Voir aussi les exemples de la section *Sockets* au chapitre 16.

binmode

X ARG

```
binmode HANDLE_FICHIER, DISCIPLINES
binmode HANDLE_FICHIER
```

Cette fonction s'arrange pour que le *HANDLE_FICHIER* ait la sémantique spécifiée par l'argument *DISCIPLINES*. Si l'on omet *DISCIPLINES*, des sémantiques binaires (ou « raw ») sont appliquées au handle de fichier. Si *HANDLE_FICHIER* est une expression, la valeur est prise de manière appropriée, comme le nom du handle de fichier ou comme une référence à un handle de fichier.

La fonction `binmode` doit être appelée après l'`open` mais avant qu'une quelconque opération d'entrée/sortie sur le handle de fichier n'ait été effectuée. Le seul moyen de réinitialiser le mode d'un handle de fichier est de rouvrir le fichier, puisque les diverses disciplines ont pu conserver divers bits et segments de données dans diverses mémoires tampons. Cette restriction est susceptible d'être levée à l'avenir.

Autrefois, `binmode` était principalement utilisé sur les systèmes d'exploitation dont les bibliothèques d'exécution distinguaient les fichiers textes des fichiers binaires. Sur ces systèmes, l'objectif de `binmode` était de désactiver la sémantique de texte par défaut. De toute façon, depuis l'avènement d'Unicode, tous les programmes sur tous les systèmes doivent connaître la distinction, même sur les systèmes Unix ou Mac. De nos jours, il n'existe qu'un type de fichiers binaire (du moins pour Perl), mais une multitude de types de fichiers texte. Ainsi, Perl ne connaît qu'un seul format interne pour les textes Unicode, UTF-8. Puisqu'il existe une variété de fichiers texte, ceux-ci doivent souvent être convertis en entrée vers UTF-8 et en sortie vers le jeu de caractère d'origine ou vers une autre représentation d'Unicode. Vous pouvez utiliser les disciplines pour indiquer à Perl comment réaliser exactement (ou inexactement) ces conversions.[2]

2. Plus précisément, vous *serez* capable d'employer les disciplines pour ceci, mais nous n'avons pas achevé leur implémentation au moment où ces lignes sont écrites.

Par exemple, une discipline de `":text"` dira à Perl de traiter les textes de façon générique sans préciser quel type de texte traiter. Mais des disciplines comme `":utf8"` ou `":latin1"` disent à Perl quel format de texte lire et écrire. D'un autre côté, la discipline `":raw"` prévient Perl de garder ses sales mains en dehors des données. Pour en savoir plus sur le fonctionnement (éventuellement futur) des disciplines, voir la fonction `open`. La suite de cette discussion décrit ce que `binmode` fait sans l'argument *`DISCIPLINES`*, c'est-à-dire la signification historique de `binmode`, qui est équivalente à :

```
binmode HANDLE_FICHIER, ":raw";
```

Sauf indication contraire, Perl suppose que votre fichier fraîchement ouvert doit être lu ou écrit en mode texte. Le mode texte signifie que `\n` (newline) est votre caractère fin de ligne interne. Tous les systèmes utilisent `\n` comme leur caractère de fin de ligne interne mais ce qu'il représente réellement varie d'un système à l'autre, d'un périphérique à un autre et même d'un fichier à un autre, selon la façon dont vous accéder au fichier. Sur de tels systèmes (comprenant MS-DOS et VMS), ce que votre programme voit comme un `\n` peut ne pas être ce qui est stocké physiquement sur le disque. Le système d'exploitation peut, par exemple, stocker des fichiers textes avec les séquences `\cM\cJ` qui sont converties en entrée pour apparaître comme `\n` dans votre programme et inversement convertir `\n` depuis votre programme vers `\cM\cJ` en sortie dans un fichier. La fonction `binmode` désactive cette conversion automatique sur de tels systèmes.

En l'absence d'un argument *`DISCIPLINES`*, `binmode` n'a aucun effet sous Unix ou MAC OS, tous deux utilisant `\n` pour terminer chaque fin de ligne et représentant ceci avec un seul caractère. (Il peut s'agir toutefois d'une autre caractère : Unix utilise `\cJ` et les anciens Macs `\cM`. Mais cela ne porte pas à conséquence.)

Les exemples suivants montrent comment un script Perl doit lire une image GIF depuis un fichier et l'afficher sur la sortie standard. Sur les systèmes qui autrement altéreraient les données littérales en quelque chose d'autre que leur exacte représentation physique, vous devez préparer les deux handles de fichiers. Alors que vous pourriez directement utiliser une discipline `":raw"` pour ouvrir le fichier GIF, vous ne pouvez pas si facilement faire de même avec les handles de fichiers déjà ouverts, tels que `STDOUT` :

```
binmode STDOUT;
open(GIF, "vim-power.gif") or die "Impossible d'ouvrir vim-power.gif :
$!\n";
binmode GIF;
while (read(GIF, $buf, 1024)) {
    print STDOUT $buf;
}
```

bless

```
bless REF, NOM_DE_CLASSE
bless REF
```

Cette fonction indique à l'objet sur lequel pointe *`REF`* qu'il est maintenant un objet du paquetage *`NOM_DE_CLASSE`* — ou du paquetage courant si aucun *`NOM_DE_CLASSE`* n'est spécifié. Si *`REF`* n'est pas une référence valide, une exception est levée. Pour des raisons pratiques, `bless` retourne la référence, puisque c'est souvent la dernière fonction d'un constructeur. Par exemple :

```
$dromadaire = Animal->nouveau(TYPE => "Chameau", NOM => "amelia");

# puis dans Amimal.pm :
sub nouveau {
    my $classe = shift;
    my %attrs = @_;
    my $obj = { %attrs };
    return bless($obj, $classe);
}
```

Vous devez généralement consacrer les objets avec `bless` dans des *`NOMS_DE_CLASSE`* en majuscules et minuscules mélangées. Les espaces de noms tout entiers en minuscules sont réservé pour des besoins internes, comme les pragmas (directives de compilation) de Perl. Les types prédéfinis (tels que « SCALAR », « ARRAY », « HASH », etc., pour ne pas mentionner la classe de base de toutes les classes, « UNIVERSAL ») sont tous en majuscules, alors peut-être vaut-il mieux que vous évitiez des noms de paquetages écrits ainsi.

Assurez-vous que *`NOM_DE_CLASSE`* ne soit pas à une valeur fausse, la consécration dans des paquetages faux n'est pas supportée et peut conduire à des résultats imprévisibles.

Ce n'est pas un bug qu'il n'y ait pas d'opérateur `curse` correspondant. (Mais il existe un opérateur `sin`, jeu de mot intraduisible, l'opérateur `sin` s'occupant de la fonction sinus et non d'un quelconque blasphème.) Voir aussi le chapitre 12, *Objet*, pour en savoir plus sur la bénédiction (et les consécrations) des objets.

caller

```
caller EXPR
caller
```

Cette fonction renvoie des informations concernant la pile d'appels en cours de sous-programmes et assimilés. Sans argument, elle renvoie le nom du paquetage, le nom du fichier et le numéro de ligne d'où la routine en cours d'exécution a été appelée :

```
($paquetage, $fichier, $ligne) = caller;
```

Voici un exemple d'une fonction extrêmement fastidieuse, qui utilise les tokens spéciaux `__PACKAGE__` et `__FILE__` décrit au chapitre 2, *Composants de Perl* :

```
sub attention {
    my ($paquetage, $fichier) = caller;
    unless ($paquetage eq __PACKAGE__ && $file eq __FILE__) {
        die "Vous n'êtes pas supposé m'appeller, $paquetage !\n";
    }
    print "Appelez-moi en toute tranquillité\n";
}

sub appel_tranquille {
    attention();
}
```

Appelée avec un argument, `caller` interprète *`EXPR`* comme étant le nombre de groupes d'éléments dans la pile (*stack frames*) à remonter avant la routine courante. Par exemple,

un argument de 0 signifie l'élément courant dans la pile ; 1, la routine appelante ; 2, la routine appelante de la routine appelante ; et ainsi de suite. La fonction renvoie également des informations supplémentaires, comme le montre l'exemple suivant :

```
$i = 0;
while (($paquetage, $fichier, $ligne, $routine,
        $args_presents, $tableau_demande, $texte_eval, $est_requis,
        $indications, $masque_de_bits) = caller($i++))
{
    ...
}
```

Si l'élément est un appel de sous-programme, `$args_presents` est vrai s'il possède son propre tableau `@_` (et non un qui soit emprunté à la routine qui l'a appelé). Sinon, `$routine` peut valoir `"(eval)"` si l'élément n'est pas un appel de sous-programme mais un `eval`. Si c'est le cas, les informations supplémentaires `$texte_eval` et `$est_requis` sont positionnées : `$est_requis` est vrai si l'élément est créé par une instruction `require` ou un `use`, et `$texte_eval` contient le texte de l'instruction `eval` *`EXPR`*. Plus particulièrement, pour une instruction `eval` *`BLOCK`*, `$fichier` vaut `"(eval)"`, mais `$texte_eval` est indéfini. (Remarquez également que chaque instruction `use` crée un élément `require` dans un élément `eval` *`EXPR`*.) `$indications et $masque_de_bits` sont des valeurs internes ; vous êtes priés de les ignorer à moins que vous ne soyez membre de la thaumatocratie.

Dans un esprit de magie encore plus poussée, `caller` positionne également le tableau `@DB::args` avec les arguments passé à l'élément donné de la pile — mais seulement lorsqu'on l'appelle depuis le paquetage `DB`. Voir le chapitre 20, *Le débogueur Perl*.

chdir

```
chdir EXPR
chdir
```

Cette fonction change le répertoire de travail du processus courant en *`EXPR`*, si cela est possible. Si *`EXPR`* est omise, on utilise le répertoire de base de l'utilisateur. La fonction renvoie vraie si elle réussit, sinon faux.

```
chdir "$prefixe/lib" or die "Impossible de faire un cd dans $prefixe/lib\n";
```

Voir aussi le module `Cwd`, décrit au chapitre 32, *Modules standard*, qui vous permet de garder automatiquement une trace de votre répertoire courant.

chmod

```
chmod LISTE
```

Cette fonction modifie les permissions d'une liste de fichiers. Le premier élément de la liste doit être le mode numérique, comme dans l'appel système *chmod*(2). La fonction renvoie le nombre de fichiers modifiés avec succès. Par exemple :

```
$compt = chmod 0755, 'fichier1', 'fichier2';
```

affectera `0`, `1` ou `2` à `$compt`, selon le nombre de fichiers modifiés. L'opération est jugée réussie par l'absence d'erreur et non par un véritable changement, car un fichier peut

avoir le même mode qu'avant l'opération. Une erreur signifie vraisemblablement que vous manquez de privilèges suffisants pour changer le mode du fichier car vous n'êtes ni son propriétaire, ni le super-utilisateur. Vérifiez `$!` pour découvrir la cause réelle de l'échec.

Voici une utilisation plus typique :

```
chmod(0755, @executables) == @executables
 or die "Modification impossible de certains éléments de @executables : $!";
```

Si vous devez connaître quels fichiers n'ont pas pu être changés, écrivez quelque chose comme ceci :

```
@immuables = grep {not chmod 0755, $_} 'fichier1', 'fichier2', 'fichier3';
die "$0 : impossible d'exécuter chmod @immuables\n" if @immuables;
```

Cet idiome utilise la fonction `grep` pour ne sélectionner que les éléments de la liste pour lesquels la fonction `chmod` a échoué.

Lorsque des données non littérales sont utilisées pour le mode, vous pouvez avoir besoin de convertir une chaîne octale en un nombre décimal en utilisant la fonction `oct`. Ceci car Perl ne devine pas automatiquement qu'une chaîne contient un nombre octal seulement parce qu'elle commence par un « 0 ».

```
$MODE_DEFAUT = 0644; # On ne peut pas utiliser de guillemets ici !
PROMPT: {
    print "Nouveau mode ? ";
    $chaine_mode = <STDIN>;
        exit unless defined $chaine_mode;  # test de fin de fichier
    if ($chaine_mode =~ /^\s*$/ {          # test de ligne à blanc
        $mode = $MODE_DEFAUT;
    }
    elsif ($chaine_mode !~ /^\d+$/) {
        print "Demande un mode numérique, $chaine_mode invalide\n";
        redo PROMPT;
    }
    else {
        $mode = oct($chaine_mode);         # convertit "755" en 0755
    }
    chmod $mode, @fichiers;
}
```

Cette fonction marche avec des modes numériques comme dans l'appel système Unix *chmod*(2). Si vous voulez une interface symbolique comme celle fournie par la commande *chmod*(1), voir le module `File::chmod` de CPAN.

Vous pouvez aussi importer les constantes symboliques `S_I*` du module `Fcntl` :

```
use Fcntl ':mode';
chmod S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH, @executables;
```

Certains considèrent que ceci est plus lisible qu'avec `0755`. Maintenant, c'est à vous de voir.

chomp

```
chomp VARIABLE
chomp LISTE
chomp
```

Cette fonction supprime (normalement) un saut de ligne à la fin d'une chaîne contenue dans une variable. Il s'agit d'une fonction légèrement plus sûre que chop (décrite ci-dessous) en ce qu'elle n'a pas d'effet sur une chaîne ne se terminant pas par un saut de ligne. Plus précisément, elle supprime la fin d'une chaîne correspondant à la valeur courante de $/, et non n'importe quel caractère de fin.

À l'inverse de chop, chomp renvoie le nombre de caractères supprimés. Si $/ vaut "" (en mode paragraphe), chomp enlève tous les caractères de fin de ligne à la fin de la chaîne sélectionnée (ou des chaînes si l'on « chompe » une *LISTE*). Vous ne pouvez pas « chomper » un littéral, seulement une variable.

Par exemple :

```
while (<PASSWD>) {
    chomp;   # évite d'avoir \n dans le dernier champ
    @tableau = split /:/;
    ...
}
```

Avec la version 5.6, la signification de chomp change légèrement en ce que les disciplines en entrée peuvent prendre le dessus sur la valeur de la variable $/ et affecter la manière dont les chaînes doivent être « chompées ». Ceci a l'avantage qu'une discipline en entrée peut reconnaître plus d'une variété de délimiteur de ligne (comme un paragraphe Unicode et les séparateurs de ligne), mais « chompe » encore en toute sécurité tout ce qui termine la ligne courante.

chop

```
chop VARIABLE
chop LISTE
chop
```

Cette fonction coupe le dernier caractère d'une chaîne et renvoie le caractère coupé. L'opérateur chop est utilisé en premier lieu pour couper le caractère fin de ligne à la fin d'un enregistrement en entrée, mais est plus efficace qu'une substitution. Si c'est tout ce que vous en faites, alors il est plus sûr d'utiliser chomp, puisque chop tronque toujours la chaîne d'un caractère quel qu'il soit, alors que chomp est plus sélectif.

Vous ne pouvez pas « choper » un littéral, seulement une variable.

Si vous « chopez » une *LISTE* de variables, chacune des chaînes de la liste est coupée :

```
@lignes = 'cat mon_fichier';
chop @lignes;
```

Vous pouvez « choper » tout ce qui est une *lvalue*, y compris une affectation :

```
chop($rep_courant = 'pwd');
chop($reponse = <STDIN>);
```

Ce qui est différent de :

```
$reponse = chop($tmp = <STDIN>); # MAUVAIS
```

qui met un caractère fin de ligne dans `$reponse` puisque `chop` renvoie le caractère éliminé et non la chaîne résultante (qui se trouve dans `$tmp`). Une manière d'obtenir le résultat voulu est d'utiliser `substr` :

```
$reponse = substr <STDIN>, 0, -1;
```

Mais on l'écrit plus fréquemment :

```
chop($reponse = <STDIN>);
```

La plupart du temps, on peut exprimer `chop` en termes de `substr` :

```
$dernier_caractere = chop($var);
$dernier_caractere = substr($var, -1, 1, ""); # la même chose
```

Une fois que vous aurez compris cette équivalence, vous pourrez l'utiliser pour faire des coupes plus évoluées. Pour couper plus d'un caractère, utilisez `substr` comme une lvalue, en assignant une chaîne vide. Ce qui suit enlève les cinq derniers caractères de `$caravane` :

```
substr($caravane, -5) = "";
```

Le paramètre négatif fait partir `substr` de la fin de la chaîne au lieu du début. Si vous voulez récupérer les caractères ainsi enlevés, vous pouvez employer la forme à quatre arguments de `substr`, créant ainsi quelque chose comme un quintuple « chop » :

```
$queue = substr($caravane, -5, 5, "");
```

chown

$! X U X T

```
chown LISTE
```

Cette fonction change le propriétaire et le groupe d'une liste de fichiers. Les deux premiers élements de la liste doivent être les UID et GID *numériques*, dans cet ordre. Une valeur de `-1`, dans l'une ou l'autre position, est interprétée par la plupart des systèmes en conservant la valeur inchangée. La fonction renvoie le nombre de fichiers modifiés avec succès. Par exemple :

```
($compt = chown($uidnum, $gidnum, 'fichier1', 'fichier2')) == 2
    or die "chown impossible pour fichier1 ou fichier2 : $!";
```

mettra `$compt` à `0`, `1` ou `2`, selon le nombre de fichiers modifiés (dans le sens où l'opération s'est déroulée avec succès, et non selon que le propriétaire est différent après coup). Voici une utilisation plus typique :

```
chown($uidnum, $gidnum, @fichiers) == @fichiers
    or die "Modification impossible de @fichiers avec chown : $!";
```

Voilà un sous-programme qui accepte un nom d'utilisateur, vérifie les UID et GID pour vous, puis effectue le `chown` :

```
sub chown_avec_le_nom {
    my ($utilisateur, @fichiers) = @_;
    chown((getpwnam($utilisateur))[2,3], @fichiers) == @fichiers
        or die "chown impossible pour @fichiers : $!";
```

```
}

chown_avec_le_nom("larry", glob("*.c"));
```

Cependant, il se peut que vous ne vouliez pas modifier le groupe ainsi que le fait la fonction précédente, car le fichier */etc/passwd* associe à chaque utilisateur, un groupe unique même si cet utilisateur peut être membre de plusieurs groupes secondaires dans */etc/groups*. Dans ce cas, vous pouvez passer `-1` pour le GID, ce qui laisse inchangé le groupe du fichier. Si vous passez `-1` comme UID et un GID valide, vous pouvez affecter le groupe sans changer l'utilisateur.

Sur la plupart des systèmes, vous n'avez la permission de changer le propriétaire du fichier que si vous êtes le super-utilisateur, bien que vous puissiez changer le groupe en n'importe lequel de vos groupes secondaires. Sur des systèmes peu sûrs, ces restrictions peuvent être assouplies, mais cette supposition n'est pas généralisable. Sur les systèmes POSIX, vous pouvez détecter quelle règle s'applique ainsi :

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
# essayez seulement si vous êtes le super-utilisateur
# ou sur un système permissif
if ($> == 0 || !sysconf(_PC_CHOWN_RESTRICTED)) {
    chown($uidnum, -1, $fichier)
        or die "chown de $fichier à $uidnum impossible : $!";
```


chr

$_

```
chr NOMBRE
chr
```

Cette fonction renvoie le caractère représenté par ce *NOMBRE* dans le jeu de caractères. Par exemple, `chr(65)` vaut « `A` » en ASCII comme en Unicode, et `chr(0x263a)` est un smiley Unicode. Pour obtenir l'inverse de `chr`, utilisez `ord`.

Si vous préférez spécifier votre caractère par son nom plutôt que par un nombre (par exemple, `"\N{WHITE SMILING FACE}"` pour un smiley Unicode), voir `charname` au chapitre 31, *Modules de pragmas*.

chroot

$_ $! [x u] [x t]

```
chroot FICHIER
chroot
```

En cas de succès, *FICHIER* devient le nouveau répertoire racine pour le processus courant, le point de départ des noms de chemin commençant par « / ». Ce répertoire est hérité par appels à `exec` et par tous les sous-processus `fork`és après l'appel à `chroot`. Il n'y a aucun moyen de revenir à l'état précédent un `chroot`. Pour des raisons de sécurité, seul le super-utilisateur peut utiliser cette fonction. Voici un morceau de code ressemblant approximativement à ce que font de nombreux serveurs FTP :

```
chroot((getpwnam('ftp'))[7]
    or die "Ftp anonyme impossible : $!\n";
```

Il est improbable que cette fonction marche sur des systèmes non-Unix. Voir *chroot*(2).

close

$! $? X ARG

```
close HANDLE
close
```

Cette fonction ferme le fichier, la socket ou le pipe associé au *HANDLE*. (Elle ferme le handle de fichier actuellement sélectionné si l'argument est omis.) Elle renvoie vrai si la fermeture se déroule avec succès, sinon faux. Vous n'avez pas besoin de fermer *HANDLE* si vous le rouvrez immédiatement, puisque le `open` suivant le ferme automatiquement pour vous. (Voir `open`.) Cependant, un `close` explicite sur un fichier en entrée remet à zéro le compteur de lignes (`$.`), ce qui n'est pas le cas de la fermeture implicite effectuée par `open`.

HANDLE peut être une expression dont la valeur peut être employée en tant que handle de fichier indirect (soit le non réel du handle de fichier, soit une référence à tout ce qui peut être interprété comme un objet handle de fichier.)

Si le handle de fichier vient d'une ouverture de pipe, `close` renverra faux si un appel système sous-jacent échoue ou si le programme à l'autre extrémité du pipe se termine avec un statut non nul. Dans ce dernier cas, le `close` force la variable `$!` (`$OS_ERROR`) à zéro. Donc, si un `close` sur un pipe renvoie un statut non nul, vérifiez `$!` pour déterminer si le problème vient du pipe lui-même (valeur non nulle) ou du programme à l'autre bout du pipe (valeur nulle). Dans tous les cas, `$?` (`$CHILD_ERROR`) contient la valeur de statut d'attente (*N.d.T.* : *wait status* ; voir son interprétation à la fonction `system`) de la commande associée à l'autre extrémité du pipe. Par exemple :

```
open(SORTIE, '| sort -rn | lpr -p') # pipe pour sort et lpr
    or die "Impossible de démarrer le pipe sortlpr : $!";
print SORTIE @lignes;               # imprime des choses sur la sortie
close SORTIE                        # attend que le sort finisse
    or warn $! ? "Erreur système lors de la fermeture du pipe sortlpr : $!"
               : "Statut d'attente du pipe sortlpr $?";
```

Un handle de fichier produit, en *dup*(2)*liquant* un pipe, est traité comme un handle de fichier ordinaire, ainsi `close` n'attendra pas le fils pour ce handle de fichier. Vous devez attendre le fils en fermant le handle de fichier d'origine. Par exemple :

```
open(NESTSTAT, "netstat -rn |")
    or die "Impossible de lancer netstat : $!";
open (STDIN, "<&NETSTAT")
    or die "Impossible de dupliquer vers stdin : $!";
```

Si vous fermez le `STDIN` ci-dessus, il n'y aura pas d'attente ; par contre, il y en aura si vous fermez `NETSTAT`.

Si vous vous débrouillez d'une manière ou d'une autre pour récolter vous-même un pipe fils qui s'est terminé, le `close` échouera. Ceci peut arriver si vous avez votre propre gestionnaire pour `$SIG{CHLD}` qui se déclenche lorsque le pipe fils se termine, ou si vous appelez intentionnellement `waitpid` avec l'identifiant de processus renvoyé par l'appel de `open`.

closedir

$! X ARG X U

```
closedir HANDLE_REP
```

Cette fonction ferme un répertoire ouvert par `opendir` et renvoie la réussite de l'opération. Voir les exemples de `readdir`. *`HANDLE_REP`* peut être une expression dont la valeur peut être employée en tant que handle de répertoire indirect, généralement le non réel du handle du répertoire.

connect

$! X ARG X T X U

```
connect SOCKET, NOM
```

Cette fonction démarre une connexion avec un autre processus qui attend par un `accept`. La fonction renvoie vrai si elle réussit, faux sinon. *`NOM`* doit être une adresse réseau binaire du type correspondant à celui de la socket. Par exemple, en supposant que `SOCK` soit une socket créée préalablement :

```
use Socket;

my ($distant, $port) = ("www.perl.com", 80);
my $adr_dest = sockaddr_in($port, inet_aton($distant));
connect SOCK, $adr_dest
    or die "Connexion impossible vers $distant sur le port $port : $!";
```

Pour déconnecter une socket, utilisez soit `close`, soit `shutdown`. Voir aussi les exemples de la section *Sockets* au chapitre 16. Voir *connect*(2).

cos

$_

```
cos EXPR
cos
```

Cette fonction renvoie le cosinus de *`EXPR`* (exprimée en radians). Par exemple, le script suivant affiche une table des cosinus d'angles mesurés en degrés :

```
# Voici la manière de ne pas se fatiguer pour convertir de degrés en radians

$pi = atan2(1,1) * 4;
$pi_apres_180 = $pi/180;

# Affiche la table
for ($deg = 0; $deg <= 90; $deg++) {
    printf "%3d %7.5f\n", $deg, cos($deg * $pi_apres_180);
}
```

Pour l'opération cosinus inverse, vous pouvez utilisez la fonction `acos()` des modules `Math::Trig` ou `POSIX`, ou encore utiliser cette relation :

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

crypt

```
crypt TEXTE_EN_CLAIR, PERTURBATION
```

Cette fonction calcule le hachage non bijectif d'une chaîne exactement comme le fait *crypt*(3). Ceci s'avère très utile pur vérifier que le fichier de mots de passe n'en contient pas qui soient trop faibles[3], bien que ce que vous vouliez réellement faire est d'empêcher les gens de commencer à en ajouter de mauvais.

`crypt` se veut une fonction non bijective, un peu comme casser des œufs pour faire une omelette. Il n'existe aucun moyen (connu) de décrypter un mot de passe chiffré, à moins d'essayer de deviner toutes les combinaisons de manière exhaustive.

Lorsque vous vérifiez une chaîne cryptée existante, vous devez utiliser le texte chiffré en tant que *PERTURBATION* (comme `crypt($texte_en_clair, $crypte) eq $crypte`). Ceci permet à votre code de fonctionner avec le `crypt` standard ainsi qu'avec d'autres implémentations plus exotiques.

Lorsque vous choisissez une nouvelle *PERTURBATION*, vous avez au moins besoin de créer une chaîne de deux caractères aléatoires appartenant à l'ensemble `[./0-9A-Za-z]` (comme `join '', ('.', '/', 0..9, 'A'..'Z', 'a'..'z')[rand 64, rand 64]`). Les anciennes implémentations de `crypt` avaient seulement besoin des deux premiers caractères de *PERTURBATION*, mais le code qui donnait seulement les deux premiers caractères est maintenant considéré comme non portable. Voir votre page de manuel locale de *crypt*(3) pour des détails intéressants.

Voici un exemple qui s'assure que quiconque lançant ce programme connaît son propre mot de passe :

```
$mot_de_passe = (getpwuid ($<))[1]  # Suppose que l'on soit sur Unix.

system "stty -echo";                # ou regarder Term::Readkey dans CPAN
print "Mot de passe : ";
chomp($mot = <STDIN>);
print "\n";
system "stty echo";

if (crypt($mot, $mot_de_passe) ne $mot_de_passe) {
    print "Désolé\n";
} else {
    print "ok\n";
}
```

Il est bien sûr déconseillé de donner votre mot de passe à quiconque le demande car c'est imprudent.

Les mots de passes cachés sont légèrement plus sûrs que les fichiers de mots de passe traditionnels et vous devez être le super-utilisateur pour y accéder. Comme peu de programmes doivent tourner avec des privilèges si puissants, il se peut que le programme maintienne son propre système d'autentification indépendant en stockant les chaînes `crypt`ées dans un fichier autre que `/etc/passwd` ou `/etc/shadow`.

3. Seules les personnes qui ont des intentions louables sont autorisées à faire ceci.

La fonction `crypt` n'est pas adaptée au chiffrement de grandes quantités de données, au moins parce que vous ne pouvez pas retrouver en sens inverse l'information en clair. Regardez dans les répertoires *by-module/Crypt* et *by-module/PGP* de votre miroir favori de CPAN pour trouver un grand nombre de modules potentiellement utiles.

dbmclose

```
dbmclose HACHAGE
```

Cette fonction brise le lien entre un fichier DBM (*DataBase Management*, gestion de base de données) et un hachage. `dbmclose` n'est en fait qu'un appel à `untie` avec les bons arguments, mais est fournie pour des raisons de compatibilité avec les anciennes versions de Perl.

dbmopen

```
dbmopen HACHAGE, NOMDB, MODE
```

Relie un fichier DBM à un hachage. (DBM signifie *DataBase Management*, gestion de base de données, et consiste en un ensemble de routines de bibliothèque C qui permettent un accès direct à des enregistrements *via* un algorithme de hachage.) *HACHAGE* est le nom de la base de données (sans l'extension *.dir* ou *.pg*). Si la base de données n'existe pas et qu'un *MODE* valide est spécifié, la base de données est créée avec les protections spécifiées par *MODE*, modifié par le *umask*. Pour empêcher la création de la base données au cas où elle n'existe pas, il est possible de spécifier un mode valant `undef`, et la fonction renverra faux si elle ne peut trouver de base existante. Les valeurs associées au hachage avant le `dbmopen` ne sont plus accessibles.

La fonction `dbmopen` n'est en fait qu'un appel à `tie` avec les arguments appropriés, mais elle est fournie pour des raisons de compatibilité avec les anciennes versions de Perl. Vous pouvez contrôler quelle est bibliothèque DBM à utiliser en employant directement l'interface `tie` ou en chargeant le module approprié avant d'appeler `open`. Voici un exemple qui fonctionne sur certains systèmes pour les versions de `DB_file` similaires à celle de votre navigateur Netscape :

```
use Db_file;
dbmopen(%NS_hist, "$ENV{HOME}/.netscape/history.dat", undef)
    or die "Impossible d'ouvrir le fichier d'historique de netscape : $!";

while (($url, $quand) = each %NS_hist) {
    next unless defined($quand);
    chop ($url, $quand);     # supprime les octets nuls à fin
    printf "Dernière visite de %s le %s.\n", $url,
        scalar(localtime(unpack("V", $quand)));
}
```

Si vous n'avez pas d'accès au fichier DBM en écriture, vous pourrez seulement lire les variables du hachage sans les modifier. Pour tester si vous pouvez écrire, utilisez soit un test de fichier comme `-w`, soit essayez de modifier une entrée dans un hachage bidon à l'intérieur d'un `eval {}`, ce qui interceptera l'exception.

Les fonctions comme `keys` et `values` peuvent renvoyer une importante liste de valeurs

lorsqu'on les utilise avec de grands fichiers DBM. Vous préférerez sans doûte utiliser la fonction `each` pour les parcourir itérativement et ne pas tout charger d'un bloc en mémoire.

Les hachages liés à des fichiers DBM ont les mêmes limitations que le type de paquetage DBM que vous utilisez, notamment pour ce qui concerne la dimension de ce qu'on peut mettre dans une cellule (*bucket*). Si vous vous restreignez à des clefs et des valeurs courtes, cela ne constitue que rarement un problème. Voir aussi le module `DB_file` au chapitre 32.

Une autre chose dont il faut tenir compte est que de nombreuses bases de données DBM existantes contiennent des clefs et des valeurs terminées par un caractère nul car elles ont été conçues pour être exploitées par des programmes C. Le fichier d'historique de Netscape et le fichier d'alias des anciens *sendmail* en sont des exemples. Il vous suffit d'utiliser `"$key\0"` lorsque vous extrayez une valeur, puis de supprimer le caractère nul dans cette dernière.

```
$alias = $les_alias{"postmaster\0"};
chop $alias;     # supprime le caractère nul
```

Il n'existe à l'heure actuelle aucun moyen intégré de verrouiller un fichier DBM générique. Certains pourraient considérer qu'il s'agit d'un bogue. Le module `GDBM_file` permet lui de verrouiller l'ensemble du fichier. En cas de doute, il vaut mieux utiliser un fichier verrou séparé.

defined

```
defined EXPR
defined
```

Cette fonction renvoie une valeur booléenne selon que *EXPR* contient une valeur définie ou non. La plupart des données que vous manipulez sont définies, mais un scalaire qui ne contient ni de chaîne valide, ni valeur numérique, ni de référence est considéré contenir la valeur indéfinie, ou pour abréger `undef`. Initialiser une variable scalaire à une valeur particulière la rendra définie et elle restera définie jusqu'à ce que vous lui assigniez une valeur indéfinie ou que vous appeliez explicitement `undef` avec cette variable comme paramètre.

De nombreuses opérations renvoient `undef` dans des conditions exceptionnelles, comme une fin de fichier, une variable non initialisée, une erreur système, etc. Puisque `undef` n'est qu'un type de valeur fausse, un simple test booléen ne fait pas de distinction entre `undef`, le chiffre zéro, la chaîne vide et la chaîne d'un seul caractère, « `0` » — toutes ces valeurs sont fausses de la même manière. La fonction `defined` vous permet de distinguer entre une chaîne vide indéfinie et une chaîne vide définie lorsque vous employez des opérateurs susceptibles de renvoyer une véritable chaîne vide.

Voici un morceau de code qui teste une valeur scalaire d'un hachage :

```
print if defined $option{D};
```

Lorsqu'on l'utilise sur un élément de hachage comme ceci, `defined` ne fait qu'indiquer si la valeur est définie, et non si la clef possède une entrée dans le hachage. Il est possible d'avoir une clef dont la valeur est indéfinie ; cependant la clef elle-même existe. Utilisez `exist` pour déterminer si une clef de hachage existe.

Dans l'exemple suivant, nous exploitons la convention qui fait que certaines opérations renvoient une valeur indéfinie lorsqu'il n'y a plus de données.

```
print "$val\n" while defined($val = pop(@tableau));
```

Et dans celui-ci, nous faisons la même chose avec la fonction `getpwent` pour obtenir des informations à propos des utilisateurs du système :

```
setpwent(); while (defined($nom = getpwent())) {
    print "< <$nom> >\n";
}
endpwent();
```

Il en va de même pour les erreurs renvoyées par les appels système qui peuvent légitimement renvoyer une valeur fausse :

```
die "readlink $sym impossible : $!"
    unless defined($valeur = readlink $sym);
```

Vous pouvez également utiliser `defined` pour déterminer si un sous-programme a d'ores et déjà été défini. Ceci évite de se surcharger avec des sous-programmes inexistants (ou des sous-programmes qui on été déclarés mais jamais définis) :

```
indirect("nom_fonction", @liste_args);
sub indirect {
    my $nom_routine = shift;
    no strict 'refs';      # ainsi nous pouvons utiliser
                           # nom_routine indirectement
    if (defined &$nom_routine) {
        &$nom_routine(@_); # ou $nom_routine->(@_);
    } else {
        warn "Appel ignoré à une fonction invalide $nom_routine";
    }
}
```

L'utilisation de `defined` sur les agrégats (hachages et tableaux) est dépréciée. (On l'utilisait pour savoir si la mémoire pour cet agrégat avait jamais été allouée.) À la place, employez un simple test booléen pour voir si si le tableau ou le hachage possède des éléments :

```
if (@un_tableau) { print "tableau possédant des éléments\n";
if (%un_hachage) { print "hachage possédant des éléments\n";
```

Voir aussi `undef` et `exists`.

delete

```
delete EXPR
```

Cette fonction supprime un élément (ou une tranche d'éléments) dans le hachage ou le tableau spécifié. (Voir `unlink` si vous voulez supprimer un fichier.) Les éléments supprimés sont renvoyés dans l'ordre spécifié, bien que ce comportement ne soit pas garanti dans le cas de variables liées comme les fichiers DBM. Après la suppression, la fonction `exists` renverra faux pour toute clef ou indice supprimé. (À l'opposé, après la fonction `undef`, la fonction `exists` continue de renvoyer vrai, car la fonction `undef` ne fait que rendre indéfinie la valeur d'un élément, mais ne supprime pas l'élément lui-même.)

Supprimer dans le hachage %ENV modifie l'environnement. Les supprimer dans un hachage qui est lié à un fichier DBM (dans lequel on a les droits en écriture) supprime l'entrée dans ce fichier DBM.

Historiquement, vous pouviez seulement supprimer dans un hachage, mais avec Perl version 5.6 vous pouvez également supprimer dans un tableau. Une suppression dans un tableau a pour conséquence que l'élément à la position spécifiée retourne dans un état non initialisé, mais sans combler le vide, puisque ceci modifierait les positions de toutes les entrées suivantes. Utilisez un splice pour cela. (Toutefois, si vous supprimez le dernier élément d'un tableau, la taille de celui-ci diminuera de un — ou plus, selon la position de l'élément existant précédent s'il y en a.)

EXPR peut être arbitrairement complexe, pourvu que l'opération ultime soit une consultation de hachage ou de tableau :

```
# installation d'un tableau de tableau de hachage
$donjon[$x][$y] = \%proprietes;

# supprimer une propriété dans le hachage
delete $donjon[$x][$y]{OCCUPE}

# supprimer trois propriétés dans le hachage d'un seul coup
delete @{ $donjon[$x][$y] }{ "OCCUPE", "HUMIDE", "ECLAIRE" };

# supprimer la référence à %proprietes dans le tableau
delete $donjon[$x][$y];
```

L'exemple naïf qui suit supprime toutes les valeurs d'un %hachage de manière inefficace :

```
foreach $clef (keys %hachage) {
    delete $hachage{$clef};
}
```

De même que celui-ci :

```
delete @hachage{keys %hachage};
```

Mais ces deux exemples sont plus lents que si l'on assigne simplement au hachage la liste vide ou qu'on le rende indéfini :

```
%hachage = ();    # vide entièrement %hachage
undef %hachage;   # oublie que %hachage a jamais existé
```

Il en va de même pour les tableaux :

```
foreach $indice (0.. $#tableau) {
    delete $tableau[$indice];
}
```

et :

```
delete @tableau[0 .. $#tableau];
```

sont moins efficaces que :

```
@tableau = ();    # vide entièrement @tableau
undef @tableau;   # oublie que @tableau a jamais existé
```

die

$@

```
die LISTE
die
```

Hors d'un `eval`, cette fonction affiche la valeur concaténée de *LISTE* sur `STDERR` et sort avec la valeur courante de `$!` (la variable `errno` de la bibliothèque C). Si `$!` vaut 0, elle sort avec la valeur de `$? >> 8` (qui est le statut du dernier fils récolté depuis un `system`, un `wait`, un `close` sur un pipe, ou une `‘commande‘`). Si `$? >> 8` vaut 0, elle sort avec 255.

À l'intérieur d'un `eval`, la fonction met le message d'erreur qui aurait autrement été produit dans la variable `$@`, puis elle interrompt le `eval`, qui renvoie `undef`. La fonction `die` peut ainsi être utilisée pour lever des exceptions nommées susceptibles d'être exploitées à un plus haut niveau dans le programme. Voir `eval` plus loin dans ce chapitre.

Si *LISTE* est une référence vers un objet unique, cet objet est supposé être un objet exception et il est renvoyé sans modification comme l'exception dans `$@`.

Si *LISTE* est vide et que `$@` contient déjà une chaîne (provenant généralement d'un précédent `eval`), cette chaîne est réutilisée en lui ajoutant `"\t...propagated"`. Ceci est utile pour propager (lever à nouveau) des exceptions :

```
eval { ... };
die unless $@ =~ /Exception attendue/;
```

Si *LISTE* est vide et que `$@` contient déjà un objet exception, la méthode `$@->PROPAGATE` est appelée pour déterminer comment l'exception doit être propagée.

Si *LISTE* est vide ainsi que `$@`, la chaîne `"Died"` est utilisée.

Si la valeur finale de *LISTE* ne se termine pas par un saut de ligne (et que vous ne passez pas d'objet exception), le nom du script courant, le numéro de ligne et le numéro de ligne du fichier d'entrée (s'il existe) sont ajoutés au message, ainsi qu'un saut de ligne. Astuce : quelque fois, le fait d'ajouter `", stopped"` à votre message le rendra plus explicite quand la chaîne `"at nom_de_script line 123"` lui sera ajoutée. Supposons que vous lanciez le script *canasta* ; considérez la différence entre les deux manières suivantes de mourir :

```
die "/usr/games ne vaut rien";
die "/usr/games ne vaut rien, stopped";
```

qui produisent respectivement :

```
/usr/games ne vaut rien at canasta line 123.
/usr/games ne vaut rien, stopped at canasta line 123.
```

Si vous désirez que vos propres messages donnent le nom de fichier et le numéro de ligne, employez les tokens spéciaux `__FILE__` et `__LINE__` :

```
die ’"’, __FILE__, ’", ligne ’, __LINE__, ", hou la honte !\n";
```

La sortie ressemblera alors à :

```
"canasta", ligne 38, hou la honte !
```

Une autre question de style — considérez les exemples équivalents suivants :

```
die "Impossible de faire un cd vers le spool : $!\n"
    unless chdir ’/usr/spool/news’;
```

```
chdir '/usr/spool/news'
    or die "Impossible de faire un cd vers le spool : $!\n";
```

Comme la partie importante est le `chdir`, on préfère généralement la seconde forme.

Voir également `exit`, `warm`, `%SIG` et le module `Carp`.

do (bloc)

```
do BLOC
```

La forme `do` *`BLOC`* exécute la séquence de commandes indiquée par *`BLOC`* et renvoie la valeur de la dernière expression évaluée dans le bloc. Lorsqu'elle est modifiée par une instruction `while` ou `until`, Perl exécute le *`BLOC`* une fois avant de tester la condition de sortie. (Pour d'autres instructions, les modificateurs de boucle testent la condition en premier.) Le `do` *`BLOC`* lui-même *ne* compte *pas* comme une boucle, ainsi les instructions de contrôle de boucle `next`, `last` ou `redo` ne peuvent pas être utilisées pour quitter ou recommencer le bloc. Voir la section *Blocs simples* au chapitre 4, *Instruction et déclaration*, pour contourner ce problème.

do (fichier)

$@

```
do FICHIER
```

La forme `do` *`FICHIER`* utilise la valeur de *`FICHIER`* comme nom de fichier et exécute son contenu comme un script Perl. Son usage premier est (ou plutôt était) d'inclure des sous-programmes depuis une bibliothèque Perl, de sorte que :

```
do 'stat.pl';
```

est identique à :

```
scalar eval 'cat stat.pl'; #'type stat.pl' sur Windows
```

sauf que la première forme est plus efficace, plus concise, elle garde la trace du nom de fichier courant pour les messages d'erreur, cherche dans tous les répertoires du tableau `@INC` et met à jour `%INC` si le fichier est trouvé. (Voir au chapitre 28, *Noms spéciaux*.) Elle diffère également dans le fait que le code évalué avec `do` *`FICHIER`* ne peut pas voir les variables lexicales dans la portée qui l'englobe, à l'inverse du code dans `eval` *`FICHIER`*. Mais elle est pourtant similaire à la seconde en ce qu'elle rescrute le fichier à chaque appel, et il vaut mieux ne pas l'employer dans une boucle à moins que le nom du fichier lui-même change à chaque itération.

Si `do` ne peut pas lire le fichier, elle renvoie `undef` et met l'erreur dans `$!`. Si `do` peut lire le fichier mais pas le compiler, elle retourne `undef` et met un message d'erreur dans `$@`. Si le fichier est compilé avec succès, `do` renvoie la valeur de la dernière expression évaluée.

L'inclusion de modules de bibliothèques (qui ont obligatoirement un suffixe *.pm*) est mieux gérée par les opérateurs `use` et `require`, qui contrôlent les erreurs et lèvent une exception s'il y a un problème. Ils ont également d'autres avantages : ils évitent de dupliquer le chargement, facilitent la programmation orientée objet et fournissent des indications au compilateur sur les prototypes de fonctions.

Mais `do` *`FICHIER`* est toujours utile pour des choses comme la lecture de fichiers de con-

figuration. On peut faire un contrôle d'erreur manuel comme ceci :

```
# lit dans les fichiers de configuration : système d'abord,
# ensuite utilisateur
for $fichier ("/usr/share/projet/defaut.rc",
              "$ENV{HOME}/.programmerc")
{
    unless ($retour = do $fichier) {
        warn "Impossible de scruter $fichier : $@" if $@;
        warn "Impossible de faire do $fichier : $!" unless defined $retour;
        warn "Impossible de lancer $fichier"        unless $retour;
    }
}
```

Un démon qui doit tourner longtemps peut périodiquement examiner la date de son fichier de configuration, et si le fichier a été modifié depuis sa dernière lecture, le démon peut utiliser `do` pour recharger le fichier. Il est plus propre de le faire avec `do` qu'avec `require` ou `use`.

do (sous-programme) $@

```
do SOUS-PROGRAMME(LISTE)
```

La forme `do` *`SOUS-PROGRAMME(LISTE)`* est une forme dépréciée d'appel d'un sous-programme. Une exception est levée si le *`SOUS-PROGRAMME`* n'est pas défini. Voir le chapitre 6, *Sous-programmes*.

dump

```
dump LABEL
dump
```

Cette fonction provoque un *core dump* immédiat. En premier lieu, ceci vous permet d'utiliser le programme *undump*(1) (non fourni) pour transformer le fichier de copie résultant en un binaire exécutable après avoir initialisé toutes les variables au début du programme. Quand le nouveau binaire sera exécuté, il commencera par un `goto` *`LABEL`* (avec toutes les restrictions inhérentes à `goto`). On peut voir le processus complet comme un saut inconditionnel avec un *core dump*, suivi d'une réincarnation. Si *`LABEL`* est omis, le programme repart du début. Attention : tout fichier ouvert au moment de la copie avec `dump`, ne le sera plus au moment de la réincarnation du programme, d'où une possible confusion de la part de Perl. Voyez également l'option de ligne de commande **`-u`** au chapitre 19, *Interface de la ligne de commande*.

Cette fonction est maintenant tout à fait obsolète, en partie parce qu'il est extrêmement difficile de convertir un *core dump* en un binaire dans le cas général, et parce que les divers compilateurs générant du bytecode portable ou du C compilable l'ont surpassé.

Si vous désirez utiliser `dump` pour accélérer vos programmes, allez voir la discussion concernant les questions d'efficacité au chapitre 24, *Usage courant*, ainsi que le compilateur en code natif au chapitre 18, *Compilation*. L'autochargement peut également présenter un certain intérêt, en ce qu'il *semble* accélérer l'exécution.

each

```
each HACHAGE
```

Cette fonction parcours pas à pas un hachage à raison d'une paire clef/valeur à la fois. Appelée dans un contexte de liste, each renvoie une liste à deux éléments composée de la clef et de la valeur de l'entrée suivante d'un hachage, il vous est ainsi possible de circuler dans la liste toute entière. Appelée dans un contexte scalaire, each renvoie seulement la clef de l'entrée suivante du hachage. Quand le hachage est entièrement lu, elle renvoie la liste vide, qui produit une valeur fausse lorsqu'elle est assignée dans un contexte scalaire, comme un test de boucle. L'appel suivant à each redémarrera une nouvelle itération. On l'utilise généralement comme dans l'exemple suivant qui utilise le hachage prédéfini %ENV :

```
while (($clef, $valeur) = each %ENV) {
    print "$clef=$valeur\n";
}
```

En interne, un hachage conserve ses entrées dans un ordre apparemment aléatoire. La fonction each boucle le long de cette séquence car chaque hachage se souvient de l'entrée renvoyée en dernier. L'ordre réel de cette séquence est susceptible de changer dans les prochaines versions de Perl, mais il est garanti qu'il sera identique à celui que renverrait la fonction keys (ou values) avec le même hachage (sans modification).

Il existe un itérateur unique pour chaque hachage, partagé par tous les appels aux fonctions each, keys et values du programme ; on peut le remettre à zéro en lisant tous les éléments du hachage, ou en évaluant keys %hachage ou values %hachage. Si vous ajoutez ou supprimez des élements d'un hachage pendant qu'il y a une itération sur celui-ci, les conséquences ne sont pas bien définies : des entrées peuvent être passées ou dupliquées.

Voir aussi keys, values et sort.

eof

```
eof HANDLE_FICHIER
eof()
eof
```

Cette fonction renvoie vrai si la lecture suivante sur *HANDLE_FICHIER* renverra une fin de fichier, ou si *HANDLE_FICHIER* n'est pas ouvert. *HANDLE_FICHIER* peut être une expression dont la valeur donne le vrai nom du handle de fichier. Un eof sans argument renvoie le statut de fin de fichier pour le dernier fichier lu. Un eof() avec des parenthèses vides teste le handle de fichier ARGV (plus généralement rencontré en tant que handle de fichier nul dans <>). Ainsi, à l'intérieur d'une boucle while (<>), un eof() avec parenthèses ne détectera la fin que du dernier d'un groupe de fichiers. Utilisez eof (sans parenthèses) pour tester *chacun* des fichiers dans une boucle while (<>). Par exemple le code suivant insère des tirets juste avant la dernière ligne du *dernier* fichier :

```
while (<>) {
    if (eof()) {
        print "-" x 30, "\n";
    }
```

```
    print;
}
```

Pour sa part, ce script remet à zéro la numérotation des lignes pour *chaque* fichier d'entrée :

```
# remet à zéro la numérotation des lignes pour chaque fichier d'entrée
while (<>) {
    next if /^\s*#/;     # passe les commentaires
    print "$.\t$_";
} continue {
    close ARGV if eof;   # Et non eof() !
}
```

De même que « $ » dans un programme *sed*, `eof` brille particulièrement dans les numérotations de lignes par zones. Voici un script qui affiche les lignes de `/motif/` à la fin de chaque fichier d'entrée :

```
while (<>) {
    print if /motif/ .. eof;
}
```

Ici, l'opérateur de bascule (`..`) évalue la correspondance du motif à chaque ligne. Jusqu'à ce que le motif corresponde, l'opérateur renvoie faux. Quand la correspondance est enfin trouvée, l'opérateur commence par renvoyer vrai, provoquant ainsi l'affichage des lignes. Quand l'opérateur `eof` renvoie finalement vrai (à la fin du fichier examiné), l'opérateur de bascule se remet à zéro et renvoie de nouveau faux pour le prochain fichier dans `@ARGV`.

Attention : la fonction `eof` lisant un octet puis le remettant dans le flux d'entrée avec *ungetc*(3), elle n'est pas très utile dans un contexte interactif. En fait, les programmeurs Perl expérimentés utilisent rarement `eof`, puisque les divers opérateurs d'entrée se comportent plutôt bien dans les expressions conditionnelles des boucles `while`. Reportez-vous à l'exemple de la description de `foreach` au chapitre 4.

eval

$_ x t

```
eval BLOC
eval EXPR
eval
```

Le mot-clé `eval` dessert en Perl deux buts différents mais néanmoins liés. Ces buts sont représentés par deux formes de syntaxe, eval BLOC et eval EXPR. La première forme intercepte les exceptions à l'exécution (erreurs) qui auraient sinon provoqué une erreur fatale, de la même manière qu'un « bloc try » en C++ ou en Java. La seconde forme compile et exécute de petits morceaux de code à la volée et intercepte également (et c'est bien commode) les exceptions exactement comme la première forme. Mais la seconde forme s'exécute beaucoup plus lentement que la première, puisqu'elle doit analyser la chaîne à chaque fois. D'un autre côté, elle est aussi plus générique. Quelle que soit la forme que vous utilisez, `eval` est la manière standard d'intercepter les exceptions en Perl.

Pour chacune des formes, la valeur renvoyée par `eval` est celle de la dernière expression

évaluée, tout comme dans un sous-programme. Vous pouvez de même utiliser l'opérateur `return` pour renvoyer une valeur au beau milieu de l'`eval`. L'expression générant la valeur de retour est évaluée dans un contexte vide, scalaire ou de liste, selon le contexte de l'`eval` lui-même. Voir `wantarray` pour plus d'informations sur la manière de déterminer un contexte.

S'il se produit une erreur interceptable (y compris une erreur engendrée par l'opérateur `die`), `eval` renvoie `undef` et met le message d'erreur (ou l'objet) dans `$@`. S'il n'y a pas d'erreur, `$@` est garantie contenir la chaîne vide, vous pouvez ainsi détecter les erreurs après coup de manière fiable. Un simple test booléen suffit.

```
eval { ... };     # intercepte les erreurs à l'exécution
if ($@) { ... }   # gère les erreurs
```

La forme eval *`BLOC`* est vérifiée syntaxiquement durant la compilation, ce qui la rend assez performante. (Cela trouble de temps à autre les personnes habituées à la forme lente `eval` *`EXPR`*.) Puisque le code du *`BLOC`* est évalué à la compilation en même temps que le code qui l'entoure, cette forme d'`eval` ne peut pas intercepter les erreurs de syntaxe.

La forme `eval` *`EXPR`* peut quant à elle intercepter les erreurs syntaxiques car elle analyse le code durant l'exécution. (Si l'analyse échoue, elle met l'erreur analysée dans `$@`, comme d'habitude.) Sinon, elle exécute la valeur de *`EXPR`* comme si c'était un petit programme Perl. Le code est exécuté dans le contexte courant du programme Perl, ce qui signifie qu'il peut voir toutes les variables lexicales de la portée qui l'englobe, tout comme les définitions de sous-programmes ou de formats. Le code de l'`eval` est traité en tant que bloc, toute variable de portée locale déclarée à l'intérieur de l'`eval` n'existe donc que le temps de l'`eval`. (Voir `my` et `local`.) Comme dans tout code à l'intérieur d'un bloc, il n'y a pas besoin de point-virgule à la fin.

Voici un simple script Perl. Il demande à l'utilisateur d'entrer une chaîne arbitraire de code Perl, la compile, l'exécute et affiche les éventuelles erreurs qui se sont produites :

```
print "\nEntrer du code Perl : ";

while (<STDIN>) {
    eval;
    print $@;
    print "\nEntrer à nouveau du code Perl : ";
}
```

Voici un programme *rename* pour changer le nom d'une quantité de fichiers en utilisant une expression Perl :

```
#!/usr/bin/perl
# rename - change les noms des fichiers
$op = shift;
for (@ARGV) {
    $ancien = $_;
    eval $op;
    die if $@;
    # la prochaine ligne appelle la fonction interne Perl,
    # non le script du même nom
    rename($ancien, $_) unless $ancien eq $_;
}
```

Vous pouvez utiliser ce programme comme ceci :

```
$ rename 's/\.orig$//'              *.orig
$ rename 'y/A-Z/a-z/ unless /^Make/' *
$ rename '$_ .= ".bad"'             *.f
```

Puisqu'`eval` intercepte des erreurs qui sinon seraient fatales, il est très utile pour déterminer si une fonctionnalité donnée (comme `fork` ou `symlink`) est implémentée.

Comme un `eval` *`BLOC`* est vérifié syntaxiquement à la compilation, toute erreur de syntaxe est reportée en amont. Ainsi, si votre code est invariant et qu'à la fois un `eval` *`EXPR`* et un `eval` *`BLOC`* vous conviennent, la dernière forme est la meilleure. Par exemple :

```
# rend la division par zéro non fatale
eval { $reponse = $a / $b; };    warn $@ if $@;

# idem, mais moins performant si on l'exécute plusieurs fois
eval '$reponse = $a / $b';       warn $@ if $@;

# une erreur de syntaxe à la compilation (non interceptée)
eval { $reponse = };             # MAUVAIS

# une erreur de syntaxe à l'exécution
eval '$reponse =';               # positionne $@
```

Ici, le code du *`BLOC`* doit être un code Perl valide pour passer la phase de compilation. Le code de l'*`EXPR`* n'est pas examiné avant l'exécution, il ne produit donc pas d'erreur avant d'être exécuté.

Le bloc d'un `eval` *`BLOCK`* ne compte *pas* comme une boucle, ainsi les instructions de contrôle de boucle `next`, `last` et `redo` ne peuvent pas être employées pour sortir ou recommencer le bloc.

exec

$! [x]

```
exec CHEMIN LISTE
exec LISTE
```

La fonction `exec` termine le programme courant, exécute une commande externe *et n'en revient jamais !!!* Utilisez `system` au lieu de `exec` si vous voulez reprendre le contrôle après la la fin de la commande. La fonction `exec` échoue et renvoie faux seulement si la commande n'existe pas *et* si elle est exécutée directement et non par l'intermédiaire d'une commande shell de votre système (nous en discuterons plus loin).

S'il n'y a qu'un seul argument scalaire, celui-ci est inspecté pour y chercher des métacaractères du shell. S'il y en a, l'argument est tout entier passé à l'interpréteur de commandes standard du shell (*/bin/sh* sous Unix). Sinon, l'argument est décomposé en mots et exécuté directement dans un souci d'efficacité, puisqu'on économise ainsi la surcharge due au traitement du shell. Cela vous donne également plus de contrôle sur la reprise d'erreur si jamais le programme n'existait pas.

S'il y a plus d'un argument dans la *`LISTE`*, ou si *LISTE* est un tableau avec plus d'une valeur, le shell du système ne sera jamais utilisé. Ceci évite également tout traitement de la commande par le shell. La présence ou l'absence de métacaractères dans les argu-

ments n'affecte pas ce comportement déclenché par la liste, ce qui en fait la forme idéale pour des programmes soucieux de la sécurité, ne voulant pas s'exposer à des fuites potentielles dans le shell.

Cet exemple entraîne le remplacement du programme Perl en train de tourner par le programme `echo`, qui affiche la liste d'arguments courante :

```
exec 'echo', 'Vos arguments sont :', @ARGV;
```

Cet autre exemple montre que vous pouvez `exec`uter un pipeline et non uniquement un seul programme :

```
exec "sort $fichier_sortie | uniq"
    or die "Impossible de faire un sort/uniq : $!\n";
```

`exec` ne revient généralement jamais — si c'est le cas, il retourne toujours faux et vous devez vérifier `$!` pour savoir ce qui s'est mal passé. Remarquez que dans les anciennes versions de Perl, `exec` (et `system`) ne vidaient pas votre tampon de sortie, vous deviez donc activer le vidage du tampon de commande en positionnant `$|` sur un ou plusieurs handles de fichier pour éviter la perte de la sortie dans le cas d'`exec`, ou une sortie désordonnée dans le cas de `system`. La version 5.6 de Perl remédie largement à cette situation.

Lorsque vous demandez au système d'exploitation d'exécuter un nouveau programme à l'intérieur d'un processus existant (comme le fait la fonction `exec` de Perl), vous indiquez au système où se trouve le nouveau programme à exécuter, mais vous donnez aussi au nouveau programme (à travers son premier argument) le nom sous lequel il a été invoqué. Habituellement, le nom que vous donnez est juste une copie de l'endroit où se trouve le programme, mais vous n'y êtes pas nécessairement obligés, puisqu'il y a deux arguments distincts au niveau du langage C. Lorsqu'il ne s'agit pas d'une copie, vous obtenez comme curieux résultat que le nouveau programme croit qu'il est lancé sous un nom qui peut être complètement différent du véritable chemin dans lequel il réside. Cela n'a souvent pas de conséquences pour le programme en question, mais certains programmes y font attention et adoptent une personnalité différente selon ce qu'ils pensent être leur nom. Par exemple, l'éditeur *vi* regarde s'il a été appelé avec « `vi` » ou avec « `view` ». S'il a été invoqué avec « `view` », il se met automatiquement en mode lecture seule, exactement comme s'il avait été appelé avec l'option de la ligne de commande **`-R`**.

Voilà où le paramètre optionnel *`CHEMIN`* de `exec` entre en jeu. Syntaxiquement, il vient se brancher sur la position d'un objet indirect comme le handle de fichier pour `print` ou `printf`. Ainsi, il n'y a pas de virgule après lui car il ne fait pas exactement partie de la liste d'arguments. (En un sens, Perl adopte l'approche opposée du système d'exploitation en ce qu'il suppose que le premier argument est le plus important et qu'il vous laisse modifier le chemin s'il diffère.) Par exemple :

```
$editeur = "/usr/bin/vi";
exec $editeur "view", @fichiers   # active le mode lecture seule
    or die "Impossible d'exécuter $editeur : $!\n";
```

Comme avec tout objet indirect, vous pouvez également remplacer le simple scalaire contenant le nom du programme avec un bloc contenant du code arbitraire, ce qui simplifie l'exemple précédent en donnant :

```
exec { "/usr/bin/vi" } "view" @fichiers # active le mode lecture seule
    or die "Impossible d'exécuter /usr/bin/vi : $!\n";
```

Comme nous l'avons déjà évoqué, exec traite une liste discrète d'arguments comme une indication qu'il doit éviter le traitement du shell. Toutefois, il y a un endroit où vous pourriez être déroutés. L'appel exec (et également system) ne distingue pas un unique argument scalaire d'un tableau ne contenant qu'un seul élément.

```
@args = ("echo surprise");  # une liste à un seul élément
exec @args                  # toujours sujet aux fuites du shell
    or die "exec : $!\n";   #   car @args == 1
```

Pour éviter ceci, vous pouvez utiliser la syntaxe avec *CHEMIN*, en dupliquant exactement le premier argument dans le chemin, ce qui force l'interprétation du reste des arguments à être une liste, même s'il n'y en a qu'un seul :

```
exec { args[0] } @args      # sécurisé même si la liste n'a qu'un argument
    or die "exec @args impossible : $!";
```

La première version, celle avec les parenthèses, lance le programme echo en lui passant « surprise » comme argument. La seconde version ne fait pas la même chose — elle tente de lancer un programme appelé littéralement *echo surprise*, ne le trouve pas (du moins on l'espère) et positionne $! à une valeur non nulle indiquant l'échec.

Comme la fonction exec est le plus souvent utilisée peu après un fork, on suppose que l'on doit passer outre tout ce qui survient normalement lorsqu'un processus Perl se termine. Sous un exec, Perl n'appellera pas les blocs END, ni les méthodes DESTROY associées aux objets. Sinon, vos processus fils se termineraient en faisant le nettoyage alors que vous vous attendiez à ce que ce soit le processus parent qui le fasse. (Nous aimerions que ce soit le cas dans la vie réelle.)

Comme c'est une erreur courante d'utiliser exec au lieu de system, Perl vous en avertit si la prochaine instruction n'est pas die, warn ou exit, lorsqu'il est lancé avec l'option populaire de la ligne de commande **-w**, ou si vous employez le pragma use warnings qw(exec syntax). Si vous voulez vraiment faire suivre un exec d'une autre instruction, vous pouvez utiliser l'un ou l'autre de ces styles pour éviter l'avertissement :

```
exec ('machin')   or print STDERR 'exec machin impossible : $!";
{ exec ('machin') }; print STDERR 'exec machin impossible : $!";
```

Comme dans la seconde ligne ci-dessus, un appel à exec qui est la dernière instruction dans un bloc est exempt de cet avertissement.

Voir aussi system.

exists

```
exists EXPR
```

Cette fonction renvoie vrai si la clef de hachage ou l'indice de tableau spécifié existe dans le hachage ou le tableau. Peu importe que la valeur correspondante soit vraie ou fausse, ou même qu'elle soit définie.

```
print "Vraie\n"   if         $hachage{$clef};
print "Définie\n" if defined $hachage{$clef};
print "Existe\n"  if exists  $hachage{$clef};
print "Vrai\n"    if         $tableau[$indice];
print "Défini\n"  if defined $tableau[$indice];
print "Existe\n"  if exists  $tableau[$indice];
```

Un élément ne peut être vrai que s'il est défini, et ne peut être défini que s'il existe, mais l'inverse n'est pas forcément valable.

EXPR peut être arbitrairement complexe, tant que l'opération finale est une consultation de clef de hachage ou d'indice de tableau.

```
if (exists $hachage{A}{B}{$clef}) { ... }
```

Bien que le dernier élément ne vienne pas à exister spontanément seulement parce qu'on a testé son existence, c'est le cas de ceux que l'on fait intervenir. Ainsi, `$$hachage{"A"}` et `$hachage{"A"}->{"B"}` se mettent tous deux à exister. Ce n'est pas une fonctionnalité de `exists` en soi ; cela se produit partout où l'opérateur flèche est utilisé (explicitement ou implicitement) :

```
undef $ref if (exists $ref->"Une clef"}) { }
print $ref; # affiche HASH(0x80d3d5c)
```

Même si l'élément `"Une clef"` ne se met pas à exister, la variable `$ref`, indéfinie auparavant, vient tout à coup contenir un hachage anonyme. C'est une instance surprise d'*autovivification* de ce qui n'apparaissait pas au premier — ou même au second — coup d'œil comme un contexte lvalue. Ce comportement sera agréablement corrigé dans une prochaine version. Autrement, vous pouvez emboîter vos appels :

```
if ($ref and exists $ref->[$x] and
exists $ref->[$x][$y] and
exists $ref->[$x][$y]{$clef} and
exists $ref->[$X][$y]{$clef}[2] ) { }
```

Si *EXPR* est le nom d'un sous-programme, la fonction `exists` renverra vrai si celui-ci est a été déclaré, même s'il n'a pas encore été défini. L'exemple suivant affiche seulement « `Existe` » :

```
sub couic;
print "Existe\n" if exists  &couic;
print "Défini\n" if defined &couic;
```

L'utilisation d'`exists` avec le nom d'un sous-programme peut être utile pour un sous-programme `AUTOLOAD` qui a besoin de savoir si un paquetage particulier veut qu'un sous-programme particulier soit défini. Le paquetage peut indiquer ceci en déclarant un entête de `sub` comme `couic`.

exit

```
exit EXPR
exit
```

Cette fonction évalue *EXPR* comme un entier et sort immédiatement avec cette valeur comme statut d'erreur final du programme. Si *EXPR* est omise, la fonction sort avec le statut `0` (signifiant « aucune erreur »). Voici un fragment qui permet à un utilisateur de sortir du programme en entrant `x` ou `X`.

```
$rep = <STDIN>;
exit if $rep =~ /^[Xx]/;
```

Vous ne devriez pas utiliser `exit` pour quitter un sous-programme s'il y a la moindre chance que quelqu'un veuille intercepter l'erreur qui s'est produite. Utilisez plutôt `die`,

qui peut être intercepté par un `eval`. Ou alors, utilisez l'une des sur-couches de `die` du module `Carp`, comme `croak` ou `confess`.

Nous avons dit que la fonction `exit` sortait immédiatement, mais c'était un mensonge d'arracheur de dents. Elle sort aussitôt que possible, mais elle appelle tout d'abord les routines `END` définies afin les gérer les événements de fin (*N.d.T.* : *at-exit*). Ces routines ne peuvent pas annuler la terminaison, bien qu'elles puissent changer la valeur de retour éventuelle en positionnant la variable `$?`. De la même manière, toute classe qui définit une méthode `DESTROY` peut invoquer cette méthode au nom de tous ses objets avant que le programme réel ne finisse. Si vous avez vraiment besoin d'outrepasser le traitement avant la terminaison, vous pouvez appeler les fonctions `_exit` du module `POSIX` pour éviter toutes les opérations de `END` et des destructeurs. Et si `POSIX` n'est pas disponible, vous pouvez faire un `exec "/bin/false"` ou quelque chose de ce genre.

exp

$_

```
exp EXPR
exp
```

Cette fonction renvoie *e* à la puissance *EXPR*. Pour obtenir la valeur de *e*, utilisez juste `exp(1)`. Pour effectuer une exponentiation générale de différentes bases, utilisez l'opérateur ** que nous avons emprunté au FORTRAN :

```
use Math::Complex;
print -exp(1) ** (i * pi); # affiche 1
```

fcntl

$! X ARG X RO X T X U

```
fcntl HANDLE_FICHIER, FONCTION, SCALAIRE
```

Cette fonction appelle les fonctions de contrôle de fichier de votre système d'exploitation, documentées dans la page de man de *fnctl*(2). Avant d'appeler `fcntl`, vous devrez probablement tout d'abord écrire :

```
use Fcntl;
```

pour charger les définitions correctes des constantes.

SCALAIRE sera lu ou écrit (ou les deux) selon la *FONCTION*. Un pointeur sur la valeur de chaîne de *SCALAIRE* sera passé comme troisième argument de l'appel véritable à *fcntl*. (Si *SCALAIRE* n'a pas de valeur chaîne mais une valeur numérique, celle-ci sera passée directement au lieu d'un pointeur sur la valeur chaîne.) Voir le module `Fcntl` pour une description des valeurs les plus communes qu'il est possible d'employer pour *FONCTION*.

La fonction `fcntl` lèvera une exception si elle est utilisée sur un système qui n'implémente pas *fcntl*(2). Sur les systèmes qui au contraire l'implémentent, vous pouvez faire des choses comme modifier les flags `close-on-exec` (si vous ne désirez pas jouer avec la variable `$^F` (`$SYSTEM_FD_MAX`)), modifier les flags d'entrées/sorties non bloquantes, émuler la fonction *lockf* (3) ou s'arranger pour recevoir un signal `SIGIO` lorsqu'il y a des entrées/sorties en attente.

Voici un exemple où l'on rend non bloquant au niveau du système, un handle de fichier nommé `DISTANT`. Ceci fait que toute opération d'entrée revient immédiatement si rien n'est disponible lors d'une lecture dans un pipe, une socket ou une ligne série qui autre-

ment aurait bloqué. De même, les opérations de sortie qui bloqueraient normalement, renvoient un statut d'échec à la place. (Pour celles-ci, vous feriez mieux de négocier également `$|`.)

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

$flags = fcntl(DISTANT, F_GETFL, 0)
            or die "Impossible de lire les flags pour la socket : $!\n";
$flags = fcntl(DISTANT, F_SETFL, $flags | O_NONBLOCK)
            or die "Impossible de modifier les flags de la sockets : $!\n";
```

La valeur de retour de `fcntl` (et de `ioctl`) se comporte comme ceci :

L'appel système renvoie	Perl renvoie
`-1`	`undef`.
`0`	La chaîne « `0 but true` ».
autre chose	Le même nombre.

Perl rend donc vrai pour un succès et faux pour un échec, tout en vous laissant facilement déterminer la véritable valeur renvoyée par le système d'exploitation :

```
$valret = fcntl() || -1;
printf "fcntl a vraiment renvoyé %d\n", $valret;
```

Ici, même la chaîne « 0 `but true` » affiche 0, grâce au format `%d`. Cette chaîne est vraie dans un contexte booléen et `0` dans un contexte numérique. (Par bonheur, elle est également exempte des avertissements habituels pour les conversions numériques inadaptées.)

fileno


```
fileno HANDLE_FICHIER
```

Cette fonction renvoie le descripteur de fichier sous-jacent d'un handle de fichier. Si ce handle de fichier n'est pas ouvert, `fileno` renvoie `undef`. Un *descripteur de fichier* est un petit entier positif comme 0 ou 1, alors que les handles de fichiers comme `STDIN` ou `STDOUT` sont des symboles. Malheureusement, le système d'exploitation ne connaît pas vos sympathiques symboles. Il ne pense aux fichiers ouverts qu'en terme de ces petits numéros de fichiers et bien que Perl fasse automatiquement la traduction pour vous, vous devrez parfois connaître le véritable descripteur de fichier.

Ainsi, par exemple, la fonction `fileno` est utile pour construire des masques de bits pour `select` et pour passer certains appels système obscurs si *syscall*(2) est implémenté. Elle sert également à vérifier que la fonction `open` vous a donné le descripteur de fichier que vous vouliez et également à déterminer si deux handles de fichiers utilisent le même descripteur de fichier système.

```
if (fileno(CECI) == fileno(CELA)) {
    print "CECI et CELA sont des duplicata\n";
}
```

Si *HANDLE_FICHIER* est une expression, sa valeur est prise comme un handle de fichier

indirect, généralement son nom ou une référence à quelque chose qui ressemble à un objet handle de fichier.

Attention : ne comptez pas sur une association entre un handle de fichier Perl et un descripteur de fichier numérique qui soit permanente tout au long de la vie du programme. Si un fichier a été fermé et rouvert, le descripteur de fichier peut changer. Perl s'efforce d'assurer que certains descripteurs de fichiers ne sont pas perdus si un `open` sur eux échoue, mais il ne fait cela que pour les descripteurs de fichiers qui ne dépassent pas la valeur courante de la variable spéciale `$^F` (`$SYSTEM_MAX_FD`) (par défaut, 2). Bien que les handles de fichiers `STDIN`, `STDOUT` et `STDERR` démarrent avec les descripteurs de fichiers 0, 1 et 2 (la convention standard Unix), même ceux-ci peuvent changer si vous commencez à les fermer et les ouvrir sans prendre garde. Ceci dit, vous ne pouvez pas avoir d'ennuis avec 0, 1 et 2 aussi longtemps que vous les rouvrez toujours immédiatement après les avoir fermés. La règle de base sur les systèmes Unix est de prendre le descripteur disponible le plus bas possible, et ce sera celui que vous venez juste de fermer.

flock

$! X ARG X U

```
flock HANDLE_FICHIER, OPÉRATION
```

La fonction `flock` est l'interface portable de Perl pour le verrouillage de fichiers, bien qu'elle ne verrouille un fichier que dans son intégralité et non des enregistrements. La fonction gère les verrous sur le fichier associé à *`HANDLE_FICHIER`*, en renvoyant vrai pour un succès et faux sinon. Pour éviter la possibilité de perdre des données, Perl vide le tampon de votre *`HANDLE_FICHIER`* avant de le verrouiller ou de le déverrouiller. Perl peut implémenter son `flock` avec *flock*(2), *fcntl*(2), *lockf* (3) ou des mécanismes de verrouillage spécifiques à d'autres plate-formes, mais si aucun d'entre eux n'est disponible, appeler `flock` lève une exception. Voir la section *Verrouillage de fichier* au chapitre 16.

`OPÉRATION` peut valoir `LOCK_SH`, `LOCK_EX` ou `LOCK_UN`, combinés éventuellement par un OU avec `LOCK_NB`. Ces constantes valent généralement `1`, `2`, `8` et `4`, mais vous pouvez employer les noms symboliques si vous les importez depuis le module `Fcntl`, soit individuellement, soit par groupe en utilisant le tag `:flock`.

`LOCK_SH` demande un verrou partagé, il est donc généralement utilisé pour les lectures. `LOCK_EX` demande un verrou exclusif, il est donc généralement utilisé pour les écritures. `LOCK_UN` relâche un verrou demandé auparavant ; fermer le fichier relâche également ses verrous. Si le bit `LOCK_NB` est employé avec `LOCK_SH` ou `LOCK_EX`, `flock` rend la main immédiatement au lieu d'attendre un verrou indisponible. Vérifiez le statut de retour pour voir si vous avez obtenu le verrou que vous avez demandé. SI vous n'utilisez pas `LOCK_NB`, vous pouvez attendre indéfiniment l'accès au verrou.

Un autre aspect difficile, mais général, de `flock` est que ses verrous sont *simplement consultatifs*. Les verrous discrets sont plus flexibles mais offrent moins de garanties que les verrous obligatoires. Cela signifie que les fichiers verrouillés avec `flock` peuvent être modifiés par les programmes qui n'utilisent pas `flock`. Les voitures qui s'arrêtent aux feux rouges s'entendent bien entre elles, mais pas avec celles qui ne s'arrêtent pas aux feux rouges. Conduisez sur la défensive.

Certaines implémentations de `flock` ne peuvent pas verrouiller quoique ce soit à travers un réseau. Bien qu'en théorie vous puissiez utiliser `fcntl`, plus spécifique au système, pour cela, le jury (ayant délibéré sur le cas pendant à peu près une décennie) est tou-

jours indécis pour déterminer si c'est une chose digne de confiance ou non.

Voici un exemple qui ajoute un message à la fin d'une boîte aux lettres sur les systèmes Unix qui utilisent *flock*(2) pour verrouiller les boîtes aux lettres :

```
use Fcntl qw/:flock/      # importer les constantes FLOCK_*
sub mon_verrou {
    flock(MBOX, LOCK_EX)
        or die "Impossible de verrouiller la boîte aux lettres : $!";
        # dans le cas ou quelqu'un ajoutait quelque chose alors que
        #nous   étions en train d'attendre et que notre tampon stdio ne
        # soit pas synchronisé
    seek(MBOX, 0, 2)
        or die "Impossible de se positionner à la fin de la boîte aux
                lettres : $!";
}

open(MBOX, "> >/usr/spool/mail/$ENV{'USER'}")
    or die "Impossible d'ouvrir la boîte aux lettres : $!";

mon_verrou();
print MBOX $msg, "\n\n";
close MBOX
    or die "Impossible de fermer la boîte aux lettres : $!";
```

Sur les systèmes qui supportent un appel système *flock*(2) réel, les verrous sont hérités à travers les appels à `fork`. Les autres implémentations ne sont pas aussi chanceuses et sont susceptibles de perdre les verrous à travers les forks. Voir également le module `DB_File` au chapitre 32 pour d'autres exemples de `flock`.

fork

$! x 0

```
fork
```

Cette fonction crée deux processus à partir d'un seul en invoquant l'appel système *fork*(2). En cas de réussite, la fonction renvoie l'ID du nouveau processus fils au processus parent et 0 au processus fils. Si le système ne dispose pas des ressources suffisantes pour allouer un nouveau processus, l'appel échoue et retourne `undef`. Les descripteurs de fichiers (et parfois les verrous sur ces descripteurs) sont partagés, alors que tout le reste est copié — ou moins paraît l'être.

Dans les versions de Perl antérieures à 5.6, les tampons non vidés le restent dans les deux processus, ce qui veut dire qu'il vous faut parfois positionner `$|` sur un ou plusieurs handles de fichiers à l'avance pour éviter des sorties dupliquées.

Voici une façon de blinder presque totalement le lancement d'un processus fils tout en testant les erreurs de type « fork impossible » :

```
use Errno qw(EAGAIN);
FORK: {
    if ($pid = fork) {
        # on est ici dans le père
        # le pid du processus fils est accessible par $pid
    }
```

```
        elsif (defined $pid) { # $pid vaut ici 0 s'il défini
        # on est ici dans le fils
        # le pid du processus père est accessible par getppid
    }
        elsif ($! == EAGAIN) {
        # EAGAIN est l'erreur de fork susceptible d'être corrigée
        sleep 5;
        redo FORK;
    }
        else {
        # erreur de fork étrange
        die "fork impossible :$!\n";
    }
}
```

Ces précautions ne sont pas nécessaires pour des opérations qui font un *fork*(2) implicite, comme `system`, les apostrophes inverses ou l'ouverture d'un processus comme un handle de fichier, car Perl réessaye automatiquement un fork en cas d'échec dans ces cas. Faites très attention à terminer le code du fils par un `exit`, faute de quoi le fils pourrait quitter par inadvertance le bloc conditionnel et commencer à exécuter du code prévu pour le seul processus père.

Si vous `fork`ez sans même attendre vos enfants, vous accumulerez les zombies (les processus terminés qui n'ont toujours pas de parents qui les attendent). Sur certains systèmes, vous pouvez éviter ceci en positionnant `$SIG{CHLD}` à `"IGNORE"` ; sur la plupart, vous devrez attendre avec `wait` vos enfants moribonds. Vous en trouverez des exemples dans la description de la fonction `wait`, ou dans la section *Signaux* du chapitre 16.

Si un fils `fork`é hérite des descripteurs fichiers systèmes comme `STDIN` ou `STDOUT` qui sont connectés à un pipe distant ou à une socket, il se peut que vous deviez les redirigez vers */dev/null* dans le fils. Ceci car même lorsque le processus père se termine, le fils vivra avec ses copies de ces handles de fichiers. Le serveur distant (mettons comme un script CGI ou une tâche lancée en arrière plan par un shell distant) semblera bloquer car il attendra que toutes les copies soient fermées. Rouvrir les handles de fichiers systèmes en les dirigeant vers autre chose corrige ceci.

Sur la plupart des systèmes supportant *fork*(2), on a porté une attention toute particulière à ce que la fonction soit performante (par exemple, utilisant une technologie de copie en écriture sur les pages de données), ce qui en fait le paradigme dominant de ces dernières décennies pour le multitâche. La fonction `fork` est susceptible de ne pas être implémentée efficacement, voire de ne pas être implémentée du tout, sur les systèmes qui ne ressemblent pas à Unix. Par exemple, Perl 5.6 émule un fork approprié sur les systèmes de Microsoft, mais sans assurance sur les performances à ce jour. Vous pouvez avoir plus de chances ici avec le module `Win32::Process`.

format

```
format NOM =
    ligne d'image
    liste de valeurs
    ...
.
```

Cette fonction déclare une séquence nommée de lignes d'images (avec les valeurs associées) à l'usage de la fonction `write`. Si *`NOM`* est omis, le nom par défaut est `STDOUT`, qui est celui du format par défaut pour le handle de fichier `STDOUT`. Puisque, comme pour une déclaration de `sub`, il s'agit d'une déclaration globale au paquetage se produisant à la compilation, toutes les variables utilisées dans la liste de valeurs doivent être visibles au point de la déclaration du format. C'est-à-dire que les variables à portée lexicale doivent être auparavant déclarées dans le fichier, alors que les variables à portée dynamique peuvent n'être initialisées qu'au moment où l'on appelle `write`. En voici un exemple (qui suppose que nous avons déjà calculé `$cout` et `$quantite` :

```
my $chn = "widget";               # Une variable de portée lexicale.

format Jolie_Sortie =
Test: @<<<<<<<< @||||| @>>>>>
      $chn,     $%,    '$' . int($num)
.

local $~ = "Jolie_Sortie";        # Sélection du format.
local $num = $cout * $quantite;   # Variable de portée dynamique.

write;
```

Tout comme les handles de fichier, les noms de format sont de simples identificateurs qui existent dans une table de symboles (paquetage) et peuvent être pleinement qualifiés par le nom du paquetage. Dans les typeglobs des entrées d'une table de symboles, les formats résident dans leur propre espace de noms, qui est distinct des handles de fichiers et de répertoires, des scalaires, des tableaux, des hachages ou des sous-programmes. Cependant, comme pour ces six autres types, un format nommé `Nimportequoi` serait aussi affecté par un `local` sur le typeglob `*Nimportequoi`. En d'autres termes, un format n'est qu'un machin de plus, contenu dans un typeglob, indépendant des autres machins.

La section *Variables de format* du chapitre 7, *Formats* contient de nombreux détails et des exemples d'utilisation. Le chapitre 28 décrit les variables internes spécifiques aux formats, et les modules `English` et `FileHandle` y fournissent un accès aisé.

formline

```
formline IMAGE, LISTE
```

Il s'agit d'une fonction interne utilisée par les `format`s, bien que vous puissiez également l'appeler vous-même. Elle formate une liste de valeurs d'après le contenu d'*`IMAGE`*, en plaçant la sortie dans l'accumulateur de sortie, `$^A` (ou `$ACCUMULATOR` si vous utilisez le module `English`). Finalement, au moment d'un `write`, le contenu de `$^A` est écrit vers un handle de fichier, mais vous pouvez aussi bien lire `$^A` par vous-même pour ensuite le remettre à `""`. Typiquement, un format lance un `formline` par ligne de format, mais la fonction `formline` elle-même ne fait pas attention au nombre de sauts de ligne inclus dans l'*`IMAGE`*. Cela veut dire que les tokens `~` et `~~` traitent *`IMAGE`* comme une seule ligne. Il peut donc être nécessaire d'employer plusieurs `formline`s pour implémenter un unique format d'enregistrement, comme le compilateur de format le fait en interne.

Attention si vous mettez des doubles apostrophes autour de l'image, car un caractère `@`

peut être confondu avec le début d'un nom de tableau. Voir le chapitre 7, *Formats*, pour des exemples d'utilisation.

getc

T X ARG

```
getc HANDLE_FICHIER
getc
```

Cette fonction renvoie le caractère suivant depuis le fichier d'entrée attaché à *HANDLE_FICHIER*. Elle renvoie `undef` à la fin du fichier ou si une erreur d'entrée/sortie se produit. Si *HANDLE_FICHIER* est omis, la fonction lit depuis `STDIN`.

Cette fonction est quelque peu lente, mais se révèle parfois utile pour les entrées d'un seul caractère (octet, réellement) au clavier — pourvu que vous vous arrangiez pour que les entrées depuis le clavier ne passent pas par un tampon. Cette fonction demande une entrée non tamponnée depuis la bibliothèque standard d'entrée/sortie. Malheureusement, la bibliothèque standard d'entrée/sortie n'est pas assez standard pour fournir une manière portable de dire au système d'exploitation sous-jacent de donner au système d'entrée/sortie des entrées non tamponnées depuis le clavier. Pour accomplir ceci, vous devez être un peu plus habile et utiliser un style dépendant du système d'exploitation. Sous Unix, vous pourriez dire ceci :

```
if ($BSD_STYLE) {
    system "stty cbreak </dev/tty >/dev/tty 2>&1";
} else {
    system "stty", "-icanon", "eol", "\001";
}

$clef = getc;

if ($BSD_STYLE) {
    system "stty -cbreak </dev/tty >/dev/tty 2&1";
} else {
    system "stty", "icanon", "eol", "^@"; # ASCII NUL
}
print "\n";
```

Ce code met le prochain caractère (octet) frappé sur le terminal dans la chaîne `$clef`. Si votre programme *stty* possède des options comme `cbreak`, vous devrez utiliser le code où `$BSD_STYLE` est vrai. Sinon, vous devrez utiliser le code où il est faux. Déterminer les options de *stty*(1) est laissé comme exercice au lecteur.

Le module `POSIX` en fournit une version plus portable grâce à la fonction `POSIX::getattr`. Voir aussi le module `Term::ReadKey` dans votre site CPAN le plus proche pour une approche plus portable et plus flexible.

getgrent

X U

```
getgrent
setgrent
endgrent
```

Ces routines parcourent votre fichier */etc/group* (ou peut-être le fichier */etc/group* de quel-

qu'un d'autre, s'il provient d'un serveur quelque part). La valeur renvoyée par `getgrent` dans un contexte de liste est :

```
($nom, $mot_de_passe, $gid, $membres)
```

où `$membres` contient une liste des noms de login des membres du groupe séparés par des espaces. Pour mettre en œuvre un hachage traduisant les noms de groupes en GID, écrivez ceci :

```
while (($nom, $mot_de_passe, $gid) = getgrent) {
    $gid{$nom} = $gid;
}
```

Dans un contexte scalaire, `getgrent` renvoie seulement le nom du groupe. Le module standard `User::grent` supporte une interface orientée *par nom* pour cette fonction. Voir *getgrent*(3).

getgrid

```
getgrid GID
```

Cette fonction recherche une entrée de fichier de groupe par le numéro du groupe. La valeur renvoyée dans un contexte de liste est :

```
($nom, $mot_de_passe, $gid, $membres)
```

où `$membres` contient une liste des noms de login des membres du groupe séparés par des espaces. Si vous voulez faire cela de façon répétitive, il vaut mieux utiliser un hachage comme cache en utilisant `getgrent`.

Dans un contexte scalaire, `getgrid` renvoie seulement le nom du groupe. Le module standard `User::grent` supporte une interface orientée *par nom* pour cette fonction. Voir *getgrid*(3).

getgrnam

```
getgrid NOM
```

Cette fonction recherche une entrée de fichier de groupe par le nom du groupe. La valeur renvoyée dans un contexte de liste est :

```
($nom, $mot_de_passe, $gid, $membres)
```

où `$membres` contient une liste des noms de login des membres du groupe séparés par des espaces. Si vous voulez faire cela de façon répétitive, il vaut mieux utiliser un hachage comme cache en utilisant `getgrent`.

Dans un contexte scalaire, `getgrnam` renvoie seulement le nom du groupe. Le module standard `User::grent` supporte une interface orientée « par nom » pour cette fonction. Voir *getgrnam*(3).

gethostbyaddr

```
gethostbyaddr ADR, TYPE_ADR
```

Cette fonction convertit des adresses en noms (et en adresses alternatives). *ADR* doit être une adresse réseau binaire empaquetée et *TYPE_ADR* en pratique doit être normalement

AF_INET (du module Socket). La valeur de retour dans un contexte de liste est :

```
($nom, $alias, $type_adr, $longueur, @adrs) =
    gethostbyaddr($adresse_binaire_empaquetée, $type_adr);
```

où @adrs est une liste d'adresses brutes. Dans le domaine de l'Internet, chaque adresse est (historiquement) longue de 4 octets et peut être découpée en écrivant quelque chose comme ceci :

```
($a, $b, $c, $d) = unpack('C4', $adrs[0]);
```

Ou alors, vous pouvez directement la convertir dans la notation de vecteur pointé avec le modificateur v de sprintf :

```
$points = sprintf "%vd", $adrs[0];
```

La fonction inet_ntoa du module Socket est très utile pour produire une version affichable. Cette approche deviendra importante si et quand nous parviendrons à passer à IPv6.

```
use Socket;
$adresse_affichable = inet_ntoa($addrs[0]);
```

Dans un contexte scalaire, gethostbyaddr renvoie seulement le nom de l'hôte.

Pour obtenir une *ADR* à partir d'un vecteur pointé, écrivez ceci :

```
use Socket;
$adr_ip = inet_aton("127.0.0.1");     # localhost
$nom_officiel = gethostbyaddr($adr_ip, AF_INET);
```

De manière intéressante, vous pouvez avec la version 5.6 de Perl vous passer de inet_aton et employer la nouvelle notation v-chaîne, qui a été inventée pour les numéros de versions, mais il est apparu qu'elle fonctionnait aussi bien pour les adresses IP :

```
$adr_ip = v127.0.0.1;
```

Voir la section *Sockets* au chapitre 16 pour plus d'exemples. Le module Net::hostent supporte une interface orientée « par nom » pour cette fonction. Voir *gethostbyaddr*(3).

gethostbyname

```
gethostbyname NOM
```

Cette fonction convertit un nom d'hôte réseau en son adresse correspondante (et en d'autres noms). La valeur de retour dans un contexte de liste est :

```
($nom, $alias, $type_adr, $longueur, @adrs) =
    gethostbyname($nom_hote_distant);
```

où @adrs est une liste d'adresses brutes. Dans le domaine de l'Internet, chaque adresse est (historiquement) longue de 4 octets et peut être découpée en écrivant quelque chose comme ceci :

```
($a, $b, $c, $d) = unpack('C4', $adrs[0]);
```

Vous pouvez directement la convertir dans la notation de vecteur avec le modificateur v de sprintf :

```
$points = sprintf "%vd", $adrs[0];
```

Dans un contexte scalaire, gethostbyname renvoie uniquement l'adresse de l'hôte :

```
use Socket;
$adr_ip = gethostbyname($hote_distant);
printf "%s a comme adresse %s\n",
    $hote_distant, inet_ntoa($adr_ip);
```

Voir la section *Sockets* au chapitre 16 pour une autre approche. Le module Net::hostent supporte une interface orientée *par nom* pour cette fonction. Voir également *gethostbyname*(3).

gethostent

```
gethostent
sethostent RESTE_OUVERT
endhostent
```

Ces fonctions parcourent votre fichier */etc/hosts* et renvoient chaque entrée une par une. La valeur de retour de gethostent dans un contexte de liste est :

```
($nom, $alias, $longueur, @adrs)
```

où @adrs est une liste d'adresses brutes. Dans le domaine Internet, chaque adresse est longue de 4 octets et peut être découpée en écrivant quelque chose comme ceci :

```
($a, $b, $c, $d) = unpack('C4', $adrs[0]);
```

Les scripts qui utilisent gethostent ne doivent pas être considérés comme portables. Si une machine utilise un serveur de noms, elle interrogera la quasi-totalité de l'Internet pour tenter de satisfaire une requête portant sur toutes les adresses de toutes les machines de la planète. Voir *gethostent*(3) pour d'autres détails.

Le module Net::hostent supporte une interface orientée *par nom* pour cette fonction.

getlogin

```
getlogin
```

Cette fonction renvoie le nom de login courant s'il est trouvé. Sur les systèmes Unix, il est lu à partir du fichier *utmp*(5). Si la fonction renvoie faux, utilisez getpwuid à la place. Par exemple :

```
$login = getlogin() || (getpwuid($<))[0]) || "Intrus !!";
```

getnetbyaddr

```
getnetbyaddr ADR, TYPE_ADR
```

Cette fonction convertit une adresse réseau vers le nom ou les noms de réseau correspondants. La valeur renvoyée dans un contexte de liste est :

```
use Socket;
($nom, $alias, $type_adr, $reseau) = getnetbyaddr(127, AF_INET);
```

Dans un contexte scalaire, getnetbyaddr renvoie seulement le nom du réseau. Le module Net::netent supporte une interface orientée *par nom* pour cette fonction. Voir *getnetbyaddr*(3).

getnetbyname

```
getnetbyname NOM
```

Cette fonction convertit un nom de réseau vers son adresse réseau correspondante. La valeur de retour dans un contexte de liste est :

```
($nom, $alias, $type_adr, $reseau) = getnetbyname("loopback");
```

Dans un contexte scalaire, `getnetbyname` renvoie seulement l'adresse du réseau. Le module `Net::netent` supporte une interface orientée *par nom* pour cette fonction. Voir *getnetbyname*(3).

getnetent

```
getnetent
setnetent RESTE_OUVERT
endnetent
```

Ces fonctions parcourent votre fichier `/etc/networks`. La valeur renvoyée dans un contexte de liste est :

```
($nom, $alias, $type_adr, $reseau) = getnetent();
```

Dans un contexte scalaire, `getnetent` renvoie seulement le nom du réseau. Le module `Net::netent` supporte une interface orientée *par nom* pour cette fonction. Voir *getnetent*(3).

Le concept de nom de réseau semble un peu désuet de nos jours ; la plupart des adresses IP étant sur des sous-réseaux innommés (et innommables).

getpeername

```
getpeername SOCKET
```

Cette fonction l'adresse socket empaquetée de l'autre extrémité de la connexion *SOCKET*. Par exemple :

```
use Socket;
$son_adr_sock     = getpeername SOCK;
($port, $son_adr) = sockaddr_in($son_adr_sock);
$son_nom_hote     = gethostbyaddr($son_adr, AF_INET);
$son_adr_chaine   = inet_ntoa($son_adr);
```

getpgrp

```
getpgrp PID
```

Cette fonction renvoie l'identifiant du groupe de processus courant pour le *PID* spécifié (employez un *PID* égal à `0` pour le processus courant). L'invocation de `getpgrp` lèvera une exception sur une machine n'implémentant pas *getpgrp*(2). Si *PID* est omis, la fonction renvoie le groupe de processus du processus courant (ce qui est identique à un *PID* de `0`). Sur les systèmes qui implémentent cet opérateur avec l'appel système POSIX *getpgrp*(2), *PID* doit être omis ou, s'il est fourni, doit être égal à `0`.

getppid

```
getppid
```

Cette fonction renvoie l'identifiant du processus parent. Sur un système UNIX typique, si l'ID du processus père se change en 1, c'est que ce dernier est mort et que le processus courant a été adopté par le programme *init*(8).

getpriority

```
getpriority QUOI, QUI
```

Cette fonction renvoie la priorité courante d'un processus, d'un groupe de processus ou d'un utilisateur. Voir *getpriority*(2). Son invocation lèvera une exception sur une machine où *getpriority*(2) n'est pas implémentée.

Le module `BSD::Resource` de CPAN fournit une interface plus pratique, comprenant les constantes symboliques `PRIO_PROCESS`, `PRIO_PGRP` et `PRIO_USER` qu'il faut fournir pour l'argument *`QUOI`*. Bien qu'elles soient généralement égales à `0`, `1` et `2` respectivement, vous ne savez vraiment pas ce qui peut se passer aux sombres confins des fichiers *#include* du C.

Une valeur de `0` pour *`QUI`* signifie le processus, le groupe de processus ou l'utilisateur courant, ainsi pour obtenir la priorité du processus en cours, utilisez :

```
$prio_cour = getpriority(0, 0);
```

getprotobyname

```
getprotobyname NOM
```

Cette fonction convertit un nom de protocole vers son numéro correspondant. La valeur renvoyée dans un contexte de liste est :

```
($nom, $alias, $numero_protocole) = getprotobyname("tcp");
```

Lorsqu'on l'appelle dans un contexte scalaire, `getprotobyname` renvoie seulement le numéro de protocole. Le module `Net::proto` supporte une interface orientée *par nom* pour cette fonction. Voir *getprotobyname*(3).

getprotobynumber

```
getprotobynumber NUMERO
```

Cette fonction convertit un numéro de protocole vers son nom correspondant. La valeur renvoyée dans un contexte de liste est :

```
($nom, $alias, $numero_protocole) = getprotobynumber(6);
```

Lorsqu'on l'appelle dans un contexte scalaire, `getprotobynumber` renvoie seulement le nom de protocole. Le module `Net::proto` supporte une interface orientée *par nom* pour cette fonction. Voir *getprotobynumber*(3).

getprotoent

```
getprotoent
setprotoent RESTE_OUVERT
endprotoent
```

Ces fonctions parcourent votre fichier */etc/protocols*. Dans un contexte de liste, la valeur renvoyée par `getprotoent` est :

```
($nom, $alias, $numero_protocole) = getprotoent();
```

Lorsqu'on l'appelle dans un contexte scalaire, `getprotoent` renvoie seulement le numéro du protocole. Le module `Net::protoent` supporte une interface orientée *par nom* pour cette fonction. Voir *getprotent(3)*.

getpwent

```
getpwent
setpwent
endpwent
```

Ces fonctions parcourent conceptuellement votre fichier */etc/passwd*, bien que ceci puisse impliquer le fichier */etc/shadow* si vous êtes le super-utilisateur et que vous utilisez les mots de passe « shadow », NIS (ex-Pages Jaunes, Yellow Pages, YP) ou NIS+. La valeur renvoyée dans un contexte de liste est :

```
($nom,$mot_de_passe,$uid,$gid,$quota,$commentaire,$nom_reel,$rep,$shell)=
    getpwent();
```

Certaines machines peuvent utiliser les champs `quota` et `commentaire` à d'autres fins que leurs noms ne le laissent supposer, mais les champs restants seront toujours les mêmes. Pour mettre en œuvre un hachage de traduction des noms de login en UID, écrivez ceci :

```
while (($nom, $mot_de_passe, $uid) = getpwent()) {
    $uid{$nom} = $uid;
}
```

Dans un contexte scalaire, `getpwent` renvoie seulement le nom d'utilisateur. Le module `User::pwent` supporte une interface orientée *par nom* pour cette fonction. Voir *getpwent*(3).

getpwnam

```
getpwnam NOM
```

Cette fonction convertit un nom d'utilisateur vers l'entrée correspondante du fichier */etc/passwd*. La valeur renvoyée dans un contexte de liste :

```
($nom,$mot_de_passe,$uid,$gid,$quota,$commentaire,$nom_reel,$rep,$shell)=
    getpwnam("daemon");
```

Sur les systèmes qui supportent les mots de passe « shadow », vous devrez être le super-utilisateur pour récupérer le véritable mot de passe. Votre bibliothèque C doit remarquer que vous avez les permissions suffisantes et ouvre votre fichier */etc/shadow* (ou

n'importe quel endroit où le fichier *shadow* est conservé). Du moins, c'est censé fonctionner ainsi. Perl essayera de faire ceci si votre bibliothèque C est trop stupide pour le remarquer.

Pour faire des recherches répétitives, mieux vaut utiliser un hachage comme cache avec `getpwent`.

Dans un contexte scalaire, `getpwnam` renvoie seulement l'UID numérique. Le module `User::pwent` supporte une interface orientée *par nom* pour cette fonction. Voir *getpwnam*(3) et *passwd*(5).

getpwuid

T U

```
getpwuid UID
```

Cette fonction convertit un UID numérique vers l'entrée correspondante du fichier */etc/passwd*. La valeur renvoyée dans un contexte de liste :

```
($nom,$mot_de_passe,$uid,$gid,$quota,$commentaire,$nom_reel,$rep,$shell)=
    getpwuid(2);
```

Pour faire des recherches répétitives, mieux vaut utiliser un hachage comme cache avec `getpwent`.

Dans un contexte scalaire, `getpwuid` renvoie seulement le nom de l'utilisateur. Le module `User::pwent` supporte une interface orientée « par nom » pour cette fonction. Voir *getpwnam*(3) et *passwd*(5).

getservbyname

U

```
getservbyname NOM, PROTO
```

Cette fonction convertit un nom de service (port) vers son numéro de port correspondant. *PROTO* est un nom de protocole comme `"tcp"`. La valeur renvoyée dans un contexte de liste est :

```
($nom, $alias, $numero_port, $nom_protocole) = getservbyname("www", "tcp");
```

Dans un contexte scalaire, `getservbyname` renvoie seulement le numéro du port de service. Le module `Net::servent` supporte une interface orientée *par nom* pour cette fonction. Voir *getservbyname*(3).

getservbyport

U

```
getservbyport PORT, PROTO
```

Cette fonction convertit un numéro de service (port) vers les noms correspondants. *PROTO* est un nom de protocole comme `"tcp"`. La valeur renvoyée dans un contexte de liste est :

```
($nom, $alias, $numero_port, $nom_protocole) = getservbyport(80, "tcp");
```

Dans un contexte scalaire, `getservbyport` renvoie seulement le nom du service. Le module `Net::servent` supporte une interface orientée *par nom* pour cette fonction. Voir *getservbyport*(3).

getservent

```
getservent
setservent RESTE_OUVERT
endservent
```

Cette fonction parcourt votre fichier */etc/services* ou son équivalent. La valeur renvoyée dans un contexte de liste est :

```
($nom, $alias, $numero_port, $nom_protocole) = getservent();
```

Dans un contexte scalaire, `getservent` renvoie seulement le nom du port de service. Le module `Net::servent` supporte une interface orientée *par nom* pour cette fonction. Voir *getservent*(3).

getsockname

```
getsockname SOCKET
```

Cette fonction renvoie l'adresse socket empaquetée de cette extrémité de la connexion *SOCKET*. (Et pourquoi ne connaîtriez vous plus votre propre adresse ? Peut-être parce que vous avez attaché une adresse contenant des jokers (*wildcards*) à la socket serveur avant de faire un `accept` et que maintenant vous avez besoin de savoir à travers quelle interface quelqu'un est connecté avec vous. Ou la socket vous a été passée par votre processus père — *inetd*, par exemple.)

```
use Socket;
$mon_adr_sock = getsockname SOCK;
($port, $mon_adr) = sockaddr_in($mon_adr_sock);
$mon_nom = gethostbyaddr($mon_adr, AF_INET);
printf "Je suis %s [%vd]\n", $mon_nom, $mon_adr;
```

getsockopt

```
getsockopt SOCKET, NIVEAU, NOM_OPTION
```

Cette fonction renvoie l'option de socket requise ou *undef* en cas d'erreur. Voir `setsockopt` pour plus d'informations.

glob

```
glob EXPR
glob
```

Cette fonction renvoie la valeur de *EXPR* avec expansion des noms de fichiers, exactement comme un shell le ferait. Il s'agit de la fonction interne implémentant l'opérateur `<*>`.

Pour des raisons historiques, l'algorithme correspond au style d'expansion de *csh*(1) et pas à celui du Bourne Shell. Les versions de Perl antérieures à 5.6 utilisaient un processus externe, mais la 5.6 et les suivantes exécutent les globs en interne. Les fichiers dont le premier caractère est un point (« `.` ») sont ignorés sauf si ce caractère correspond explicitement. Un astérisque (« * ») correspond à n'importe quelle séquence de n'importe quels caractères (y compris aucun). Un point d'interrogation (« ? ») correspond à un

unique caractère quel qu'il soit. Une séquence entre crochets (« [. . .] ») spécifie une simple classe de caractères, comme « [chyO-9] ». Les classes de caractères peuvent être niées avec un accent circonflexe, comme dans « *.[^oa] », qui correspond à tous les fichiers ne commençant pas par un point et dont les noms contiennent un point suivi par un seul caractère qui ne soit ni un « a », ni un « o » à la fin du nom. Un tilde (« ~ ») se développe en un répertoire maison, comme dans « ~/.*rc » pour les fichiers « rc » de l'utilisateur courant, ou « ~caro/Mail/* » pour tous les fichiers mails de Caro. On peut employer des accolades pour les alternatives, comme dans « ~/.{mail,ex,csh,twm,}rc » pour obtenir ces fichiers rc particuliers.

Si vous voulez globber sur des noms de fichiers qui peuvent contenir des espaces, vous devrez utilisez le module File::Glob directement, puisque les grands-pères de glob utilisaient les espaces pour séparer plusieurs motifs comme <*.c *.h>. Pour plus de détail, voir File::Glob au chapitre 32. L'appel de glob (ou de l'opérateur <*>) fait automatiquement un use sur ce module, donc si se module s'évapore mystérieusement de votre bibliothèque, une exception est levée.

Lorsque vous appelez open, Perl ne développe pas les jokers, y compris les tildes. Vous devez d'abord globber les résultats.

```
open(MAILRC, "~/.mailrc")           # FAUX : le tilde est un truc du shell
    or die "Impossible d'ouvrir ~/.mailrc : $!";
open(MAILRC, (glob("~/.mailrc"))[0]) # développe d'abord le tilde
    or die "Impossible d'ouvrir ~/.mailrc : $!";
```

La fonction glob n'a aucun rapport avec la notion Perl de typeglobs, sinon qu'elles utilisent toutes deux un * pour représenter plusieurs éléments.

Voir aussi la section *Opérateur de globalisation des noms de fichier* au chapitre 2.

gmtime

```
gmtime EXPR
gmtime
```

Cette fonction convertit une heure renvoyée par la fonction time en un tableau de neuf éléments dont l'heure est en phase avec le méridien de Greenwich (alias GMT ou UTC, ou même Zoulou dans certaines cultures, dont ne fait bizarrement pas partie la culture Zoulou). Elle est généralement employée comme suit :

```
#  0    1     2      3     4      5      6      7       8
($sec,$min,$heure,$mjour,$mois,$annee,$sjour,$ajour,$est_dst) = gmtime();
```

Si, comme dans ce cas, *EXPR* est omise, la fonction effectue gmtime(time()). Le module Time::Local de la bibliothèque Perl contient un sous-programme, timegm, qui peut à l'inverse convertir la liste vers une valeur de time.

Tous les éléments sont numériques et proviennent directement d'une struct tm (une structure de programmation C ; pas d'affolement). Cela signifie en particulier que $mois se situe dans l'intervalle 0..11 avec janvier comme le mois 0 et que $sjour est dans 0..6 avec dimanche comme jour 0. Vous pouvez vous rappeler de ceux qui commencent par zéro car ce sont ceux que vous utilisez toujours comme indices dans les tableaux débutant par zéro contenant les noms des mois et des jours.

Par exemple, pour afficher le mois courant à Londres, vous pouvez écrire :

```
$mois_londres = (qw((Jan Fev Mar Avr Mai Jun
                     Jui Aou Sep Oct Nov Dec))[(gmtime)[4]];
```

`$annee` est le nombre d'années depuis 1900 ; c'est-à-dire qu'en 2023, `$annee` vaudra `123`, et pas simplement `23`. Pour obtenir l'année sur 4 chiffres, il suffit d'écrire `$annee+1900`. Pour l'avoir sur 2 chiffres (par exemple « 01 » en 2001), utilisez `sprintf("%02d", $annee % 100)`.

Dans un contexte scalaire, `gmtime` renvoie une chaîne ressemblant à *ctime*(3) basée sur la valeur de l'heure GMT. Le module `Time::gmtime` supporte une interface orientée « par nom » pour cette fonction. Voir aussi `POSIX::strftime` pour une approche permettant de formater des dates avec une granularité plus fine.

Cette valeur scalaire *n'est pas* dépendante des « locales » mais plutôt interne à Perl. Voir aussi le module `Time::Local` et les fonctions *strftime*(3) et *mktime*(3) disponibles *via* le module `POSIX`. Pour obtenir des chaînes pour la date qui soient similaires mais locales, positionnez vos variables d'environnement locales de manière appropriée (merci de voir la page de man pour *perllocale*) et essayez :

```
use POSIX qw(strftime);
$chaine_maintenant = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

Les caractères d'échappement `%a` et `%b`, qui représentent la forme abrégée pour le jour de la semaine et le mois de l'année, peuvent ne pas être nécessairement de trois caractères dans toutes les « locales ».

goto


```
goto ETIQUETTE
goto EXPR
goto &NOM
```

`goto` *ETIQUETTE* trouve l'instruction étiquetée par *ETIQUETTE* et continue l'exécution à partir de là. Si l'*ETIQUETTE* est impossible à trouver, une exception est levée. Elle ne peut pas être utilisée pour aller dans une construction demandant une initialisation, comme un sous-programme ou une boucle `foreach`. De même, elle ne peut être employée pour rentrer dans une construction optimisée. Elle peut être utilisée pour aller quasiment partout ailleurs dans l'espace de portée d'adressage dynamique,[4] y compris pour sortir des sous-programmes, mais il vaut alors mieux employer une autre instruction comme `last` ou `die`. L'auteur de Perl n'a jamais ressenti le besoin d'utiliser cette forme de `goto` (en Perl — en C, c'est une autre histoire).

Pour aller plus loin dans l'orthogonalité (et plus profond dans la stupidité), Perl autorise `goto` *EXPR* qui s'attend à ce que *EXPR* donne un nom d'étiquette, dont l'emplacement *ne peut jamais* être résolu avant l'exécution puisque l'étiquette est inconnue lorsqu'on compile l'instruction. Cela permet des `goto`s calculés comme en FORTRAN mais ce n'est pas vraiment recommandé[5] lorsqu'on cherche à optimiser la maintenabilité :

4. Cela veut dire que si elle ne trouve pas l'étiquette dans la routine courante, elle cherche dans celles qui l'ont appelée, rendant ainsi la maintenance du programme quasiment impossible.

5. Les euphémismes sont réputés être amusants, alors nous avons pensé en essayer un ici.

```
goto +("TRUC", "MACHIN", "CHOUETTE")[$i];
```

Sans aucun rapport, le `goto &NOM` fait de la grande magie en substituant un appel au sous-programme nommé en lieu et place du sous-programme actuellement en cours. Cette construction est utilisée sans scrupule par les sous-programmes `AUTOLOAD` qui veulent en charger un autre pour ensuite prétendre que celui-ci — et non l'original — a été appelé en premier lieu (sauf que toute modification de `@_` dans le sous-programme d'origine est propagée dans le sous-programme de remplacement). Après le `goto`, même `caller` ne pourra pas dire que la routine `AUTOLOAD` originelle avait été appelée en premier lieu.

grep

```
grep EXPR, LISTE
grep BLOC LISTE
```

Cette fonction évalue *EXPR* ou *BLOC* dans un contexte booléen pour chaque élément de *LISTE*, en positionnant temporairement `$_` avec chaque élément tout à tour, un peu comme la construction `foreach`. Dans un contexte de liste, elle renvoie la liste des éléments pour lesquels l'expression est vraie. (L'opérateur a été nommé d'après le bien-aimé programme UNIX qui extrait les lignes d'un fichier qui correspondent à un motif particulier. En Perl, l'expression est souvent un motif, mais ce n'est pas obligatoire.) Dans un contexte scalaire, `grep` renvoie le nombre d'occurrences où l'expression est vraie.

À supposer que `@toutes_les_lignes` contienne des lignes de code, cet exemple élimine les lignes de commentaires :

```
@lignes_de_code = grep !/^\s*#/, @toutes_les_lignes;
```

`$_` étant un alias implicite à chaque valeur de la liste, l'altération de `$_` modifiera les éléments de la liste originelle. Bien que ceci soit utile et supporté, les résultats peuvent devenir bizarres si l'on ne s'y attend pas. Par exemple,

```
@liste = qw(barney fred dino wilma);
@liste_grep = grep { s/^[bfd]// } @liste;
```

`@liste_grep` contient maintenant « `arney` », « `red` », « `ino` », mais `@liste` est devenu « `arney` », « `red` », « `ino` », « `wilma` » ! Danger de bourde de programmeur.

Voir également `map`. Les deux instructions suivantes sont fonctionnellement équivalentes :

```
@sortie = grep { EXPR } @entree;
@sortie = map { EXPR ? $_ : () } @entree
```

hex $_

```
hex EXPR
hex
```

Cette fonction interprète *EXPR* comme une chaîne hexadécimale et renvoie la valeur décimale équivalente. S'il y un « `0x` » au début, il est ignoré. Pour interpréter des chaînes commençant soit par `0`, soit par `0b`, ou par `0x`, voir `oct`. Le code suivant met `$nombre` à

4 294 906 560 :

```
$nombre = hex("ffff12c0");
```

Pour la fonction inverse, employez `sprintf` :

```
sprintf "%lx", $nombre;    # (la lettre L en minuscule, pas le chiffre un.)
```

Les chaînes hexadécimales ne peuvent représenter que des entiers. Les chaînes qui causeraient un débordement de la limite pour les entiers déclenchent un avertissement.

import

```
import NOM_CLASSE LISTE
import NOM_CLASSE
```

Il n'existe pas de fonction `import` intégrée. Il s'agit essentiellement d'une méthode de classe ordinaire définie (ou héritée) par les modules qui désirent exporter des noms vers un autre module par l'intermédiaire de l'opérateur `use`. Voir `use` pour plus de détails.

index

```
index CHAINE, SOUS_CHAINE, DECALAGE
index CHAINE, SOUS_CHAINE
```

Cette fonction recherche une chaîne à l'intérieur d'une autre. Elle renvoie la position de la première occurrence de *SOUS_CHAINE* dans *CHAINE*. Le *DECALAGE*, s'il est spécifié, indique à partir de combien de caractère débute la recherche. Les positions partent de 0 (ou de là où vous avez positionné la variable de base des indices `$[`, mais ne faites pas ça). Si la sous-chaîne n'est pas trouvée, la fonction renvoie un de moins que la base, généralement `-1`. Pour balayer une chaîne, on peut écrire :

```
$pos = -1;
while (($pos = index($chaine, $chaine_recherchee, $pos)) > -1) {
    print "Trouvé en $pos\n";
    $pos++;
}
```

int


```
int EXPR
int
```

Cette fonction renvoie la partie entière de *EXPR*. Si vous êtes un programmeur C, vous êtes susceptible d'oublier fréquemment d'utiliser `int` en conjonction avec la division, qui est une opération en virgule flottante en Perl :

```
$age_moyen = 939/16;      # donne 58.6875 (58 en C)
$age_moyen = int 939/16;  # donne 58
```

Vous ne devez pas employer cette fonction pour arrondir dans tous les cas, car elle tronque vers 0 et parce que la représentation pour la machine des nombres en virgule flottante peut quelque fois produire des résultats qui ne sont pas intuitifs. Par exemple, `int(-6.725/0.025)` donnera `-268` au lieu de la valeur correcte `-269` ; ceci car la valeur est plus proche de `-268.99999999999994315658`. Généralement, les fonctions `sprintf`,

`printf` ou `POSIX::floor` ou encore `POSIX::ceil` vous serviront plus que ne le ferait `int`.

```
$n = sprintf("%.0f", $f);   # arrondit (et ne tronque pas)
                            # vers l'entier le plus proche
```

ioctl

$! X ARG X RO X T X U

```
ioctl HANDLE_FICHIER, FONCTION, SCALAIRE
```

Cette fonction implémente l'appel système *ioctl*(2) qui contrôle les entrées/sorties. Il vous faudra peut-être d'abord écrire pour obtenir les définitions de fonctions correctes :

```
require "sys/ioctl.ph";     # probablement /usr/local/lib/perl/sys/ioctl.ph
```

Si *sys/ioctl.ph* n'existe pas ou ne comporte pas les définitions correctes, vous devrez écrire les vôtres, en vous basant sur vos en-têtes de fichiers C comme *sys/ioctl.h*. (La distribution Perl contient un script appelé *h2ph* pour vous y aider, mais cela reste complexe.) *SCALAIRE* sera lu ou écrit (ou les deux) selon la *FONCTION* — un pointeur sur la valeur chaîne de *SCALAIRE* sera passé comme troisième argument de l'appel réel à *ioctl*(2). (Si *SCALAIRE* n'a pas de valeur chaîne, mais une valeur numérique, c'est cette dernière qui sera passée directement au lieu d'un pointeur sur la valeur chaîne.) Les fonctions `pack` et `unpack` servent notamment à manipuler les valeurs des structures utilisées par `ioctl`. L'exemple suivant détermine combien d'octets sont disponibles pour être lus en utilisant un `FIONREAD` :

```
require 'sys/ioctl.ph';

$taille = pack("L", 0);
ioctl(HF, FIONREAD(), $taille)
    or die "Impossible d'appeler ioctl : $!\n";
$taille = unpack("L", $taille);
```

Si *h2ph* n'a pas été installé ou ne fonctionne pas dans votre cas, vous pouvez faire un `grep` à la main sur les fichiers d'en-têtes ou écrire un petit programme C pour afficher la valeur.

La valeur de retour d'`ioctl` (et de `fcntl`) est comme suit :

L'appel système renvoie	Perl renvoie
`-1`	`undef`.
`0`	La chaîne « `0 but true` ».
autre chose	Le même nombre.

Perl renvoie donc vrai pour un succès et faux pour un échec, tout en vous laissant facilement déterminer la valeur renvoyée par le système d'exploitation :

```
$val_ret = ioctl(...) || -1;
printf "ioctl a en fait renvoyé %d\n", $val_ret;
```

La chaîne spéciale « `0 but true` » (*N.d.T.* : 0 mais vrai) est exempte des avertissements de **`-w`** à propos des conversions numériques inadaptées.

Les appels à `ioctl` ne doivent pas être considérés comme portables. Si, par exemple,

vous voulez simplement désactiver l'écho une seule fois pour le script tout entier, il est beaucoup plus portable d'écrire :

```
system "stty -echo"; # Fonctionne sur la plupart des Unix
```

Ce n'est pas parce qu'il est *possible* de faire quelque chose en Perl qu'il *faille* le faire. Pour citer l'apôtre Saint Paul : « Tout est acceptable — mais tout n'est pas bénéfique. »

Pour une meilleur portabilité, vous devriez regarder le module de CPAN `Term::ReadKey`.

join

```
join EXPR, LISTE
```

Cette fonction joint les chaînes séparées de *LISTE* en une chaîne unique dont les champs sont séparés par la valeur de *EXPR*, et renvoie la chaîne résultante. Par exemple :

```
$enreg = join ':', $login,$mot_de_passe,$uid,$gid,$nom_reel,$rep,$shell;
```

Pour faire l'inverse, voir `split`. Pour joindre des éléments dans des champs à position fixe, voir `pack`. La manière la plus efficace de concaténer des chaînes consiste à les `join`dre avec la chaîne vide :

```
$chaine = join "", @tableau;
```

Au contraire de `split`, `join` ne prend pas un motif comme premier argument, et elle produit un avertissement si vous tentez de le faire.

keys

```
keys HACHAGE
```

Cette fonction renvoie une liste constituée de toutes les clefs du *HACHAGE* indiqué. Les clefs sont renvoyées dans un ordre apparemment aléatoire, mais c'est le même que celui produit par les fonctions `values` ou `each` (à supposer que le hachage n'a pas été modifié entre les appels). Comme effet de bord, elle réinitialise l'itérateur de *HACHAGE*. Voici une manière (un peu tordue) d'afficher votre environnement :

```
@clefs   = keys   %ENV;    # les clefs sont dans le même ordre que
@valeurs = values %ENV;    # les valeurs, comme ceci le démontre
while (@clefs) {
    print pop(@clefs), '=', pop(@valeurs), "\n";
}
```

Vous préférez peut-être plus voir votre environnement affiché dans l'ordre des clefs :

```
foreach $clef (sort keys %ENV) {
    print $clef, '=', $ENV{$clef}, "\n";
}
```

Vous pouvez trier les valeurs d'un hachage directement, mais c'est plutôt inutile en l'absence d'une solution pour retrouver les clefs à partir des valeurs. Pour trier un hachage par ses valeurs, vous avez généralement besoin de trier le résultat de `keys` en fournissant une fonction de comparaison qui accède aux valeurs en fonction des clefs. Voici un tri numérique décroissant d'un hachage par ses valeurs :

```
foreach $clef (sort { $hachage{$b} <=> $hachage{$a} } keys %hachage) {
    printf "%4d %s\n", $hachage{$clef}, $clef;
}
```

L'utilisation de keys sur un hachage lié à un gros fichier DBM produirait une liste volumineuse, et donc un processus volumineux. Il vaut mieux que vous utilisiez dans ce cas la fonction each, qui parcourt les entrées du hachage une à une sans les absorber toutes dans une seule liste gargantuesque.

Dans un contexte scalaire, keys renvoie le nombre d'éléments du hachage (et réinitialise l'itérateur de each). Cependant, pour obtenir cette information à partir d'un hachage lié, y compris à un fichier DBM, Perl doit le parcourir en entier, ce qui n'est pas très efficace dans ce cas. L'appel de keys dans un contexte vide facilite ceci.

Utilisée en tant que lvalue, keys augmente le nombre de cellules de hachage (*N.d.T.* : *hash buckets*) allouées pour le hachage donné. (C'est la même chose que d'étendre *a priori* un tableau en assignant un nombre élevé pour $#tableau.) Étendre *a priori* un hachage peut se traduire par un gain de performance, s'il vous arrive de savoir si le hachage va être gros, et quelle grosseur il atteindra. Si vous écrivez :

```
keys %hachage = 1000;
```

le %hachage aura alors au moins 1000 cellules allouées pour lui (vous aurez en fait 1024 cellules, puisqu'on arrondit à la prochaine puissance de deux). Vous pouvez diminuer le nombre de cellules allouées pour un hachage en utilisant keys de cette manière (mais vous n'aurez pas à vous inquiéter de faire cela par accident, puisqu'essayer est sans effet). Les cellules seront conservées même si vous faites %hachage = (). Utilisez undef %hachage si vous voulez libérer de la place en mémoire alors que %hachage est toujours dans la portée courante.

Voir également each, value et sort.

kill

$! X ARG X U X T

```
kill SIGNAL, LISTE
```

Cette fonction envoie un signal à une liste de processus. Pour le *SIGNAL*, vous pouvez utiliser soit un entier, soit le nom d'un signal entre apostrophes (sans le « SIG » au début). Une tentative d'employer un *SIGNAL* non-reconnu lève une exception. La fonction renvoie le nombre de processus qui ont réussi à recevoir le signal. Si *SIGNAL* est négatif, la fonction touchera des groupes de processus au lieu des processus. (Sur System V, un numéro de processus négatif touchera également des groupes de processus, mais ce n'est pas portable.) Un PID égal à zéro enverra le signal à tous les processus du même groupe que celui de l'émetteur. Par exemple :

```
$compt = kill 1, $fils1, $fils2;
kill 9, @mourants;
kill 'STOP', getppid          # Peut suspendre *ainsi* le shell de login
        unless getppid == -1; # (Mais ne titillez pas init(8).)
```

Un *SIGNAL* valant 0 teste si un processus est toujours en vie et que vous avez toujours les permissions pour lui envoyer un signal. Aucun signal n'est envoyé. Vous pouvez de cette manière détecter si un processus est toujours en vie et s'il n'a pas changé d'UID.

```
use Errno qw(ESRCH EPERM);
```

```
if (kill 0 => $larbin) {
    print "$larbin est en vie !\n";
} elsif ($! == EPERM) {                 # UID changé
    print "$larbin a échappé à mon contrôle !\n";
} elsif ($! == ESRCH) {                 # ou zombie
    print "$larbin est décédé.\n";
} else {
    warn "Bizarre : je n'ai pas pu vérifier l'état de $larbin : $!\n";
}
```

Voir la section *Signaux* au chapitre 16.

last

$@

```
last ETIQUETTE
last
```

L'opérateur `last` sort immédiatement de la boucle en question, exactement comme l'instruction `break` en C ou en Java (lorsqu'elle est utilisée dans les boucles). Si l'*ETIQUETTE* est omise, l'opérateur se réfère à la boucle la plus intérieure qui l'encadre. Le bloc `continue`, s'il existe, n'est pas exécuté.

```
LIGNE: while (<MSG_MAIL>) {
    last LIGNE if /^$/; # sort lorsqu'on en a fini avec les en-têtes
    # suite de la boucle ici
}
```

`last` ne peut pas être employé pour sortir d'un bloc qui renvoie une valeur, comme `eval {}`, `sub {}` ou `do {}`, et ne doit pas être utilisé pour sortir d'une opération `grep` ou `map`. Si les avertissements sont actifs, Perl vous préviendra si vous faites un `last` pour sortir d'une boucle qui est hors de votre portée lexicale, comme une boucle dans une sous-routine appelante.

Un bloc en soi-même est sémantiquement identique à une boucle qui ne s'exécute qu'une seule fois. Ainsi `last` peut être employé pour effectuer une sortie anticipée d'un tel bloc.

Voir aussi le chapitre 4 pour des illustrations du fonctionnement de `last`, `next`, `redo` et `continue`.

lc

$_ T

```
lc EXPR
lc
```

Cette fonction renvoie une version en minuscules de *EXPR*. Il s'agit de la fonction interne implémentant l'échappement `\L` dans les chaînes entre guillemets. La valeur de votre locale `LC_CTYPE` est respectée si `use locale` est actif, bien que la manière dont les locales interagissent avec Unicode est toujours un sujet à l'étude, comme ils disent. Voir la page de man *perllocale* pour les résultats les plus à jour.

lcfirst

$_ T

```
lcfirst EXPR
lcfirst
```

Cette fonction renvoie une version de *EXPR* avec le premier caractère en minuscule. Il s'agit de la fonction interne implémentant l'échappement `\l` dans les chaînes entre guillemets. La valeur de votre locale `LC_CTYPE` est respectée si vous employez `use locale` et nous tâchons de comprendre le rapport avec Unicode.

length

$_

```
length EXPR
length
```

Cette fonction renvoie la longueur en caractères de la valeur scalaire de *EXPR*. Si *EXPR* est omise, la fonction renvoie la longueur de `$_` (mais prenez garde que ce qui suit ne ressemble pas au début d'une *EXPR* ression, faute de quoi le moteur lexical de Perl sera perturbé. Par exemple, `length < 10` ne compilera pas. En cas de doute, mettez des parenthèses.)

N'essayez pas d'utiliser `length` pour trouver la taille d'un tableau ou d'un liste. `scalar @tableau` donne la taille d'un tableau et `scalar keys %hachage` celle des couples clef/valeur d'un hachage. (Le `scalar` est habituellement omis lorsqu'il est redondant.)

Pour trouver la taille d'une chaîne en octets plutôt qu'en caractères, écrivez :

```
$oct_long = do { use bytes; length $chaine; };
```

ou

```
$oct_long = bytes::length $chaine; # doit d'abord faire un use bytes
```

link

$! X U X T

```
link ANCIEN_FICHIER, NOUVEAU_FICHIER
```

Cette fonction crée un nouveau nom de fichier lié à l'ancien. La fonction renvoie vrai en cas de succès, faux sinon. Voir également `symlink` plus loin dans ce chapitre. Il est peu probable que cette fonction soit implémentée sur les systèmes de fichiers non-Unix.

listen

$! X ARG X U

```
listen SOCKET, TAILLE_FILE
```

Cette fonction informe le système que vous allez accepter des connexions sur cette *SOCKET* et que le système peut mettre en file d'attente le nombre de connexions spécifiées par *TAILLE_FILE*. Imaginez un signal d'appel sur votre téléphone, avec la possibilité de mettre 17 appels en file d'attente. (Encore faut-il le vouloir !) La fonction renvoie vrai si elle a réussi, sinon faux.

```
use Socket;
listen(SOCK_PROTO, SOMAXCONN)
    or die "Impossible de mettre une file d'attente avec
            listen sur SOCK_PROTO : $!";
```

Voir `accept`. Voir également la section *Sockets* au chapitre 16. Voir *listen*(2).

local

```
local EXPR
```

Cet opérateur ne crée pas de variable locale ; utilisez `my` pour le faire. À la place, il rend locales des variables existantes ; c'est-à-dire qu'il donne à une ou plusieurs variables globales une portée locale à l'intérieur du bloc, de l'`eval` ou du fichier le plus intérieur les encadrant. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses car la priorité de l'opérateur est plus élevée que celle des virgules. Tous les éléments listés doivent être des lvalues légales, c'est-à-dire quelque chose auquel vous pouvez assigner une valeur ; ceci peut comprendre les éléments individuels de tableaux ou de hachages.

Cet opérateur fonctionne en sauvegardant les valeurs courantes des variables spécifiées dans une pile cachée et en les restaurant au moment de sortir du bloc, du sous-programme, de l'`eval` ou du fichier. Après que le `local` a été exécuté, mais avant de sortir de sa portée, tout sous-programme ou tout format exécuté verra les valeurs locales internes et non les valeurs externes précédentes car la variable reste globale, bien que sa valeur soit localisée. Le terme technique est « portée dynamique ». Voir la section *Déclarations avec portée* au chapitre 4.

On peut assigner une valeur à *EXPR* si besoin, ce qui permet d'initialiser vos variables au moment où vous les rendez locales. Si aucune initialisation explicite n'est effectuée, tous les scalaires sont initialisés à `undef` et tous les tableaux et hachages à `()`. Comme pour les affectations ordinaires, si l'on utilise des parenthèses autour des variables à gauche (ou si la variable est un tableau ou un hachage), l'expression de droite est évaluée dans un contexte de liste. Sinon, l'expression de droite est évaluée dans un contexte scalaire.

Dans tous les cas, l'expression de droite est évaluée avant de rendre les variables locales, mais l'initialisation est effectuée après, vous permettant ainsi d'initialiser une variable rendue locale avec la valeur qu'elle contenait avant de l'être. Par exemple, ce code démontre comment modifier temporairement un tableau global :

```
if ($op eq '-v') {
    # initialiser le tableau local avec le tableau global
    local @ARGV = @ARGV;
    unshift @ARGV, 'echo';
    system @ARGV;
}
# @ARGV restauré
```

Vous pouvez aussi modifier temporairement des hachages globaux :

```
# ajoute temporairement quelques entrées au hachage %chiffres
if ($base12) \{
    # (NB : ce n'est pas forcément efficace !)
    local(%chiffres) = (%chiffres, T => 10, E => 11);
    &parcours_nombres();
}
```

Vous pouvez utiliser `local` pour donner des valeurs temporaires aux éléments individuels de tableaux ou de hachages, même s'ils ont une portée lexicale :

```
if ($protege) {
    local $SIG{INT} = 'IGNORE';
    precieux();     # pas d'interruption durant cette fonction
}                   # le gestionnaire précédent (s'il y en a) est restauré
```

Vous pouvez utiliser `local` avec les typeglobs pour créer des handles de fichiers locaux sans charger des modules objets encombrants :

```
local *MOTD;                # protège tout handle MOTD global
my $hf = do { local *HF };  # crée un nouveau handle de fichier indirect
```

(Depuis la version 5.6 de Perl, un `my $hf;` ordinaire suffit car il vous donne une variable indéfinie là où un handle de fichier est attendu, comme le premier argument de `open` ou `socket`, Perl donne maintenant la vie à un tout nouveau handle de fichier pour vous.)

Mais en général, préférez `my` au lieu de `local`, car `local` n'est pas vraiment ce que la plupart des gens définit comme « local », ni même « lo-cal ». Voir `my`.

localtime

```
localtime EXPR
localtime
```

Cette fonction convertit une heure renvoyée par la fonction `time` en un tableau de neuf éléments dont l'heure est en phase avec le méridien local. Elle est généralement employée comme suit :

```
#  0    1     2      3     4      5      6      7      8
($sec,$min,$heure,$mjour,$mois,$annee,$sjour,$ajour,$est_dst) = localtime;
```

Si, comme dans ce cas, *EXPR* est omise, la fonction effectue `localtime(time())`.

Tous les éléments sont numériques et proviennent directement d'une `struct tm` (un jargon de programmation C — pas d'affolement). Cela signifie en particulier que `$mois` se situe dans l'intervalle `0..11` avec janvier comme le mois `0` et que `$sjour` est dans `0..6` avec dimanche comme jour `0`. Vous pouvez vous rappeler de ceux qui commencent par zéro car ce sont ceux que vous utilisez toujours comme indices dans les tableaux débutant par zéro contenant les noms des mois et des jours.

Par exemple, pour obtenir le jour actuel de la semaine :

```
$ce_jour = (qw(Dim Lun Mar Mer Jeu Ven Sam))[(localtime)[6]];
```

`$annee` est le nombre d'années depuis 1900 ; c'est-à-dire qu'en 2023, `$annee` vaudra `123`, et pas simplement `23`. Pour obtenir l'année sur 4 chiffres, il suffit d'écrire `$annee+1900`. Pour l'avoir sur 2 chiffres (par exemple « 01 » en 2001), utilisez `sprintf("%02d", $annee % 100)`.

Le module `Time::Local` de la bibliothèque Perl contient un sous-programme, `timelocal`, qui peut faire la conversion dans le sens l'inverse.

Dans un contexte scalaire, `localtime` renvoie une chaîne ressemblant à *ctime*(3). Par exemple, la commande *date*(1) peut (presque[6]) être émulée par :

6. *date*(1) affiche le fuseau horaire, alors que `scalar localtime` ne le fait pas.

```
perl -le 'print scalar localtime';
```

Voir aussi la fonction `strftime` du le module standard `POSIX` pour une approche permettant de formater des dates avec une granularité plus fine. Le module `Time::localtime` supporte une interface orientée « par nom » pour cette fonction.

lock

```
lock CHOSE
```

La fonction `lock` place un verrou sur une variable, un sous-programme ou un objet référencé par *`CHOSE`* jusqu'à ce que le verrou sorte de la portée courante. Pour assurer une compatibilité antérieure, cette fonction n'est interne que si Perl a été compilé en autorisant les threads et si vous avez écrit `use Threads`. Sinon, Perl supposera qu'il s'agit d'une fonction définie par l'utilisateur. Voir le chapitre 17, *Threads*.

log

$_ $@

```
log EXPR
log
```

Cette fonction renvoie le logarithme naturel (c'est-à-dire en base *e*) de *`EXPR`*. Si *`EXPR`* est négative, un exception est levée. Pour obtenir le log dans une autre base, utilisez l'algèbre de base : le log en base-*N* d'un nombre est égal au log de ce nombre divisé par le log naturel de *N*. Par exemple :

```
sub log10 {
    my $n = shift;
    return log($n)/log(10);
}
```

Pour l'inverse du `log`, voir `exp`.

lstat

$_ $! x u

```
lstat EXPR
lstat
```

Cette fonction est similaire à la fonction `stat` de Perl (y compris en positionnant le handle de fichier spécial _), mais si le dernier composant du nom de fichier est un lien symbolique, elle agit sur le lien symbolique lui-même au lieu du fichier sur lequel pointe le lien en question. (Si les liens symboliques ne sont pas implémentés, c'est un `stat` normal qui est lancé en lieu et place.)

m//

T x t

```
/MOTIF/
m/MOTIF/
```

Il s'agit de l'opérateur de correspondance, qui interprète *`MOTIF`* comme une expression régulière. L'opérateur est évalué comme une chaîne entre guillemets plutôt que comme une fonction. Voir le chapitre 5, *Recherche de motif*.

map

```
map BLOC LISTE
map EXPR, LISTE
```

Cette fonction évalue le *BLOC* ou l'*EXPR* pour chaque élément de *LISTE* (en affectant localement `$_` à chaque élément) et renvoie la liste composée des résultats de chacune de ces évaluations. Elle évalue *BLOC* ou *EXPR* dans un contexte de liste, ce qui fait que chaque élément de *LISTE* peut produire zéro, un ou plusieurs éléments dans la valeur renvoyée. Ces éléments sont tous rassemblés en une seule liste plate. Par exemple :

```
@mots = map { split ' ' } @lignes;
```

éclate une ligne en une liste de mots. Mais on observe souvent une correspondance bijective entre les valeurs de sortie et d'entrée :

```
@caracteres = map chr, @nombres;
```

convertit une liste de nombres vers leurs caractères correspondants. Voici un exemple de correspondance un-vers-deux :

```
%hachage = map { gen_clef($_) => $_ } @tableau;
```

qui n'est qu'une façon amusante et fonctionnelle d'écrire ceci :

```
%hachage = ();
foreach $_ (@tableau) {
    $hachage{gen_clef($_)} = $_;
}
```

Comme `$_` est un alias (une référence implicite) pour les valeurs de la liste, on peut utiliser cette variable pour modifier les éléments dans le tableau. C'est utile et supporté bien que cela puisse générer d'étranges résultats si la *LISTE* n'est pas un tableau nommé. L'utilisation d'une boucle `foreach` habituelle dans cet objectif peut paraître plus clair. Voir également `grep` ; `map` diffère de `grep` en ce que `map` renvoie une liste composée des résultats de chaque évaluation successive de *EXPR*, alors que `grep` renvoie une liste composée de chaque valeur de *LISTE* pour laquelle *EXPR* est évaluée à vrai.

mkdir


```
mkdir REPERTOIRE, MASQUE
mkdir REPERTOIRE
```

Cette fonction crée le répertoire spécifié par *REPERTOIRE* en lui donnant les permissions spécifiées par le *MASQUE* numérique, modifié par le `umask` courant. Si l'opération réussie, elle renvoie vrai, sinon faux.

Si *MASQUE* est omis, on suppose un masque de `0777`, qui de toute façon est ce que vous voulez dans la plupart des cas. En général, créer des répertoires avec un *MASQUE* permissif (comme `0777`) et laisser l'utilisateur modifier cela avec son `umask` est une meilleure habitude plutôt que de fournir un *MASQUE* restrictif sans donner à l'utilisateur un moyen d'être plus permissif. L'exception à cette règle est le cas où le fichier ou le répertoire doit rester privé (les fichiers mail, par exemple). Voir `umask`.

Si l'appel système *mkdir*(2) n'est pas intégré à votre bibliothèque C, Perl l'émule en appelant le programme *mkdir*(1). Pour créer une longue liste de répertoires sur ce genre

de système, il sera plus efficace d'appeler vous-mêmes le programme *mkdir* avec la liste des répertoires pour éviter de créer des palanquées de sous-processus.

msgctl

```
msgctl ID, CMD, ARG
```

Cette fonction invoque l'appel system *msgctl*(2) des IPC System V ; voir *msgctl*(2) pour plus de détails. Il se peut que vous deviez faire un `use IPC::SysV` pour obtenir les définitions de constante correctes. Si *CMD* vaut `IPC_STAT`, *ARG* doit alors être une variable qui contiendra la structure C `msqid_ds` renvoyée. Les valeurs de retour fonctionnent comme celles de `ioctl` et `fcntl` : `undef` pour une erreur, « `0 but true` » (*N.d.T.* : 0 mais vrai) pour zéro ou la valeur réelle dans les autres cas.

Cette fonction n'est disponible que sur les machines qui supportent les IPC System V, ce qui s'avère bien plus rare que celles qui supportent les sockets.

msgget

```
msgget CLEF, FLAGS
```

Cette fonction invoque l'appel système *msgget*(2) des IPC System V. Voir *msgget*(2) pour plus de détails. La fonction renvoie l'*ID* de file de messages ou `undef` s'il y a une erreur. Avant d'appeler, vous devez écrire `use IPC::SysV`.

Cette fonction n'est disponible que sur les machines qui supportent les IPC System V.

msgrcv

```
msgrcv ID, VAR, TAILLE, TYPE, FLAGS
```

Cette fonction invoque l'appel système *msgrcv*(2) des IPC System V pour recevoir un message depuis la file de messages *ID* dans la variable *VAR* avec une taille maximum *TAILLE*. Voir *msgrcv*(2) pour plus de détails. Lorsqu'un message est reçu, le type de message sera la première chose dans *VAR* et la longueur maximale de *VAR* sera *TAILLE* plus la taille du type de message. La fonction renvoie vrai si elle réussit ou undef s'il y a une erreur. Avant d'appeler, vous devez écrire `use IPC::SysV`.

Cette fonction n'est disponible que sur les machines qui supportent les IPC System V.

msgsnd

```
msgsnd ID, MSG, FLAGS
```

Cette fonction invoque l'appel système *msgsnd*(2) des IPC System V pour envoyer le message *MSG* dans la file de messages *ID*. Voir *msgsnd*(2) pour plus de détails. *MSG* doit commencer par l'entier long représentant le type de message. Vous pouvez créer un message comme ceci :

```
$msg = pack "L a*", $type, $texte_du_message;
```

La fonction renvoie vrai si elle réussit ou undef s'il y a une erreur. Avant d'appeler, vous devez écrire `use IPC::SysV`.

Cette fonction n'est disponible que sur les machines qui supportent les IPC System V.

my

```
my TYPE EXPR : ATTRIBUTS
my EXPR : ATTRIBUTS
my TYPE EXPR
my EXPR
```

Cet opérateur déclare une ou plusieurs variables privées comme n'existant que dans le bloc, le sous-programme, l'`eval` ou le fichier le plus intérieur les encadrant. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses, car l'opérateur lie plus fortement que la virgule. Seuls les scalaires simples ou les tableaux et les hachages complets peuvent être déclarés de cette façon.

Le nom de la variable ne peut pas être qualifié par un paquetage, car les variables de paquetage sont toutes globalement accessibles par le biais de leur table de symboles correspondante tandis que les variables lexicales sont sans rapport avec une quelconque table de symboles. À l'inverse de `local`, cet opérateur n'a rien à voir avec les variables globales, sinon masquer une autre variable du même nom dans sa portée (c'est-à-dire, là où la variable privée existe). Cependant, une variable globale peut toujours être atteinte sous sa forme qualifiée par paquetage ou par le biais d'une référence symbolique.

La portée d'une variable privée ne commence pas avant l'instruction qui *suit* sa déclaration. Elle s'étend à n'importe quel bloc contenu ensuite, jusqu'à la fin de la portée de la variable elle-même.

Cela signifie cependant que tout sous-programme que vous appelez dans la portée d'une variable privée ne peut pas la voir, à moins que le bloc qui définit le sous-programme lui-même ne fasse textuellement partie de la portée de cette variable. Le terme technique consacré est *portée lexicale*, nous les appelons donc souvent *variables lexicales*. Dans la culture C, elles sont appelées variables auto, car elles sont automatiquement allouées et désallouées respectivement à l'entrée et à la sortie de la portée.

Vous pouvez si besoin assigner une valeur à *EXPR*, ce qui vous permet d'initialiser les variables locales. (Si aucune initialisation explicite n'est effectuée, tous les scalaires sont initialisés à la valeur indéfinie et tous les tableaux et hachages à la liste vide). Comme pour les affectations ordinaires, si l'on utilise des parenthèses autour des variables à gauche (ou si la variable est un tableau ou un hachage), l'expression de droite est évaluée dans un contexte de liste. Sinon, l'expression de droite est évaluée dans un contexte scalaire. Par exemple, les paramètres formels de sous-programme peuvent être nommés par une affectation de liste, comme ceci :

```
my ($amis, $romains, $paysans) = @_;
```

Mais prenez garde à ne pas oublier les parenthèses indiquant une affectation de liste, comme ceci :

```
my $pays = @_;   # Est-ce correct ou pas ?
```

Ici, c'est la longueur du tableau (c'est-à-dire le nombre d'arguments du sous-programme) qui est assignée à la variable, car le tableau est évalué dans un contexte scalaire. Vous pouvez cependant employer l'affectation scalaire pour un paramètre formel, tant que vous utilisez l'opérateur `shift`. En fait, comme les méthodes sont passées aux objets dans le premier argument, de nombreux sous-programmes méthodes commencent comme ceci :

```
sub simple_comme {
    my $obj = shift;      # affectation scalaire
    my ($a,$b,$c) = @_;   # affectation de liste
    ...
}
```

Si vous tentez de déclarer un sous-programme de portée lexicale avec `my sub`, Perl mourra avec le message que cette fonctionnalité n'a pas encore été implémentée. (Sauf si bien sûr, cette fonctionnalité *a* déjà été implémenté.)

Les *TYPE* et *ATTRIBUTS* sont optionnels, ce qui se justifie puisqu'ils doivent tous deux être considérés comme expérimentaux. Voici à quoi ressemble une déclaration qui les utilise :

```
my Chien $medor :oreilles(courtes) :queue(longue);
```

Le *TYPE*, s'il est spécifié, indique quelle sorte de scalaire ou de scalaires sont déclarés dans *EXPR*, soit directement comme une ou plusieurs variables scalaires, soit indirectement par le biais d'un tableau ou d'un hachage. Si *TYPE* est le nom d'une classe, on supposera que les scalaires contiennent des références à des objets de ce type, ou à des objets compatibles avec ce type. En particulier, les classes dérivées sont considérées comme compatibles. C'est-à-dire, en supposant que `Colley` soit dérivée de `Chien`, vous pouvez déclarer :

```
my Chien $lassie = new Colley;
```

Votre déclaration affiche que vous désirez utiliser l'objet `$lassie` en accord avec son appartenance à un objet `Chien`. Le fait que ce soit vraiment un objet `Colley` ne doit pas avoir d'influence tant que vous vous lui faites faire des choses de `Chien`. Malgré la magie des méthodes virtuelles, l'implémentation de ces méthodes de `Chien` peuvent très bien être dans la classe `Colley`, mais la déclaration ci-dessus ne parle que de l'interface et pas de l'implémentation. En théorie.

De manière intéressante, dans la version 5.6.0, la seule fois où Perl accorde une attention à la déclaration de *TYPE* est lorsque la classe correspondante a déclaré des champs avec le pragma `use fields`. Prises ensembles, ces déclarations permettent à l'implémentation d'une classe en tant que pseudo-hachage de « s'exhiber » au code qui se trouve à l'extérieur de la classe, permettant ainsi aux recherches dans le hachage d'être optimisées par le compilateur comme étant des recherches dans un tableau. En un sens, le pseudo-hachage *est* l'interface pour une telle classe, notre théorie reste donc intacte, si ce n'est un peu cabossée. Pour plus d'informations sur les pseudo-hachages, voir la section *Pseudo-hachages* au chapitre 8, *Références*.

À l'avenir, d'autres types de classes pourront interpréter le *TYPE* différemment. La déclaration de *TYPE* doit être perçue comme une interface de type générique qui pourra un jour être instanciée de différentes manières selon la classe. En fait, le *TYPE* pourra même ne pas être un nom de classe officiel. Nous réservons les noms de types en minuscules pour Perl, car l'une des manières que nous préférons pour étendre le type d'interface est d'autoriser les déclarations de type de bas niveau comme `int`, `num`, `str` et `ref`. Ces déclarations ne seront pas destinées à un typage fort ; elles seront plutôt des indications pour le compilateur, en lui disant d'optimiser le stockage de la variable en supposant que cette variable sera le plus souvent utilisée comme elle a été déclarée. La sémantique des scalaires restera à peu près la même — vous serez toujours capables d'additionner

deux scalaires `str` (*N.d.T.* : chaînes) ou d'afficher un scalaire `int` (*N.d.T.* : entier), exactement comme s'ils étaient les scalaires polymorphes auxquels vous êtes familiers. Mais avec une déclaration `int`, Perl pourra décider de ne stocker que la valeur entière et d'oublier de sauver dans le cache la chaîne résultante comme il le fait actuellement. Les boucles avec des variables de boucle `int` pourront tourner plus rapidement, spécialement dans le code compilé en C. Les tableaux de nombres pourront en particulier être stockés de manière plus compacte. Comme cas limite, la fonction interne `vec` deviendra même obsolète alors que nous pourrons écrire des déclarations comme :

```
my bit @chaine_de_bits;
```

La déclaration *`ATTRIBUTS`* est encore plus expérimentale. Nous n'avons pas fait grand chose de plus que réserver la syntaxe et prototyper l'interface interne ; voir le pragma `use attributes` au chapitre 31 pour plus d'informations à ce sujet. Le premier attribut que nous implémenterons est susceptible d'être `constant` :

```
my num $PI : constant = atan2(1,1) * 4;
```

Mais il y a beaucoup d'autres possibilités, comme établir des valeurs par défaut pour les tableaux et les hachages ou laisser les variables partagées entre des interpréteurs coopérant. Comme l'interface de type, l'interface d'attribut doit être perçue comme une interface générique, une sorte de cadre de travail pour inventer de nouvelles syntaxes et sémantiques. Nous ne savons pas comment Perl évoluera dans les dix prochaines années. Nous savons seulement que nous pouvons le rendre plus simple pour nous-mêmes en planifiant cela à l'avance.

Voir également `local`, `our` et la section *Déclarations avec portée* au chapitre 4.

new

```
new NOM_CLASSE  LISTE
new NOM_CLASSE
```

Il n'existe pas de fonction interne `new`. Il s'agit plutôt d'un constructeur de méthode ordinaire (c'est-à-dire un sous-programme défini par l'utilisateur) qui est défini ou hérité par la classe *`NOM_CLASSE`* (c'est-à-dire le paquetage) pour vous laisser construire des objets de type *`NOM_CLASSE`*. Beaucoup de constructeurs sont appelés « `new` », mais seulement par convention, juste pour jouer un tour aux programmeurs C++ en leur faisant croire qu'ils savent ce qu'il se passe. Lisez toujours la documentation de la classe en question pour savoir comment appeler ses constructeurs ; par exemple, le constructeur qui crée une « list box » dans le widget `Tk` est juste appelé `Listbox()`. Voir le chapitre 12.

next $@

```
next ETIQUETTE
next
```

L'opérateur `next` est semblable à l'instruction continue en C ; il démarre l'itération suivante de la boucle désignée par *`ETIQUETTE`* :

```
LIGNE: while (<STDIN>) {
    next LIGNE if /^#/;    # élimine les commentaires
    ...
}
```

S'il y avait un bloc `continue` dans cet exemple, il s'exécuterait immédiatement après l'invocation de `next`. Lorsqu'*`ETIQUETTE`* est omise, la commande se réfère à la boucle la plus intérieure l'encadrant.

Un bloc en soi-même est sémantiquement identique à une boucle qui ne s'exécute qu'une seule fois. Ainsi `next` peut quitter de manière anticipée un tel bloc (*via* le bloc `continue`, s'il y en a un).

`next` ne peut pas être employé pour sortir d'un bloc qui renvoie une valeur, comme `eval {}`, `sub {}` ou `do {}`, et ne doit pas être utilisé pour sortir d'une opération `grep` ou `map`. Si les avertissements sont actifs, Perl vous préviendra si vous faites un `next` pour sortir d'une boucle qui est hors de votre portée lexicale, comme une boucle dans une sous-routine appelante. Voir la section *Instructions de boucle* au chapitre 4.

no

$@

```
no MODULE LISTE
```

Voir l'opérateur `use`, qui est l'inverse de `no`, plus ou moins. La plupart des modules standard ne « non-importent » rien, faisant de `no` une « non-opération ». Les modules pragmatiques ont tendance ici à être plus serviables. Si le *`MODULE`* n'est pas trouvé, une exception est levée.

oct

$_

```
oct EXPR
oct
```

Cette fonction interprète *`EXPR`* comme une chaîne octale et renvoie la valeur décimale équivalente. S'il se trouve que *`EXPR`* commence par « `0x` », on l'interprète comme une chaîne hexadécimale à la place. Si *`EXPR`* débute par « `0b` », on l'interprète comme une chaîne de chiffres binaires. Ce qui suit convertit proprement en nombres n'importe quelle chaîne en base décimale, binaire, octale ou hexadécimale dans la notation standard C ou C++ :

```
$val = oct $val id $val =~ /^0/;
```

Pour exécuter la fonction inverse, utilisez sprintf avec le format approprié :

```
$perms = (stat("nom_fichier"))[2] & 07777;
$perms_oct = sprintf "%lo", $perms;
```

La fonction `oct` est habituellement employée lorsqu'une donnée sous forme de chaîne, comme « `644` » doit être convertie vers un mode de fichier, par exemple. Bien que Perl convertisse automatiquement les chaînes en nombres si besoin est, cette conversion automatique suppose que l'on soit en base 10.

open

$! X ARG X U X T

```
open HANDLE_FICHIER, MODE, LISTE
open HANDLE_FICHIER, EXPR
open HANDLE_FICHIER
```

La fonction `open` associe un *`HANDLE_FICHIER`* interne à la spécification d'un fichier ex-

terne donnée par *EXPR* ou *LISTE*. Elle peut être appelée avec un, deux ou trois arguments (ou plus si le troisième argument est une commande et que vous lancez au moins la version 5.6.1 de Perl). Si trois arguments ou plus sont présents, le deuxième argument spécifie le *MODE* d'accès avec lequel le fichier doit être ouvert, et le troisième argument (*LISTE*) fournit le véritable nom de fichier ou la commande à exécuter, en fonction du *MODE*. Dans le cas d'une commande, des arguments supplémentaires peuvent être fournis si vous désirez invoquer la commande directement sans impliquer un shell, un peu comme `system` ou `exec`. Sinon, la commande peut être fournie comme un seul argument (le troisième), auquel cas la décision d'invoquer le shell dépend du fait que la commande contienne ou non des métacaractères du shell. (N'utilisez pas plus de trois arguments si les arguments sont des fichiers ordinaires ; cela ne fonctionnera pas.) Si le *MODE* n'est pas reconnu, `open` lève une exception.

Si seulement deux arguments sont présents, on suppose que le mode et le nom de fichier ou de commande sont assemblés dans le second argument. (Et si vous ne spécifiez aucun mode dans le second argument mais uniquement un nom de fichier, le fichier est alors ouvert en lecture seule pour raison de sécurité.)

Avec un seul argument, la variable scalaire de paquetage qui porte le même nom (la variable, pas le paquetage !) que le *HANDLE_FICHIER*, doit contenir le nom du fichier et le mode optionnel :

```
$LOG = ">fichier_journal";    # $LOG ne doit pas être déclarée avec my !
    open LOG or die "Impossible d'ouvrir le fichier journal : $!";
```

Mais ne faites pas cela ! Ça manque de style. Oubliez que nous en avons parlé.

La fonction `open` renvoie vrai si elle réussit, sinon `undef`. Si le `open` commence par un pipe sur un processus fils, la valeur de retour sera le PID de ce nouveau processus. Comme avec tout appel système, vérifiez toujours la valeur de retour de `open` pour être sûr qu'il a marché. Mais ce n'est pas du C ou du Java, alors n'employez pas d'instruction `if` alors que l'opérateur `or` fait très bien l'affaire. Vous pouvez également employer `||`, mais dans ce cas, utilisez des parenthèses autour du `open`. Si vous choisissez d'omettre les parenthèses autour de l'appel de fonction pour le changer en opérateur de liste, prenez garde à bien utiliser « `or die` » après la liste plutôt que « `|| die` », car la préséance de `||` est supérieure à celle des opérateurs de liste comme `open`, et le `||` serait lié à votre dernier argument et non à l'`open` tout entier :

```
open LOG, ">fichier_journal"
    || die "Impossible de créer le fichier journal : $!"; # FAUX
open LOG, ">fichier_journal"
    or die "Impossible de créer le fichier journal : $!"; # ok
```

Cela semble plutôt exagéré, mais généralement vous devriez insérer un quelconque caractère d'espacement pour que la fin de l'opérateur de liste vous saute aux yeux :

```
open LOG, ">fichier_journal"
    or die "Impossible de créer le fichier journal : $!";
```

Comme le montre cet exemple, l'argument *HANDLE_FICHIER* n'est souvent qu'un simple identificateur (normalement en majuscules), mais il peut aussi être une expression dont la valeur donne une référence vers le véritable handle de fichier. (La référence peut être soit une référence symbolique vers le nom du handle de fichier, soit une référence en dur vers tout objet qui puisse être interprété comme un handle de fichier.) On appelle

ceci un *handle de fichier indirect*, et toute fonction qui prend comme premier argument un *HANDLE_FICHIER* accepte les handles de fichiers aussi bien indirects que directs. Mais open est spécial, car si vous lui donnez une variable indéfinie comme handle de fichier indirect, Perl définira automatiquement cette variable pour vous, c'est-à-dire qu'elle sera autovivifiée pour contenir la référence de handle de fichier adéquat. Ceci a l'avantage de fermer automatiquement le handle de fichier lorsqu'il n'y a plus de référence sur lui, généralement lorsque la variable quitte la portée courante :

```
{
        my $hf;                         # (non initialisée)
        open($hf, ">fichier_journal")  # $hf est autovivifié
            or die "Impossible de créer le fichier journal : $!";
        ...                             # faites ce que vous voulez avec $hf
}                                       # fermeture de $hf ici
```

On peut lisiblement incorporer la déclaration my $hf à l'intérieur de l'open :

```
open my $hf, ">fichier_journal" or die
```

Le symbole > que vous venez de voir devant le nom de fichier est un exemple de mode. Historiquement, la forme avec deux arguments de open est apparue en premier. Le récent ajout de la forme avec trois arguments vous permet de séparer le mode du nom de fichier, qui a l'avantage d'éviter toute confusion possible entre les deux. Dans l'exemple suivant, nous savons que l'utilisateur n'essaie pas d'ouvrir un nom de fichier commençant par « > ». Nous pouvons être sûrs qu'il spécifie un *MODE* valant « > », ce qui ouvre en écriture le fichier indiqué par *EXPR*, en le créant s'il n'existait pas et en l'effaçant s'il existait déjà :

```
open(LOG, ">", "fichier_journal")
    or die "Impossible de créer le fichier journal : $!";
```

Dans les formes plus courtes, le nom de fichier et le mode se trouvent dans la même chaîne. La chaîne est analysée un peu comme un shell typique traite les redirections de fichiers et de pipes. La chaîne est alors examinée, par n'importe quelle extrémité si besoin, pour trouver les caractères spécifiant comment le fichier doit être ouvert. Un espace est autorisé entre le mode et le nom de fichier.

Les modes qui indiquent comment ouvrir un fichier ressemblent aux symboles de redirection du shell. Le *tableau 29-1* donne une liste de ces symboles. (Pour accéder à un fichier en combinant des modes d'ouvertures non présentés dans cette table, voir la fonction de bas niveau sysopen.)

Tableau 29-1. Modes pour open

Mode	Accès en lecture	Accès en écriture	Ajout uniquement	Créer si inexistant	Vidage de l'existant
< *CHEMIN*	O	N	N	N	N
> *CHEMIN*	N	O	N	O	O
>> *CHEMIN*	N	O	O	O	N
+< *CHEMIN*	O	O	N	N	N
+> *CHEMIN*	O	O	N	O	O
+>> *CHEMIN*	O	O	O	O	N

Tableau 29-1. Modes pour open

Mode	Accès en lecture	Accès en écriture	Ajout uniquement	Créer si inexistant	Vidage de l'existant
| *COMMANDE*	N	O	aucun	aucun	aucun
COMMANDE |	O	N	aucun	aucun	aucun

Si le mode est « < » ou rien du tout, un fichier existant est ouvert en lecture. Si le mode est « > », le fichier est ouvert en écriture, ce qui tronque les fichiers existants et crée ceux qui n'existent pas. Si le mode est « >> », le fichier est créé si besoin et ouvert en ajout, et toutes les écritures sont automatiquement placées à la fin du fichier. Si un nouveau fichier est créé parce que vous avez utilisé « > » ou « >> » et que le fichier n'existait pas auparavant, les permissions d'accès dépendront de l'`umask` actuel du processus avec les règles décrites pour cette fonction. Voici quelques exemples courants :

```
open(INFO,        "fichier_de_donnees")
    || die("Impossible d'ouvrir le fichier de données : $!");
open(INFO,      "< fichier_de_donnees")
    || die("Impossible d'ouvrir le fichier de données : $!");
open(RESULTATS, "> stats_en_cours")
    || die("Impossible d'ouvrir les stats en cours : $!");
open(LOG,       ">> fichier_journal")
    || die("Impossible d'ouvrir le fichier journal : $!");
```

Si vous préférez la version avec moins de ponctuation, vous pouvez écrire :

```
open INFO,        "fichier_de_donnees"
     or die("Impossible d'ouvrir le fichier de données : $!");
open INFO,      "< fichier_de_donnees"
     or die("Impossible d'ouvrir le fichier de données : $!");
open RESULTATS, "> stats_en_cours"
     or die("Impossible d'ouvrir les stats en cours : $!");
open LOG,       ">> fichier_journal"
     or die("Impossible d'ouvrir le fichier journal : $!");
```

Lors d'une ouverture en lecture, le nom de fichier spécial « - » se réfère à `STDIN`. Lorsqu'il est ouvert en écriture, ce même nom de fichier spécial se réfère à `STDOUT`. Normalement, on les spécifie respectivement avec « <- » et « -> ».

```
open(ENTREE, "-" ) or die; # rouvre l'entrée standard en lecture
open(ENTREE, "<-") or die; # idem mais explicitement
open(SORTIE, "->") or die; # rouvre la sortie standard en écriture
```

De cette façon, l'utilisateur peut fournir à un programme un nom de fichier qui utilisera l'entrée ou la sortie standard, mais l'auteur du programme n'a pas besoin d'écrire de code particulier pour connaître ceci.

Vous pouvez également placer un «»+«» devant n'importe lequel de ces trois modes pour demander un accès en lecture/écriture simultané. Cependant, le fait que le fichier soit vidé ou créé et qu'il doit déjà exister est toujours déterminé par votre choix du signe inférieur ou supérieur. Cela signifie que « +< » est toujours un meilleur choix pour des mises à jour en lecture/écriture, plutôt que le mode douteux « +> » qui viderait tout d'abord le fichier avant que vous puissiez y lire quoique ce soit. (N'utilisez ce mode que

si vous ne voulez relire que ce que vous venez d'écrire.)

```
open(DBASE, "+<base_de_donnees")
    or die "Impossible d'ouvrir la base de données en mise à jour : $!";
```

Vous pouvez traiter un fichier ouvert en mise à jour comme une base de données en accès direct et utiliser `seek` pour vous déplacer vers un nombre d'octets particulier, mais les enregistrements de longueur variable rendent généralement irréalisables l'emploi du mode lecture/écriture pour mettre à jour de tels fichiers. Voir l'option de la ligne de commande **`-i`** au chapitre 19 pour une approche différente des mises à jour.

Si le premier caractère dans *`EXPR`* est un symbole pipe, `open` déclenche un nouveau processus et connecte un handle de fichier en écriture seule avec la commande. De cette façon, vous pouvez écrire dans ce handle et ce que vous y écrivez sera passé à l'entrée standard de cette commande. Par exemple :

```
open(IMPRIMANTE, "| lpr -Plp1) or die "Impossible de forker : $!";
print IMPRIMANTE "truc\n";
close(IMPRIMANTE) or die "Échec de la fermeture de lpr : $?/$!";
```

Si le dernier caractère de *`EXPR`* est un symbole pipe, `open` lancera ici aussi un nouveau processus, mais cette fois connecté à un handle de fichier en lecture seule. Ceci permet à tout ce que la commande écrit sur sa sortie standard d'être passé dans votre handle pour être lu. Par exemple :

```
open(RESEAU, "netstat -i -n |") or die "Impossible de forker : $!";
while (<RESEAU>) { ... }
close RESEAU or die "Impossible de fermer netstat : $!/$?";
```

La fermeture explicite de tout handle de fichier pipé entraîne l'attente du processus père de la terminaison du processus fils et le retour d'un code de statut dans `$?` (`$CHILD_ERROR`). Il est également possible que `close` positionne `$!` (`$OS_ERROR`). Voir les exemples dans les descriptions de `close` et de `system` pour savoir comment interpréter ces codes d'erreur.

Toute commande pipée contenant des métacaractères shell comme des jokers (*N.d.T.* : *wildcards*) ou des redirections d'entrée/sortie est passée à votre shell canonique (*/bin/sh* sur Unix), ces constructions spécifiques au shell sont donc analysées en premier lieu. Si aucun métacaractère n'est trouvé, Perl lance le nouveau processus lui-même sans appeler le shell.

Vous pouvez également utiliser la forme à trois arguments pour démarrer des pipes. En utilisant ce style, l'équivalent des ouvertures de pipe précédentes serait :

```
open(IMPRIMANTE, "|-", "lpr -Plp1") or die "Impossible de forker : $!";
open(RESEAU, "-|", "netstat -i -n") or die "Impossible de forker : $!";
```

Ici le caractère moins dans le deuxième argument représente la commande du troisième argument. Ces commandes n'invoquent pas le shell, mais si vous souhaitez qu'aucun traitement du shell ne survienne, les nouvelles versions de Perl vous permettent d'écrire :

```
open(IMPRIMANTE, "|-", "lpr", "-Plp1")
    or die "Impossible de forker : $!";
open(RESEAU, "-|", "netstat", "-i", "-n")
    or die "Impossible de forker : $!";
```

Si vous utilisez la forme à deux arguments pour ouvrir un pipe ou la commande spéciale « - »[7], un `fork` implicite est réalisé en premier lieu. (Sur les système qui ne savent pas `forker`, ceci lève une exception. Les systèmes Microsoft ne supportaient pas le `fork` avant la version 5.6 de Perl.) Dans ce cas, le caractère moins représente votre nouveau processus fils, qui est une copie du père. La valeur de retour de cet `open` forkant est l'ID du processus fils lorsqu'elle est examinée par le père, `0` lorsque c'est par le fils et la valeur indéfinie `undef` si le `fork` échoue — auquel cas, il n'y a pas de fils. Par exemple :

```
defined($pid = open(DEPUIS_LE_FILS, "-|"))
    or die "Impossible de forker : $!";

if ($pid) {
    @lignes_pere = <DEPUIS_LE_FILS>;   # code du père
} else {
    print STDOUT @lignes_fils;         # code du fils
}
```

Le handle de fichier se comporte normalement pour le père, mais pour le processus fils, l'entrée (ou la sortie) du père est pipée depuis (ou vers) le `STDOUT` (ou le `STDIN`) du fils. Le processus fils ne voit pas le handle de fichier du père comme étant ouvert. (Ceci est utilement indiqué par le PID de `0`.)

Généralement vous devriez utiliser cette construction à la place d'un `open` pipé classique lorsque vous voulez exercer plus de contrôle sur la manière dont la commande pipée s'exécute (comme lorsque vous exécutez setuid) et que vous ne désirez pas examiner si les commandes shells contiennent des métacaractères. Les `open` pipés suivant sont fortement équivalents :

```
open HF, "|            tr 'a-z' 'A-Z'";      # pipe vers une commande shell
open HF, "|-",        'tr', 'a-z', 'A-Z';    # pipe vers une commande nue
open HF, "|-" or exec 'tr', 'a-z', 'A-Z' or die; # pipe vers un fils
```

tout comme celles-ci :

```
open HF, "             cat -n 'fichier' |";  # pipe vers une commande shell
open HF, "-|",        'cat', '-n' 'fichier'; # pipe vers une commande nue
open HF, "-|" or exec 'cat', '-n', 'fichier' or die; # pipe vers un fils
```

Pour des utilisations plus élaborées de open forké, voir la section *Soliloque* au chapitre 16 et *Nettoyage de votre environnement* au chapitre 23, *Sécurité*.

Lorsque vous démarrez une commande avec `open`, vous devez choisir soit l'entrée soit la sortie : « `cmd|` » pour la lecture ou « `|cmd` » pour l'écriture. Vous ne pouvez pas utilisez `open` pour démarrer une commande qui possèdent des pipes à la fois en entrée et en sortie, comme la notation illégale (pour le moment) « `|cmd|` » semblerait l'indiquer. Cependant, les routines standard de la bibliothèque `IPC::Open2` et `IPC::Open3` vous donne une équivalence relativement proche. Pour plus de détail sur les pipes à double sens, voir la section *Communication bidirectionnelle* au chapitre 16.

Vous pouvez également spécifier, dans la plus pure tradition du Bourne Shell, une *`EXPR`* commençant par `>&`, auquel cas le reste de la chaîne est interprété comme le nom d'un

7. Ou vous pouvez la considérer comme la porte de sortie de la commande des formes à trois arguments ci-dessus.

handle de fichier (ou un descripteur de fichier, s'il est numérique) que l'on doit dupliquer en utilisant l'appel système *dup2*(2).[8] Vous pouvez utiliser `&` après `>`, `>>`, `+>`, `+>>` et `+<`. (Le mode spécifié doit correspondre au mode du handle de fichier originel.)

Une raison pour laquelle vous pourriez vouloir faire cela serait le cas où vous disposiez d'ores et déjà un handle de fichier ouvert et que vous vouliez obtenir un autre handle de fichier qui soit réellement un duplicata du premier.

```
open(SORTIE_SAUVEE, ">&ERREUR_SAUVEE")
    or die "Impossible de dupliquer ERREUR_SAUVEE : $!";
open(MON_CONTEXTE, "<&4")
    or die "Impossible de dupliquer le descfic4 : $!";
```

Cela signifie que si une fonction attend un nom de fichier, mais si vous ne voulez pas lui en donner parce que vous disposez déjà d'un fichier ouvert, vous n'avez qu'à passer le handle de fichier préfixé par une esperluette. Il vaut mieux tout de même utiliser un handle pleinement qualifié, pour le cas où la fonction se trouve dans un autre paquetage :

```
une_fonction("&main::FICHIER_JOURNAL");
```

Une autre raison de *dupliquer* un handle de fichier est de rediriger temporairement un handle de fichier existant sans perdre la trace de la destination d'origine. Voici un script qui sauvegarde, redirige et restaure `STDOUT` et `STDERR` :

```
#!/usr/bin/perl
open SORTIE_SAUVEE, ">&STDOUT";
open ERREUR_SAUVEE, ">&STDERR";

open STDOUT, ">truc.out" or die "Impossible de rediriger stdout";
open STDERR, ">&STDOUT"  or die "Impossible de dupliquer stdout";

select STDERR; $| = 1;      # active le vidage de tampon automatique
select STDOUT; $| = 1;      # active le vidage de tampon automatique

print STDOUT "stdout 1\n";  # ces flux d'entrée/sortie se propagent
print STDERR "stderr 1\n";  # également aux sous-processus
system ("une commande");    # utilise les nouveaux stdout/stderr

close STDOUT;
close STDERR;

open STDOUT, ">&SORTIE_SAUVEE";
open STDERR, ">&ERREUR_SAUVEE";

print STDOUT "stdout 2\n";
print STDERR "stderr 2\n";
```

Si le handle ou le descripteur de fichier est précédé par une combinaison `&=` au lieu d'un simple `&`, alors au lieu de créé un descripteur de fichier complètement neuf, Perl fait de

8. Cela ne fonctionne pas (pour le moment) avec les objets d'entrée/sortie sur les références de typeglobs par autovivification, mais vous pouvez toujours employer `fileno` pour aller chercher le descripteur de fichier et le dupliquer.

HANDLE_FICHIER un alias du descripteur existant en utilisant l'appel *fdopen*(3) de la bibliothèque C. Ceci est quelque peu plus parcimonieux en terme de ressources système, bien que ce ne soit plus un gros problème de nos jours.

```
$df = $ENV{"DF_MON_CONTEXTE};
open(MON_CONTEXTE, "<&=$num_df"
    or die "fdopen sur le descripteur $num_fd impossible : $!\";
```

Les handles de fichiers STDIN, STDOUT et STDERR restent toujours ouvert à travers un exec. Les autres handles de fichiers ne le restent pas, par défaut. Sur les systèmes qui supportent la fonction fcntl, vous pouvez modifiez le flag close-on-exec (*N.d.T.* : indicateur de fermeture lors d'un exec) pour un handle de fichier :

```
use Fcntl qw(F_GETFD F_SETFD);
$flags = fcntl(HF, F_SETFD, 0)
    or die "Impossible de nettoyer le flag close-on-exec pour HF : $!\n";
```

Voir aussi la variable spéciale $^F ($SYSTEM_MAX_FD) au chapitre 28.

Avec les formes à un ou à deux arguments de open, vous devez faire attention lorsque vous utilisez une variable chaîne comme nom de fichier, puisque la variable peut contenir arbitrairement des caractères étranges (particulièrement lorsque le nom du fichier a été obtenu à partir d'étranges caractères depuis Internet). Si vous n'y prenez pas garde, certaines parties du nom de fichier peuvent être interprétées comme une chaîne de *MODE* , des espaces à ignorer, une spécification de duplication ou un signe moins. Voici une manière historiquement intéressante de vous isoler :

```
$chemin =~ s#^(\s)#./$1#;
open (HF, "< $chemin\0") or die "Impossible d'ouvrir $chemin : $!";
```

Mais vous êtes toujours vulnérables de plusieurs manières. À la place, utiliser la forme à trois arguments de open pour ouvrir proprement tout nom fichier arbitraire et sans aucun risque (supplémentaire) concernant la sécurité :

```
open(HF, "<", $chemin) or die "Impossible d'ouvrir $chemin : $!";
```

D'un autre côté, si ce que vous recherchez est un véritable appel système dans le style de *open*(2) en C, avec les cloches et sirènes d'arrêt qui s'en suivent, alors investissez sur sysopen :

```
use Fcntl;
sysopen(HF, $chemin, O_RDONLY) or die "Impossible d'ouvrir $chemin : $!";
```

Si vous êtes sur un système qui fait la distinction entre les fichiers textes et binaires, vous pouvez avoir besoin de placer votre handle de fichier en mode binaire — ou oubliez de faire cela, comme cela peut s'avérer — pour éviter de mutiler vos fichiers. Sur de tels systèmes, si vous utilisez le mode texte avec des fichiers binaires, ou l'inverse, vous n'apprécierez certainement pas les résultats.

Les systèmes qui ont besoin de la fonction binmode se distinguent des autres par le format employé pour les fichiers textes. Ceux qui s'en passent, finissent chaque ligne avec un seul caractère correspondant à ce que le C pense être un saut de ligne, \n. Unix et Mac OS tombent dans cette catégorie. Les systèmes d'exploitation des autres variétés VMS, MVS, MS-biduletrucmachinchouette et S&M traitent différemment les entrées/sorties dans les fichiers textes et binaires, ils ont donc besoin de binmode.

Ou de son équivalent. Depuis la version 5.6 de Perl, vous pouvez spécifier le mode bi-

naire dans la fonction open sans un appel séparé à binmode. En tant que composantes de l'argument *MODE* (mais uniquement dans la forme à trois arguments), vous pouvez spécifier diverses disciplines d'entrées/sorties. Pour obtenir l'équivalent d'un binmode, employez la forme à trois arguments de open et donnez une discipline valant :raw après les autres caractères de *MODE* :

```
open(HF, "<:raw", $chemin) or die "Impossible d'ouvrir $chemin : $!";
```

Puisqu'il s'agit d'une toute nouvelle fonctionnalité, il existera certainement plus de disciplines au moment où vous lirez ceci que lorsque nous l'avons écrit. Cependant, nous pouvons raisonnablement prédire qu'il y aura en toute vraisemblance des disciplines qui ressembleront pour toutes ou partie à celles du *tableau 29-2*.

Tableau 29-2. Disciplines d'entrées/sorties

Discipline	Signification
:raw	Mode binaire ; ne pas traiter.
:text	Traitement par défaut de texte.
:def	Défaut déclaré par « use open ».
:latin1	Le fichier doit être en ISO-8859-1.
:ctype	Le fichier doit être en LC_TYPE.
:utf8	Le fichier doit être en UTF-8.
:utf16	Le fichier doit être en UTF-16.
:utf32	Le fichier doit être en UTF-32.
:uni	Pressent de l'Unicode (UTF-*).
:any	Pressent de l'Unicode/Latin1/LC-CTYPE.
:xml	Utilise l'encodage spécifié dans le fichier.
:crlf	Pressent des sauts de lignes.
:para	Mode paragraphe.
:slurp	Mode par avalement tout rond.

Vous serez capables d'empiler les disciplines susceptibles d'être empilées, ainsi, vous pourrez écrire par exemple :

```
open(HF, "<:para:crlf:uni", $chemin)
    or die "Impossible d'ouvrir $chemin : $!";
while ($para ) <HF>) { ... }
```

Ceci positionnera les disciplines :

- en lisant certaines formes d'Unicode et en les convertissant vers le format interne de Perl UTF-8 si le fichier n'est pas d'ores et déjà en UTF-8 ;
- en cherchant les variantes des séquences de fin de ligne et en les convertissant toutes vers \n, et ;
- en traitant le fichier paragraphe par paragraphe, un peu comme le fait $/="".

Si vous voulez positionner le mode d'ouverture par défaut (:def) à autre chose que :text, vous pouvez déclarer ceci au tout début de votre fichier avec le pragma open :

```
use open IN => ":any", OUT => ":utf8";
```

En fait, ce serait vraiment beau si un jour ou l'autre c'était la discipline `:text` par défaut. Ceci reflète bien l'esprit du dicton : « Soyez tolérant avec ce que vous acceptez et strict avec ce que vous produisez. »

opendir

$! X ARG X U X T

```
opendir HANDLE_REPERTOIRE, EXPR
```

Cette fonction ouvre un répertoire nommé *EXPR* pour le traiter par `readdir`, `telldir`, `seekdir`, `rewinddir` et `closedir`. La fonction renvoie vrai en cas de réussite. Les handles de répertoires possèdent leur propre espace de nommage, séparé de celui des handles de fichiers.

ord

$_

```
ord EXPR
ord
```

Cette fonction renvoie la valeur numérique (ASCII, Latin-1 ou Unicode) du premier caractère de *EXPR*. La valeur de retour est toujours non-signée. Si vous désirez une valeur signée, utilisez `unpack('c', EXPR)`. Si vous voulez convertir tous les caractères de la chaîne en une liste de nombres, utilisez `unpack('U*', EXPR)` à la place.

our

```
our TYPE EXPR : ATTRIBUTS
our EXPR : ATTRIBUTS
our TYPE EXPR
our EXPR
```

Un `our` déclare une ou plusieurs variables comme étant des variables globales valides à l'intérieur du bloc, du fichier ou de l'`eval` l'encadrant. C'est-à-dire que l'on détermine la visibilité de `our` avec les mêmes règles qu'une déclaration `my`. Mais `our` ne crée pas de variable privée ; il permet simplement l'accès sans entraves à la variable de paquetage globale existante. Si plus d'une valeur est listée, la liste doit être placée entre parenthèses.

L'utilisation principale d'un `our` est de cacher la variable des effets d'une déclaration `use strict "vars"` ; puisque la variable est masquée comme une variable `my`, vous avez la permission d'utiliser la variable déclarée globale sans la qualifier avec son paquetage. Cependant, tout comme une variable `my`, cela ne marche que dans la portée lexicale de la déclaration `our`. En regard de ceci, `our` diffère de `use vars`, qui affecte le paquetage dans son entier et qui n'a pas de portée lexicale.

`our` est également semblable à `my` en ce que vous êtes autorisé à déclarer une variable avec un *TYPE* et des *ATTRIBUTS*. Voici la syntaxe :

```
our Chien $medor :oreilles(courtes) :queue(longue);
```

Au moment où nous écrivons ceci, sa signification n'est pas tout à fait claire. Les attributs peuvent affecter soit la variable globale, soit l'interprétation locale de `$medor`. D'un

côté, cela ressemblerait fort aux variables `my` pour les attributs que l'on protège d'un voile dans la vue locale actuelle de `$medor`, sans interférer avec les autres vues de la variable globale à d'autres endroits. De l'autre côté, si un module déclare `$medor` comme étant un `Chien` et qu'un autre déclare `$medor` en tant que `Chat`, vous pouvez finir avec des chiens miaulant et des chats aboyant. C'est un sujet de recherche active, ce qui est une manière polie de dire que nous ne savons pas de quoi nous parlons à l'heure actuelle. (Excepté que nous savons quoi faire avec une déclaration de *`TYPE`* lorsque la variable se réfère à un pseudo-hachage — voir *Gestion des données d'instance* au chapitre 12.)

Une autre similitude entre `our` et `my` réside dans sa visibilité. Une déclaration `our` déclare une variable globale qui sera visible tout au long de sa portée lexicale, même au-delà des frontières de paquetages. Le paquetage dans lequel la variable est située est déterminé à l'endroit de la déclaration, pas à l'endroit où on l'utilise. Cela signifie que le comportement suivant est valide et qu'il est jugé être une fonctionnalité :

```
package Truc;
our $machin;       # $machin est $Truc::machin pour le reste de la
                   # portée lexicale
$machin = 582;

package Machin
print $machin;     # $affiche 582, comme si 'our' avait été 'my'
```

Toutefois, la distinction entre `my` qui crée une nouvelle variable privée et `our` qui expose une variable globale existante est importante, particulièrement pour les affectations. Si vous combinez une affectation à l'exécution avec une déclaration `our`, la valeur de la variable globale ne disparaîtra pas une fois que le `our` sortira de la portée. Pour cela, vous avez besoin de local :

```
($x, $y) = ("un", "deux");
print "avant le bloc, x vaut $x, y vaut $y\n";
{
    our $x = 10;
    local our $y = 20;
    print "dans le bloc, x vaut $x, y vaut $y\n";
}
print "après le bloc, x vaut $x, y vaut $y\n";
```

Cela affiche :

```
avant le bloc, x vaut un, y vaut deux
dans le bloc, x vaut 10, y vaut 20
après le bloc, x vaut 10, y vaut deux
```

De multiples déclarations `our` dans la même portée sont autorisées si elles se trouvent dans des paquetages différents. Si elles se produisent dans le même paquetage, Perl émettra des avertissements, si vous lui demandez de le faire.

```
use warnings;
package Truc;
our $machin;         # déclare $Truc::machin pour le reste de la
                     # portée lexicale
$machin = 20;
```

```
package Machin;
our $machin = 30;    # déclare $Machin::machin pour le reste de la
                     # portée lexicale
print $machin;       # affiche 30

our $machin;         # émet un avertissement
```

Voir aussi `local`, `my` et la section *Déclarations avec portée* au chapitre 4.

pack

`$@`

```
pack CANEVAS, LISTE
```

Cette fonction prend une *`LISTE`* de valeurs ordinaires Perl, les convertit en une chaîne d'octets en fonction du *`CANEVAS`* et renvoie cette chaîne. La liste d'arguments est complétée ou tronquée selon les besoins. C'est-à-dire que si vous fournissez moins d'arguments que n'en requiert le *`CANEVAS`*, `pack` suppose que les arguments additionnels sont nuls. À l'inverse, si vous fournissez plus d'arguments que n'en requiert le *`CANEVAS`*, les arguments supplémentaires sont ignorés. Les formats inconnus du *`CANEVAS`* lèveront des exceptions.

Le *`CANEVAS`* décrit une structure de chaîne en tant que séquence de champs. Chaque champ est représenté par un seul caractère qui décrit le type de la valeur et son encodage. Par exemple, un caractère de format valant `N` spécifie un entier non-signé de quatre octets dans l'ordre de la notation gros-boutiste (*N.d.T.* : *big-endian*).

Les champs sont empaquetés dans l'ordre donné par le *`CANEVAS`*. Par exemple, pour empaqueté un entier signé d'un octet et un nombre en virgule flottante avec une simple précision dans une chaîne, vous devez écrire :

```
$chaine = pack("Cf", 244, 3.14);
```

Le premier octet de la chaîne renvoyée a la valeur 244. Les octets restants représentent l'encodage de 3.14 en tant que nombre flottant en simple précision. L'encodage particulier du nombre à virgule flottante dépend du matériel de votre ordinateur.

Les points importants à considérer lorsque vous utilisez `pack` sont :

- le type de donnée (comme un entier, un nombre à virgule flottante ou une chaîne) ;
- l'intervalle de valeurs (comme lorsque vos entiers occupent un, deux, quatre ou peut-être même huit octets ; ou si vous empaquetez des caractères 8-bits ou Unicode) ;
- si vos entiers sont signés ou non ;
- l'encodage à employer (comme un empaquetage de bits ou d'octets en natif, en petit-boutiste ou en gros-boutiste.

Le *tableau 29-3* liste les caractères de format et leurs significations. (D'autres caractères peuvent apparaître dans les formats ; ils seront décrits plus tard.)

Vous pouvez librement placer des espaces et des commentaires dans vos *`CANEVAS`*. Les commentaires commencent par le symbole `#` habituel et se prolongent jusqu'à la première fin de ligne (s'il y en a) dans le *`CANEVAS`*. Chaque lettre peut être suivie par un nombre indiquant le *compteur*, que l'on interprète comme le compteur de répétitions

Tableau 29-3. Caractères de canevas pour pack/unpack

Caractère	Signification
a	Une chaîne d'octets, complétée par des nuls.
A	Une chaîne d'octets, complétée par des blancs.
b	Une chaîne de bits, dans l'ordre croissant des bits à l'intérieur de chaque octet (comme vec).
B	Une chaîne de bits, dans l'ordre décroissant des bits à l'intérieur de chaque octet.
c	Une valeur caractère signée (entier 8-bits).
C	Une valeur caractère non-signée (entier 8-bits) ; voir U pour Unicode.
d	Un nombre à virgule flottante en double précision au format natif.
f	Un nombre à virgule flottante en simple précision au format natif.
h	Une chaîne hexadécimale, quartet de poids faible d'abord.
H	Une chaîne hexadécimale, quartet de poids fort d'abord.
i	Une valeur entière signée au format natif.
I	Une valeur entière non-signée au format natif.
l	Une valeur entière longue signée, toujours de 32 bits.
L	Une valeur entière longue non-signée, toujours de 32 bits.
n	Un court de 16 bits, dans l'ordre « réseau » (gros-boutiste).
N	Un long de 32 bits, dans l'ordre « réseau » (gros-boutiste).
p	Un pointeur sur une chaîne terminée par un caractère nul.
P	Un pointeur sur une chaîne de longueur fixe.
q	Une valeur quad (entier de 64 bits) signée.
Q	Une valeur quad (entier de 64 bits) non-signée.
s	Une valeur entière courte signée, toujours de 16 bits.
S	Une valeur entière courte non-signée, toujours de 16 bits.
u	Une chaîne uuencodée.
U	Un nombre en caractère Unicode.
v	Un court de 16 bits dans l'ordre « VAX » (petit-boutiste).
V	Un long de 32 bits dans l'ordre « VAX » (petit-boutiste).
w	Un entier compressé au format BER.
x	Un octet nul (avance d'un octet).
X	Recule d'un octet.
Z	Une chaîne d'octets terminée par un caractère nul (et complétée par des nuls).
@	Remplissage avec un caractère nul à la position absolue.

ou la longueur en quelque sorte, selon le format. Avec tous les formats exceptés a, A, b, B, h, H, P et Z, compteur est un compteur de répétition, pack engloutit autant de valeurs dans la *LISTE*. Un * pour le compteur signifie autant d'entités qu'il en reste.

Les formats `a`, `A` et `Z` n'engloutissent qu'une seule valeur, mais l'empaquètent comme une chaîne d'octets de longueur *compteur*, en la complétant avec des caractères nuls ou des espaces si besoin. Lorsqu'on dépaquète, `A` dépouille les espaces et les caractères nuls qui traîne, `Z` dépouille tout ce qui se trouve après le premier caractère nul et `a` renvoie la donnée littérale indemne. Lorsqu'on empaquète, `a` et `Z` sont équivalents.

De même, les formats `b` et `B` empaquètent une chaîne longue de *compteur* bits. Chaque octet du champ d'entrée génère 1 bit dans le résultat basé sur le bit le moins significatif de chaque octet d'entrée (c'est-à-dire, sur `ord($octet) % 2`). Commodément, cela signifie que les octets `0` et `1` génèrent les bits 0 et 1. En partant du début de la chaîne d'entrée, chaque 8-tuple d'octets est convertit vers un unique octet en sortie. Si la longueur de la chaîne d'entrée n'est pas divisible par 8, le reste est empaqueté comme s'il était complété par des 0s. De même, durant le dépaquetage avec `unpack` les bits supplémentaires sont ignorés. Si la chaîne d'entrée est plus longue que nécessaire, les octets supplémentaires sont ignorés. Un * comme *compteur* signifie l'utilisation de tous les octets du champ d'entrée. Durant le dépaquetage avec `unpack`, les bits sont convertis vers une chaîne de `0`s et de `1`s.

Les formats `h` et `H` empaquètent une chaîne de *compteur* quartets (groupes de 4 bits souvent représentés comme des chiffres hexadécimaux).

Le format `p` empaquète un pointeur sur une chaîne terminée par un caractère null. Il vous incombe de vous assurer que la chaîne n'est pas une valeur temporaire (qui est susceptible d'être désallouée avant que vous n'ayez fini d'utiliser le résultat empaqueté) Le format `P` empaquète un pointeur vers une structure dont la taille est indiquée par *compteur*. Un pointeur nul est créé si la valeur correspondante pour `p` ou `P` est `undef`.

Le caractère `/` permet d'empaqueter et de dépaqueter des chaînes dans lesquelles la structure empaquetée contient un compteur sur un octet suivi par la chaîne elle-même. On écrit *`entité-longueur`*`/`*`entité-chaîne`*. L'*`entité-longueur`* peut être n'importe quelle lettre canevas de `pack` et décrit comment la valeur représentant la longueur est empaquetée. Les canevas les plus susceptibles d'être utilisés sont ceux empaquetant des entiers comme `n` (pour les chaînes Java), `w` (pour de l'ASN.1 ou du SNMP) et `N` (pour du XDR de Sun). L'*`entité-chaîne`* doit, à présent, être `A*`, `a*` ou `Z*`. Pour `unpack`, la longueur de la chaîne est obtenue à partir de l'*`entité-longueur`*, mais si vous avez mis un *, elle sera ignorée.

```
unpack 'C/a', "\04Gurusamy";              # donne 'Guru'
unpack 'a3/A* A*', '007 Bond J ';         # donne ('Bond','J')
pack 'n/a* w/a*','bonjour,','le monde';   # donne "bonjour, le monde"
```

unpack ne renvoie pas explicitement l'*`entité-longueur`*. L'ajout d'un *compteur* à la lettre de l'*`entité-longueur`* n'est pas susceptible d'être utile à quoique ce soit, sauf si cette lettre est `A`, `a` ou `Z`. L'empaquetage avec une *`entité-longueur`* de `a` ou `Z` introduit des caractères nuls (`\0`), que Perl ne considère pas comme légaux dans les chaînes numériques.

Les formats d'entiers `s`, `S`, `l` et `L` peuvent être immédiatement suivis par un `!` pour signifier des entiers courts ou longs natifs à la place de mesurer exactement 16 et 32 bits, respectivement. À ce jour, c'est un problème principalement pour les plates-formes 64 bits, où les entiers cours et longs natifs tel que les voient le compilateur local C peuvent être complètement différents de ces valeurs. (`i!` et `I!` fonctionnent mais seulement à cause de la complétude ; ils sont identiques à `i` et `I`.)

La taille réelle (en octets) des entiers, des courts, des longs et des « longs longs » natifs sur la plate-forme où Perl a été compilé est également disponible *via* le module `Config` :

```
use Config;
print $Config{shortsize},    "\n";
print $Config{intsize},      "\n";
print $Config{longsize},     "\n";
print $Config{longlongsize}, "\n";
```

Le simple fait que *Configure* connaisse la taille d'un « long long », n'implique pas forcément que vous disposiez des formats `q` et `Q`. (Certains systèmes en disposent, mais vous n'en possédez certainement pas. Du moins pas encore.)

Les formats d'entiers d'une longueur plus grande qu'un seul bit (`s`, `S`, `i`, `I`, `l` et `L`) sont intrinsèquement non portables entre processeurs différents, car ils obéissent à l'ordre des octets natifs et au paradigme petit-boutiste/gros-boutiste. Si vous voulez des entiers portables, utilisez les formats `n`, `N`, `v` et `V` ; l'ordre de leurs bits et leur taille sont connus.

Les nombres à virgules flottantes ne se trouvent que dans le format natif de la machine. À cause de la variété des formats des flottants et de l'absence d'une représentation « réseau » standard, il n'existe pas d'inter-échanges facilités. Cela signifie que si l'on écrit des données à virgule flottante empaquetées sur une machine, elles pourront ne pas être lisibles sur une autre. Ceci est un problème même lorsque les deux machines utilise l'arithmétique flottante IEEE, car l'ordre (petit-boutiste/gros-boutiste) de la représentation en mémoire ne fait pas partie de la spécification IEEE.

Perl utilise en interne des doubles pour tous les calculs en virgule flottante, donc la conversion depuis un double vers un float, puis la conversion inverse perdra de la précision. Cela signifie que `unpack("f", pack("f", $truc))` ne sera généralement pas égal à `$truc`.

Vous êtes responsable de toute considération concernant les alignements ou les remplissages exigés par d'autres programmes, particulièrement par ces programmes créés par un compilateur C avec sa propre idiosyncrasie en matière de présentation d'une `struct` C sur l'architecture particulière en question. Vous devrez ajouter suffisamment de `x` durant l'empaquetage pour compenser ceci. Par exemple, une déclaration C de :

```
struct machin {
    unsigned char c;
    float f;
}
```

pourrait être écrite dans un format « `C x f` », ou « `C x3 f` », ou encore « `f C` » — pour n'en citer que quelques uns. Les fonctions `pack` et `unpack` gèrent leurs entrées et leurs sorties comme des séquences plates d'octets car il n'y a aucun moyen pour elles de savoir où vont les octets, n d'où ils viennent.

Regardons quelques exemples. Cette première paire empaquète des valeurs numériques dans des octets :

```
$sortie = pack "CCCC", 65, 66, 67, 68;    # $sortie eq "ABCD"
$sortie = pack "C4", 65, 66, 67, 68;      # idem
```

Celui-ci fait la même chose avec des lettres Unicode :

```
$machin = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
```

Celui-là est similaire, avec une paire de caractères nuls rentrant en jeu :

```
$sortie = pack "CCxxCC", 65, 66, 67, 68   # $sortie eq "AB\0\0CD"
```

Empaqueter vos entiers courts n'implique pas que vous soyez portable :

```
$sortie = pack "s2", 1, 2;    # "\1\0\2\0" en petit-boutiste
                              # "\0\1\0\2" en gros-boutiste
```

Dans les paquets binaires et hexadécimaux, le *compteur* se réfère au nombre de bits ou de quartets, pas au nombre d'octets produits :

```
$sortie = pack "B32", "01010000011001010111001001101100";
$sortie = pack "H8", "5065726c";      # les deux donnent "Perl"
```

La longueur dans un champ `a` ne s'applique qu'à une seule chaîne :

```
$sortie = pack "a4", "abcd", "x", "y", "z";      # "abcd"
```

Pour contourner cette limitation, utilisez plusieurs spécificateurs :

```
$sortie = pack "aaaa", "abcd", "x", "y", "z";    # "axyz"
$sortie = pack "a" x 4, "abcd", "x", "y", "z";   # "axyz"
```

Le format `a` remplit avec des caractères nuls :

```
$sortie = pack "a14", "abcdefg";      # "abcdefg\0\0\0\0\0\0\0"
```

Ce canevas empaquète un enregistrement de `struct tm` C (du moins sur certains systèmes) :

```
$sortie = pack "i9pl", gmtime(), $tz, $toff;
```

Généralement, le même canevas peut également être utilisé dans la fonction `unpack`, bien que certains formats se comportent différemment, notamment `a`, `A` et `Z`.

Si vous voulez joindre des champs de texte de taille fixe, utilisez `pack` avec un *`CANEVAS`* de format `A` ou `a` :

```
$chaine = pack("A10" x 10, @data);
```

Si vous voulez joindre des champs de texte de taille variable avec un séparateur, utilisez plutôt la fonction `join`:

```
$chaine = join(" and ", @donnee);
$chaine = join("", @donnee);       # séparateur nul
```

Bien que tous nos exemples utilisent des chaînes littérales comme canevas, il n'y a rien qui vous interdise d'aller chercher vos données dans un fichier sur disque. Vous pouvez construire une base de données relationnelle complète autour de cette fonction. (Nous ne nous étendrons pas sur ce que cela prouverait sur votre nature.)

package

```
package ESPACE_DE_NOM
package
```

Il ne s'agit pas vraiment d'une fonction, mais d'une déclaration qui affirme que le reste de la portée la plus intérieure l'encadrant appartient à la table de symboles ou l'espace de nom, indiqué. (La portée d'une déclaration `package` est ainsi la même que la portée d'une déclaration `my` ou `our`.) À l'intérieur de cette portée, la déclaration force le com-

pilateur à résoudre tous les identificateurs non qualifiés en les recherchant dans la table de symboles du paquetage déclaré.

Une déclaration `package` n'affecte que les variables globales — y compris celles sur lesquelles vous avez employé `local` — et pas les variables lexicales déclarées avec `my`. Elle affecte seulement les variables globales non qualifiées ; celles qui sont qualifiées avec leur propre nom de paquetage ignorent le paquetage actuellement déclaré. Les variables globales déclarées avec `our` ne sont pas qualifiées et respectent par conséquent le paquetage courant, mais seulement à l'endroit de leur déclaration, après quoi elles se comportent comme des variables `my`. C'est-à-dire que pour le reste de leur portée lexicale, les variables `our` sont « clouées » au paquetage courant au moment de leur déclaration, même si une déclaration de paquetage ultérieure intervient.

La déclaration `package` est généralement la première d'un fichier à inclure avec un opérateur `require` ou `use`, bien qu'elle puisse se trouver en tout endroit légal pour une instruction. Lorsqu'on crée un fichier de module traditionnel ou orienté objet, il est courant d'appeler le paquetage avec le même nom que le fichier pour éviter toute confusion. (Il est également courant de mettre la première lettre du nom du paquetage en majuscule car les modules en minuscules sont interprétés par convention comme étant des pragmas.)

Vous pouvez passer dans un paquetage donné par plus plus d'un endroit ; cela influence le choix de la table des symboles utilisée par le compilateur pour le reste de ce bloc. (Si le compilateur rencontre un autre paquetage déclaré au même niveau, la nouvelle déclaration écrase l'ancienne.) Votre programme principal est censé commencer par une déclaration `package main`.

Vous pouvez vous référer à des variables, des sous-programmes, des handles de fichier et des formats appartenant à d'autres paquetages en préfixant l'identificateur avec le nom du paquetage et un double deux-points : `$Paquetage::Variable`. Si le nom du paquetage est nul, le paquetage principal est pris par défaut. C'est-à-dire que `$::heureuse` est équivalente à `$main::heureuse`, tout comme à `$main'heureuse`, que l'on rencontre encore parfois dans d'anciens codes.

Voici un exemple :

```
package main; $heureuse = "un poker de dames";
package Donne; $heureuse = "truquée";
package Nimporte;
print "Ma main heureuse est $main::heureuse.\n";
print "La donne heureuse est $Donne::heureuse.\n";
```

Ce qui affiche :

```
Ma main heureuse est un poker de dames.
Ma donne heureuse est truquée.
```

La table de symboles d'un paquetage est stockée dans un hachage avec un nom se terminant par un double deux-points. La table de symboles du paquetage principal est par exemple appelée `%main::`. On peut donc aussi accéder au symbole `*main::heureuse` par `$main::{"heureuse"}`.

Si *`ESPACE_DE_NOM`* est omis, il n'y aura alors aucun paquetage courant et tous les identifiants devront être pleinement qualifiés ou déclarés comme lexicaux. C'est encore plus strict qu'un `use strict`, puisque cela s'étend également aux noms de fonctions.

Voir le chapitre 10, *Paquetages*, pour plus d'informations sur les paquetages. Voir `my` plus haut dans ce chapitre pour d'autres considérations sur les portées.

pipe

$! X ARG X U

```
pipe HANDLE_LECTURE, HANDLE_ECRITURE
```

Comme l'appel système correspondant, cette fonction ouvre deux pipes connectés, voir *pipe*(2). Cet appel est presque toujours employé avant un `fork`, après lequel le lecteur du pipe doit fermer *`HANDLE_ECRITURE`* et celui qui écrit doit fermer *`HANDLE_LECTURE`*. (Sinon le pipe n'indiquera pas la fin de fichier à celui qui lit quand celui qui écrit le fermera.) Si vous lancez une boucle de processus par des pipes, un verrou mortel peut se produire à moins d'être très prudent. Notez de plus que les pipes de Perl utilisent des tampons d'entrées/sorties standard et qu'il vous faut donc positionner `$|` (`$OUTPUT_AUTOFLUSH`) sur votre *`HANDLE_ECRITURE`* pour vider les tampons après chaque opération de sortie, selon l'application - voir `select`(handle de fichier de sortie).

(Comme avec `open`, si l'un des handle de fichier est indéfini, il sera autovivifié.)

Voici un petit exemple :

```
pipe(LISEZ_MOI, ECRIVEZ_MOI);
unless ($pid = fork) {       # fils
    defined $pid or die "Impossible de forker : $!";
    close(LISEZ_MOI);
    for $i (1..5) { print ECRIVEZ_MOI "ligne $i\n" }
    exit;
}
$SIG{CHLD} = sub { waitpid($pid, 0) };
close(ECRIVEZ_MOI);
@chaines = <LISEZ_MOI>;
close(LISEZ_MOI);
print "Reçu: \n", @chaines;
```

Remarquez comment l'écrivain ferme l'extrémité de lecture et comment à l'inverse le lecteur ferme l'extrémité d'écriture. Vous ne pouvez pas utiliser un seul pipe pour une communication à double sens. Utilisez pour cela soit deux pipes différents, soit l'appel système `socketpair`. Voir la section *Pipes* au chapitre 16.

pop

```
pop TABLEAU
pop
```

Cette fonction traite un tableau comme une pile — elle dépile (enlève) et renvoie la dernière valeur du tableau en le raccourcissant d'un élément. Si *`TABLEAU`* est omis, la fonction dépile `@_` dans la portée lexicale des sous-programmes et formats ; elle dépile `@ARGV` dans la portée des fichiers (généralement le programme principal) ou à l'intérieur de la portée lexicale établie par les constructions `eval CHAINE`, `BEGIN {}`, `CHECK {}`, `INIT {}` et `END {}`. Elle entraîne les mêmes effets que :

```
$tmp = $TABLEAU{$#TABLEAU--};
```

ou que :

```
$tmp = splice @TABLEAU, -1;
```

S'il n'y a pas d'élément dans le tableau, `pop` renvoie `undef`. (Mais ne vous basez pas sur ceci pour vous dire que le tableau est vide si votre tableau continent des valeurs `undef` !) Voir également `push` et `shift`. Pour dépiler plus d'un élément, utilisez `splice`.

Le premier argument de `pop` doit être un tableau et non une liste. Pour obtenir le dernier élément d'une liste, utilisez ceci :

```
( LISTE  )[-1]
```

pos

$_

```
pos SCALAIRE
pos
```

Cette fonction renvoie l'endroit de *SCALAIRE* où la dernière recherche `m//g` s'est arrêtée. Elle renvoie le décalage du caractère *suivant* celui qui correspondait. (C'est-à-dire que c'est équivalent à `length($‘) + length($&)`.) Il s'agit du décalage où la prochaine recherche `m//g` sur cette chaîne démarrera. Rappelez-vous que le décalage début de la chaîne est `0`. Par exemple :

```
$grafitto = "fee fie foe foo";
while ($grafitto =~ m/e/g) {
    print pos $grafitto, "\n";
}
```

affiche `2`, `3`, `7` et `11`, les décalages de chacun des caractères suivant un « e » . On peut assigner à `pos` une valeur pour indiquer à `m//g` d'où partir :

```
$grafitto = "fee fie foe foo";
pos $grafitto = 4; # Sauter le fee, commencer à fie
while ($grafitto =~ m/e/g) {
    print pos $grafitto, "\n";
}
```

Cela n'affiche que `7` et `11`. L'assertion `\G` ne recherche de correspondance qu'à l'endroit spécifié par `pos` à ce moment dans la chaîne dans laquelle on recherche. Voir la section *Positions* au chapitre 5.

print

$_ $! X ARG

```
print HANDLE _FICHIER  LISTE
print LISTE
print
```

Cette fonction affiche une chaîne ou une liste de chaînes séparées par des virgules. Si la variable `$\` (`$OUTPUT_RECORD_SEPARATOR`) est positionnée, son contenu sera affiché implicitement à la fin de la liste. La fonction renvoie vrai en cas de réussite, sinon faux. *HANDLE_FICHIER* peut être un nom de variable scalaire (non indicée), auquel cas la variable contient la nom du véritable handle de fichier ou une référence à un objet handle de fichier d'une quelconque sorte. *HANDLE_FICHIER* peut également être un bloc qui renvoie une telle valeur :

```
print { $OK ? "STDOUT" : "STDERR" } "$truc\n";
print { $handle_entrees_sorties[$i] } "$truc\n";
```

Si *HANDLE_FICHIER* est une variable et que le token suivant est un terme, elle peut être prise à tort pour un opérateur à moins que vous n'interposiez un + ou ne mettiez des parenthèses autour des arguments. Par exemple :

```
print $a - 2;   # envoie $a - 2 sur le handle par défaut (en général STDOUT)
print $a (- 2); # envoie -2 sur le handle de fichier spécifié par $a
print $a -2;    # idem (règles d'analyse bizarres :-)
```

Si *HANDLE_FICHIER* est omis, la fonction affiche sur le handle de fichier de sortie actuellement sélectionné, initialement STDOUT. Pour mettre le handle de fichier de sortie par défaut à autre chose que STDOUT, utilisez l'opération `select` *HANDLE_FICHIER*.[9] Si *LISTE* est également omise, la fonction affiche `$_`. Comme `print` prend une *LISTE* en argument, tout élément de cette *LISTE* est évalué dans un contexte de liste. Ainsi quand vous écrivez :

```
print SORTIE <STDIN>;
```

ce n'est pas la ligne suivante de l'entrée standard qui va être affichée, mais toutes les lignes de l'entrée standard jusqu'à la fin du fichier, puisque c'est ce que <STDIN> renvoie dans un contexte de liste. Si vous voulez autre chose, écrivez :

```
print SORTIE scalar <STDIN>;
```

De même, en se souvenant de la règle si-ça-ressemble-à-une-fonction-c'est-une-fonction, faites attention à ne pas faire suivre le mot-clé `print` d'une parenthèse gauche à moins que la parenthèse droite correspondante ne termine les arguments du `print`- interposez un + ou mettez des parenthèses à tous les arguments :

```
print (1+2)*3, "\n";   # mauvais
print +(1+2)*3, "\n";  # ok
print ((1+2)*3, "\n"); # ok
```

printf

$_ $! x ARG

```
printf HANDLE_FICHIER FORMAT, LISTE
printf FORMAT, LISTE
```

Cette fonction affiche une chaîne formatée sur *HANDLE_FICHIER* ou, s'il est omis, sur le handle de fichier de sortie courant, initialement STDOUT. Le premier élément de *LISTE* doit être une chaîne qui indique comme formater les éléments suivants. La fonction est similaire à *printf* (3) et *fprintf* (3). La fonction est exactement équivalente à :

```
print HANDLE_FICHIER sprintf FORMAT, LISTE
```

sauf que `$\` (`$OUTPUT_RECORD_SEPARATOR`) n'est pas ajouté à la fin. Si `use locale` est actif, le caractère utilisé comme séparateur de la partie décimale dans les nombres à virgule flottante est affecté par la locale `LC_NUMERIC`.

Une exception n'est levée que si un type de référence invalide est employé pour l'argu-

9. STDOUT n'est donc pas vraiment le handle de fichier par défaut pour `print`. Il est en fait le handle de fichier *par défaut* du défaut.

ment *HANDLE_FICHIER*. Les formats inconnus sont sautés sans modification. Ces deux situations déclenchent des avertissements si ceux-ci sont activés.

Voir les fonctions `print` et `sprintf` ailleurs dans ce chapitre. La description de `sprintf` inclut la liste des spécifications des formats. Nous l'aurions volontiers reproduite ici, mais ce livre est d'ores et déjà un désastre écologique.

Si vous omettez à la fois le *FORMAT* et la *LISTE*, `$_` est utilisé — mais dans ce cas, vous auriez dû utiliser `print`. Ne tombez pas dans le piège d'utiliser un `printf` lorsqu'un simple `print` aurait très bien fait l'affaire. La fonction `print` est bien plus efficace et bien moins sujette à erreurs.

prototype


```
prototype FONCTION
```

Renvoie le prototype d'une fonction dans une chaîne (ou `undef` si la fonction n'a pas de prototype). *FONCTION* est une référence vers, ou le nom de, la fonction pour laquelle vous désirez trouver le prototype.

Si *FONCTION* est une chaîne commençant par `CORE::`, la suite est prise en tant que nom pour les fonctions internes de Perl, et une exception est levée s'il n'existe pas de telle fonction interne. Si la fonction interne n'est pas *surchargeable* (comme `qw//`) ou si ses arguments ne peuvent pas être exprimés par un prototype (comme `system`), la fonction renvoie `undef` car la fonction interne ne se comporte pas véritablement comme une fonction Perl. Sinon, la chaîne décrivant le prototype équivalent est renvoyée.

push

```
push TABLEAU
push
```

Cette fonction traite *TABLEAU* comme une pile et empile les valeurs de *LISTE* à la fin du tableau. La taille de *TABLEAU* est allongée de la longueur de *LISTE*. La fonction renvoie cette nouvelle taille. La fonction `push` entraîne les mêmes effets que :

```
foreach $valeur (fonction_liste()) {
    $tableau[++$#tableau] = $valeur;
}
```

ou que :

```
splice @tableau, @tableau, 0, fonction_liste();
```

mais est beaucoup plus performante (à la fois pour vous et pour votre ordinateur). Vous pouvez utilisez `push` associée avec `shift` pour construire une file ou un registre à décalage assez efficace :

```
for (;;) {
    push @tableau, shift @tableau;
    ...
}
```

Voir également `pop` et `unshift`.

q/CHAINE/

```
q/CHAINE/
qq/CHAINE/
qr/CHAINE/
qw/CHAINE/
qx/CHAINE/
```

Protections généralisées. Voir la section *Choisissez vos délimiteurs* au chapitre 2. Pour les annotations de statut pour `qx//`, voir `readpipe`. Pour les annotations de statut pour `qr//`, voir `m//`. Voir également *Garder le contrôle* au chapitre 5.

quotemeta


```
quotemeta EXPR
quotemeta
```

Cette fonction renvoie la valeur de *EXPR* avec tous les caractères non alphanumériques précédés d'un backslash (\). (C'est-à-dire que tous les caractères ne correspondant pas à `/[A-Za-z_0-9]/` seront précédés d'un backslash dans la chaîne renvoyée, sans tenir compte des valeurs des locales.) Il s'agit de la fonction interne implémentant l'échappement `\Q` dans les contextes d'interpolation (comprenant les chaînes entre guillemets, entre apostrophes inverses ou les motifs).

rand

```
rand EXPR
rand
```

Cette fonction renvoie un nombre à virgule flottante pseudo-aléatoire supérieur ou égal à 0 et inférieur à la valeur de *EXPR*. (*EXPR* doit être positif.) Si *EXPR* est omise, la fonction renvoie un nombre à virgule flottante entre 0 et 1 (comprenant 0, mais excluant 1). `rand` appelle automatiquement `srand` sauf si la fonction `srand` a déjà été appelée. Voir également `srand`.

Pour obtenir une valeur entière, comme pour un lancer de dé, combinez ceci avec `int`, comme dans :

```
$lancer = int(rand 6) +1;    # $lancer est maintenant un entier
                             # compris entre 1 et 6
```

Comme Perl utilise la fonction pseudo-aléatoire de votre propre bibliothèque C, comme *random*(3) ou *drand48*(3), la qualité de la distribution n'est pas garantie. Si vous avez besoin d'aléatoire plus fort, comme pour de la cryptographie, vous devriez plutôt consulter la documentation de *random*(4) (si votre système possède un périphérique */dev/random* ou */dev/urandom*), le module `Math::TrulyRandom` de CPAN, ou un bon bouquin sur la génération de nombres pseudo-aléatoires en informatique, comme le second volume de *Knuth*.[10]

10. Knuth, D.E. *The Art of Computer Programming, Seminumerial Algorithms*, vol. 2, 3rd ed. (Reading, Mss.: Addison-Wesley, 1997). ISBN 0-201-89684-2.

read

$! T ARG RO

```
read HANDLE_FICHIER, SCALAIRE, LONGUEUR, DECALAGE
read HANDLE_FICHIER, SCALAIRE, LONGUEUR
```

Cette fonction essaye de lire *LONGUEUR* octets de données dans la variable *SCALAIRE* depuis le *HANDLE_FICHIER* spécifié. La fonction renvoie le nombre d'octets réellement lus ou `0` en fin de fichier. `undef` est renvoyée en cas d'erreur. *SCALAIRE* est ajusté à la taille réellement lue. Le *DECALAGE*, s'il est spécifié, indique l'endroit de la variable à partie duquel commencer à écrire des octets, vous pouvez ainsi lire au milieu d'une chaîne.

Pour copier des données du handle de fichier `DEPUIS` dans le handle `VERS`, vous pouvez écrire :

```
while (read DEPUIS, $tampon, 16384) {
    print VERS $tampon;
}
```

L'inverse d'un `read` est simplement `print`, qui connaît déjà la longueur de la chaîne à écrire, et qui peut écrire une chaîne de n'importe quelle longueur. Ne faites pas l'erreur d'employer `write`, qui est uniquement utilisé dans les formats.

La fonction `read` de Perl est en fait implémentée par la fonction d'entrée/sortie standard *fread*(3), ce qui fait que le véritable appel système *read*(2) peut lire plus que *LONGUEUR* octets pour remplir le tampon d'entrée et que *fread*(3) peut faire plus d'un appel système à *read*(2) pour remplir ce tampon. Vous pouvez spécifier l'appel système réel par `sysread` pour mieux contrôler les entrées/sorties. Les appels à `read` et à `sysread` ne doivent pas être mélangés à moins d'être adepte des sciences occultes (ou des soirées cuir et chaînes). Quelle que soit celle que vous employez, faites attention : lorsque vous lisez depuis un fichier contenant un encodage Unicode ou n'importe quel encodage sur plusieurs octets, les limites du tampon peuvent tomber au milieu d'un caractère.

readdir

$! T ARG U

```
readdir HANDLE_REPERTOIRE
```

Cette fonction lit les entrées de répertoire (qui ne sont que de simples noms de fichiers) depuis un handle de répertoire ouvert par `opendir`. Dans un contexte scalaire, cette fonction renvoie l'entrée de répertoire suivante si elle existe, `undef` sinon. Dans un contexte de liste, elle renvoie le reste des entrées du répertoire, ce qui donnera bien sûr une liste nulle s'il n'y en a plus. Par exemple :

```
opendir(CE_REP, "." or die "sérieusement endommagé : $!";
@tous_les_fichiers = readdir CE_REP;
closedir CE_REP;
print "@tous_les_fichiers\n";
```

Ceci affiche tous les fichiers du répertoire courant sur une seule ligne. Pour éviter les entrées `.` et `..` , employez l'une de ces incantations (quelle que soit celle à laquelle vous pensez, elle sera moins illisible) :

```
@tous_les_fichiers = grep { $_ ne '.' and $_ ne '..' } readdir CE_REP;
@tous_les_fichiers = grep { not /^[.][.]?\z/ } readdir CE_REP;
@tous_les_fichiers = grep { not /^\.{1,2}\z/ } readdir CE_REP;
@tous_les_fichiers = grep !/^\.\.?\z/, readdir CE_REP;
```

Et pour éviter tous les fichiers .*, écrivez ceci :

```
@tous_les_fichiers = grep !/^\./, readdir CE_REP;
```

Pour ne lister que les fichiers texte, écrivez :

```
@fichiers_texte = grep -T, readdir CE_REP;
```

Mais attention à ce dernier exemple, car le répertoire doit être recollé au résultat de readdir si ce n'est pas le répertoire courant — comme ceci :

```
opendir(CE_REP, $chemin); or die "opendir $chemin impossible : $!" ;
@fichiers_points = grep { /^\./ && -f } map { "$chemin/$_" }
readdir(CE_REP);
closedir CE_REP;
```

readline

$! T X ARG

```
readline HANDLE_FICHIER
```

Il s'agit de la fonction interne implémentant l'opérateur <HANDLE_FICHIER>, mais vous pouvez l'utiliser directement. La fonction lit l'enregistrement suivant depuis *HANDLE_FICHIER*, qui peut être un nom de handle de fichier ou une expression de handle de fichier indirect renvoyant soit le nom d'un véritable handle de fichier, soit une référence à quoique ce soit ressemblant à une objet handle de fichier, comme un typeglob. (Les versions de Perl antérieures à la 5.6 acceptent seulement un typeglob.) Dans un contexte scalaire, chaque appel lit et renvoie le prochain enregistrement jusqu'à ce que l'on atteigne la fin de fichier, avec laquelle l'appel suivant renvoie undef. Dans un contexte de liste, readline lit tous les enregistrements jusqu'à atteindre la fin de fichier et renvoie alors une liste d'enregistrements. Par « enregistrements », il faut normalement entendre une ligne de texte, mais si la variable spéciale $/ ($INPUT_RECORD_SEPARATOR) est différente de sa valeur par défaut, l'opérateur « avalera » le texte différemment. De même, certaines disciplines en entrée, comme :para (mode paragraphe) renverront des enregistrements par morceaux, autres que des lignes. Le positionnement de la discipline :slurp (ou $/ = undef) entraînera l'avalement du fichier entier.

Lorsqu'on avale des fichiers tout rond dans un contexte scalaire, s'il vous arrive de gober un fichier vide, readline renvoie "" la première fois et undef toutes les fois suivantes. Si l'on avale tout rond en partant du handle de fichier magique ARGV, chaque fichier renvoie une seule bouchée (encore une fois, les fichiers vides renvoient ""), suivi par un seul undef lorsque tous les fichiers on été digérés.

De plus amples détails sur l'opérateur <*HANDLE_FICHIER*> sont donnés dans la section *Opérateurs d'entrée* au chapitre 2.

```
$ligne = <STDIN>;
$ligne = readline(STDIN);          # idem
$ligne = readline(*STDIN);         # idem
$ligne = readline(\*STDIN);        # idem

open my $hf, "<&=STDIN" or die;
bless $hf => 'UneVieilleClasse';
$ligne = readline($hf);            # idem
```

readlink

$_ $! T X U

```
readlink EXPR
readlink
```

Cette fonction renvoie le nom d'un fichier pointé par un lien symbolique. *EXPR* doit donner le nom d'un fichier dont la dernière composante est un lien symbolique. S'il ne s'agit pas d'un lien symbolique, ou si les liens symboliques ne sont pas implémentés dans le système de fichier, ou encore si une erreur se produit, la fonction renvoie la `undef` et vous devrez vérifier le code d'erreur dans `$!`.

Prenez garde car le lien symbolique renvoyé peut être relatif à l'endroit spécifié. Par exemple, vous pouvez écrire :

```
readlink "/usr/local/src/exprimez/vous.h"
```

et `readlink` pourrait renvoyer :

```
../exprimez.1.23/includes/vous.h
```

ce qui n'est pas franchement utile à moins que le répertoire courant ne soit justement `/usr/local/src/exprimez`.

readpipe

$! $? T X T X U

```
readpipe scalar EXPR
readpipe LIST  (à l'étude)
```

Il s'agit de la fonction interne implémentant la construction protégée `qx//` (connue également en tant qu'opérateur apostrophes inverses). Elle est parfois commode lorsque vous avez besoin de spécifier votre *EXPR* d'une manière qui ne serait pas pratique avec les apostrophes. Prenez garde car nous pourrions modifier à l'avenir cette interface pour supporter un argument *LISTE* qui la ferait plus ressembler à la fonction `exec`, n'imaginez donc pas qu'elle continuera à fournir un contexte scalaire pour *EXPR*. Indiquez vous-même `scalar` ou utilisez la forme avec *LISTE*. Qui sait, peut-être cela fonctionnera-t-il au moment où vous lirez ceci.

recv

$! T X ARG X RO X U

```
recv SOCKET, SCALAIRE, LONGUEUR, FLAGS
```

Cette fonction reçoit un message sur une socket. Elle s'attend à recevoir *LONGUEUR* octets de données dans la variable *SCALAIRE* depuis le handle de fichier *SOCKET*. La fonction renvoie l'adresse de l'expéditeur, ou `undef` s'il y a une erreur. *SCALAIRE* est ajusté à la longueur effectivement lue. Les *FLAGS* de la fonction sont les mêmes que ceux de *recv*(2). Voir la section *Sockets* au chapitre 16.

redo

$@

```
redo ETIQUETTE
redo
```

L'opérateur `redo` recommence le bloc d'une boucle sans réévaluer la condition. Le bloc `continue`, s'il existe, n'est pas exécuté. Si l'*ETIQUETTE* est omise, l'opérateur se réfère à la

boucle la plus interne l'encadrant. Cet opérateur est normalement utilisé par les programmes qui souhaitent dissimuler à eux-même ce qui vient d'être entré :

```
# Une boucle qui joint les lignes coupées par un antislash.
while (<STDIN>) {
    if (s/\\\n$// && defined($ligne_suivante = <STDIN>)) {
        $_ .= $ligne_suivante;
        redo LIGNE;
    }
    print; # ou ce que vous voulez
}
```

On ne peut pas utiliser `redo` pour sortir d'un bloc qui renvoie une valeur comme `eval {}`, `sub {}` ou `do {}`, ni d'une opération `grep` ou `map`. Si les avertissements sont actifs, Perl vous avertira si vous faites un `redo` dans une boucle qui ne se trouve pas dans votre portée lexicale courante.

Un bloc en soi-même est sémantiquement identique à une boucle qui ne s'exécute qu'une seule fois. Ainsi `redo` à l'intérieur un tel bloc le changera en une construction de boucle. Voir la section *Contrôle de boucle* au chapitre 4.

ref $_

```
ref EXPR
ref
```

L'opérateur `ref` renvoie une valeur vraie si *EXPR* est une référence, sinon faux. La valeur renvoyée dépend du type d'objet auquel il est fait référence. Les types intégrés comprennent :

```
SCALAR
ARRAY
HASH
CODE
GLOB
REF
LVALUE
IO::Handle
```

Si l'objet référencé a été consacré (avec `bless`) dans un paquetage, c'est le nom du paquetage qui est renvoyé. Vous pouvez voir `ref` comme une sorte d'opérateur « typeof ».

```
if (ref($r) eq "HASH") {
    print "r est une référence à un hachage.\n";
} elsif (ref($r) eq "Bosse") {     # Vilain, voir ci-dessous.
    print "r est une référence à un objet Bosse.\n";
} elsif (not ref ($r)) {
    print "r n'est pas du tout une référence.\n";
}
```

On considère que tester l'égalité de votre classe d'objet avec une classe particulière est une erreur de style OO (orienté objet), puisqu'une classe dérivée aura un nom différent mais devra permettre l'accès aux méthodes de la classe de base. Il vaut mieux utiliser la méthode `isa` de la classe `UNIVERSAL` comme ceci :

```
if ($r->isa("Bosse")) {
    print "r est une référence à un objet Bosse, ou à une sous-classe.\n";
}
```

Il vaut mieux normalement ne rien tester du tout, puisque le mécanisme OO n'enverra pas l'objet à votre méthode sauf s'il pense qu'elle est la plus appropriée. Voir les chapitres 8 et 12 pour plus de détails. Voir aussi la fonction `reftype` dans le pragma `use attributes` au chapitre 31.

rename

```
rename ANCIEN_NOM, NOUVEAU_NOM
```

Cette fonction change le nom d'un fichier. Elle renvoie vrai en cas de succès et faux sinon. Elle ne fonctionnera pas (normalement) à travers les frontières des systèmes de fichiers, bien que sur un système Unix, la commande *mv* puisse parfois être utilisée pour compenser cela. Si un fichier nommé *NOUVEAU_NOM* existe déjà, il sera détruit. Les systèmes non-Unix peuvent comporter des restrictions supplémentaires.

Voire le module standard `File::Copy` pour renommer les fichiers d'un système de fichiers à l'autre.

require

```
require VERSION
require EXPR
require
```

Cette fonction exprime une sorte de dépendance envers son argument.

Si l'argument est une chaîne, `require` charge et exécute le code Perl trouvé dans le fichier séparé dont le nom est donné par la chaîne. Ceci est similaire à l'exécution d'un `do`, mis à part que `require` vérifie si le fichier de bibliothèque a déjà été chargé et lève une exception si un quelconque problème est rencontré. (On peut donc l'utiliser pour exprimer des dépendances de fichiers sans se soucier d'une duplication de la compilation.) Comme ses cousines `do` et `use`, `require` sait comment rechercher le chemin d'inclusion, stocké dans le tableau `@INC` et sait également mettre à jour `%INC` en cas de succès. Voir le chapitre 28.

Le fichier doit renvoyer vrai comme dernière valeur pour indiquer l'exécution réussie du code d'initialisation, il est donc habituel de terminer ce genre de fichier par un `1;`, à moins que vous ne soyez sûrs qu'il renverra vrai de toute façon.

Si l'argument de `require` est un numéro de version de la forme 5.6.2, `require` exige que la version de Perl entrain d'être exécuté soit égale ou supérieure à ce numéro de version. (Perl accepte également un nombre à virgule flottante comme 5.005_03 pour une compatibilité avec les anciennes versions de Perl, mais cette forme est maintenant fortement déconseillée car les foules venant d'autres cultures ne la comprenne pas.) Ainsi, un script qui requiert Perl version 5.6 peut commencer par :

```
require 5.6.0           # ou require v5.6.0
```

et les versions plus anciennes de Perl échoueront. Cependant, comme tous les `require`, la vérification est réalisée lors de l'exécution. Vous pouvez préférer écrire `use 5.6.0`

pour un test à la compilation. Voir aussi `$PERL_VERSION` au chapitre 28.

Si l'argument de `require` est le nom dénudé d'un paquetage (voir `package`), `require` suppose que le suffixe est automatiquement `.pm`, facilitant ainsi le chargement de modules standard. Cela ressemble à `use`, mis à part le fait que cela se produit à l'exécution et non à la compilation et que la méthode d'`import` n'est pas appelée. Par exemple, pour charger *Socket.pm* sans introduire aucun symbole dans le package courant, écrivez ceci :

```
require Socket;         # au lieu de "use Socket;"
```

Mais on peut toutefois obtenir le même effet avec ce qui suit, qui procure l'avantage de donner des avertissements à la compilation si *Socket.pm* ne peut être trouvé :

```
use Socket ();
```

L'utilisation de `require` sur un nom dénudé de paquetage remplace également tout `::` dans le nom d'un paquetage par le séparateur de répertoires de votre système, généralement `/`. En d'autres termes, si vous essayez ceci :

```
require Machin::Truc;   # un formidable nom dénudé
```

la fonction `require` cherchera le fichier *Machin/Truc.pm* dans les répertoires spécifiés par le tableau `@INC`. Mais si vous essayez cela :

```
$classe = 'Machin::Truc';
require $classe;        # $classe n'est pas un nom dénudé
```

ou encore cela :

```
require "Machin::Truc"; # un littéral entre guillemets n'est pas un
                        #nom dénudé
```

la fonction `require` recherchera le fichier `Machin::Truc` dans le tableau `@INC` et se plaindra de ne pas y trouver `Machin::Truc`. Si cela arrive, vous pouvez faire ceci :

```
eval "require $classe";
```

Voir également les commandes `do` *`FICHIER`*, `use`, le pragma `use lib` et le module standard `FindBin`.

reset

```
reset EXPR
reset
```

On use (ou on abuse) de cette fonction généralement au début d'une boucle ou dans un bloc `continue` à la fin d'une boucle, pour nettoyer les variables globales ou réinitialiser les recherches `??` pour qu'elles puissent à nouveau fonctionner. L'*`EXPR`* est interprétée comme une liste de caractères uniques (les tirets sont autorisés pour les intervalles). Toutes les variables scalaires et tous les tableaux et hachages commençant par l'une de ces lettres sont remises dans leur état d'origine. Si l'*`EXPR`* est omise, les recherches de correspondance unique (`?`*`MOTIF`*`?`) sont réinitialisées pour correspondre à nouveau. La fonction réinitialise les variables ou les recherches uniquement pour le package courant. Elle renvoie toujours vrai.

Pour réinitialiser toutes les variables en « `X` » , écrivez ceci :

```
reset 'X';
```

Pour réinitialiser toutes les variables en minuscules, écrivez cela :

```
reset 'a-z';
```

Enfin, pour ne réinitialiser que les recherches en `??`, écrivez :

```
reset;
```

La réinitialisation de « `A-Z` » dans le package principal `main` n'est pas recommandée car les tableaux et hachages `ARGV`, `INC`, `ENV` et `SIG` seront probablement balayés.

Les variables lexicales (créées par `my`) ne sont pas affectées. L'emploi de `reset` est vaguement déprécié car il efface un peu trop facilement des espaces de noms entiers et parce que l'opérateur `??` est lui-même vaguement déprécié.

Voir également la fonction `delete_package()` du module standard `Symbol` et la discussion entière sur les compartiments sécurisés dans la section *Compartiments sécurisés* au chapitre 23.

return

$@

```
return EXPR
return
```

Cet opérateur entraîne le retour immédiat de la sous-routine courante (ou de l'`eval` ou du `do` *FICHIER*) avec la valeur spécifiée. Toute tentative d'utiliser `return` en dehors de ces trois endroits lève une exception. Remarquez également qu'un `eval` ne peut pas faire un `return` pour le compte de la sous-routine qui a appelé l'`eval`.

EXPR peut être évaluée dans un contexte de liste, scalaire ou vide, selon la manière dont la valeur de retour sera utilisé, ce qui peut varier d'une exécution à l'autre. C'est-à-dire que l'expression fournie sera évaluée dans le contexte de l'invocation du sous-programme. Si le sous-programme a été appelé dans un contexte scalaire, *EXPR* est également évaluée dans un contexte scalaire. Si le sous-programme a été appelé dans un contexte de liste, *EXPR* est également évaluée dans un contexte de liste. Un `return` sans argument renvoie la valeur scalaire `undef` dans un contexte scalaire et une liste vide `()` dans un contexte de liste, et (naturellement) rien du tout dans un contexte vide. Le contexte de l'appel au sous-programme peut être déterminé de l'intérieur du sous-programme par la fonction (mal nommée) `wantarray`.

reverse

```
reverse LISTE
```

Dans un contexte de liste, cette fonction renvoie une valeur liste composée des éléments de *LISTE* dans l'ordre inverse. La fonction peut être utilisée pour créer des séquences décroissantes :

```
for (reverse 1 .. 10) { ... }
```

Comme les hachages se transforment en listes quand ils sont passés en tant que *LISTE*, `reverse` peut également être employée pour inverser un hachage, à supposer que les valeurs soient uniques :

```
%machintruc = reverse %trucmachin;
```

Dans un contexte scalaire, la fonction concatène tous les caractères de *LISTE* et renvoie alors l'inverse, caractère par caractère, de la chaîne résultante.

Petite astuce : l'inversion d'une liste triée par une fonction définie par l'utilisateur peut parfois être plus facilement accomplie en triant d'abord la liste dans le sens opposé.

rewinddir

$! X ARG X U

```
rewinddir HANDLE_REPERTOIRE
```

Cette fonction se positionne au début du répertoire pour la routine `readdir` sur *HANDLE_REPERTOIRE*. La fonction peut ne pas être disponible sur toutes les machines supportant `readdir` — `rewindir` meurt si elle n'est pas implémentée. Elle renvoie vrai en cas de succès, faux sinon.

rindex

```
rindex CHAINE, SOUS_CHAINE, POSITION
rindex CHAINE, SOUS_CHAINE
```

Cette fonction agit exactement comme `index` mis à part le fait qu'elle renvoie la position de la *dernière* occurrence de *SOUS_CHAINE* dans *CHAINE* (un `index` inverse). La fonction renvoie `$[-1` si aucune *SOUS_CHAINE* n'a été trouvée. Comme `$[` est le plus souvent à `0` de nos jours, la fonction renvoie presque toujours `-1`. *POSITION*, si elle est spécifiée, est la position la plus à droite qui puisse être renvoyée. Pour scruter une chaîne à l'envers, écrivez :

```
$pos = length $chaine;
while (($pos = rindex $chaine, $a_chercher, $pos) >= 0) {
    print "Trouvé en $pos\n";
    $pos--;
}
```

rmdir

$_ $! X T

```
rmdir REPERTOIRE
rmdir
```

Cette fonction supprime le répertoire spécifié par *REPERTOIRE* si ce répertoire est vide. Si la fonction réussit, elle renvoie vrai ; sinon, elle renvoie faux. Voir également le module `File::Path` si vous voulez d'abord supprimer le contenu du répertoire et si vous n'envisagez pas d'appeler le shell avec `rm -r`, pour une raison ou une autre. (Si par exemple vous ne disposez pas de shell ou de commande *rm*, parce que vous n'avez pas encore PPT[11].)

11. *N.d.T.* : *Perl Power Tools: The Unix Reconstruction Project*, un projet dont le but avoué est : « de manière assez simple, de réimplémenter le jeu de commandes classiques Unix en Perl pur, et de nous amuser autant que possible en faisant cela ».

s///

T X RO X T

```
s///
```

L'opérateur de substitution. Voir la section *Opérateurs de correspondance de motif* au chapitre 5.

scalar

```
scalar EXPR
```

Cette pseudo-fonction peut être utilisée dans une *LISTE* pour forcer *EXPR* à être évaluée dans un contexte scalaire quand l'évaluation dans un contexte de tableau produirait un résultat différent. Par exemple :

```
my ($var_suivante) = scalar <STDIN>;
```

empêche `<STDIN>` de lire toutes les lignes depuis l'entrée standard avant d'effectuer l'affectation, puisqu'une affectation à une liste (même à une `my` liste) donne un contexte de liste. (Sans l'utilisation de `scalar` dans cet exemple, la première ligne de `<STDIN>` serait toujours assignée à `$var_suivante`, mais les lignes suivantes seraient lues et poubellisées, puisque la liste à laquelle nous sommes en train d'assigner ne peut recevoir qu'une seule valeur scalaire.)

Bien sûr, il serait plus simple et moins encombrant d'enlever les parenthèses, transformant ainsi le contexte de liste en un contexte scalaire :

```
my $var_suivante = <STDIN>;
```

Puisqu'une fonction `print` est un opérateur de *LISTE*, il vous faudrait écrire :

```
print "La longueur est ", scalar(@TABLEAU), "\n";
```

pour afficher la longueur de `@TABLEAU`.

Il n'existe aucune fonction « `list` » correspondant à `scalar`, puisqu'en pratique, on n'a jamais besoin de forcer l'évaluation dans un contexte de liste. Car toute opération demandant une *LISTE* fournit déjà un contexte de liste à ses arguments de liste, et ceci gratuitement.

Comme `scalar` est un opérateur unaire, si vous utilisez accidentellement une liste entre parenthèses pour l'*EXPR*, cette dernière se comportera comme une expression scalaire avec des virgules, en évaluant tous les éléments sauf le dernier dans un contexte vide et renvoyant l'élément final évalué dans un contexte scalaire. C'est rarement ce que vous voulez. La seule instruction suivante :

```
print uc(scalar(&machin,$truc)),$bidule;
```

est l'équivalent (im)moral de ces deux instructions :

```
&machin; print(uc($truc),$bidule);
```

Voir le chapitre 2 pour plus de détails sur le séparateur virgule. Voir la section *Prototypes* au chapitre 6 pour plus d'informations sur les opérateurs unaires.

seek

$! X ARG

```
seek HANDLE_FICHIER, DECALAGE, DEPART
```

Cette fonction positionne le pointeur de fichier pour *HANDLE_FICHIER*, exactement comme l'appel d'entrée/sortie standard *fseek*(3). La première position dans un fichier correspond à un décalage de 0, et non de 1. Les décalages se réfèrent à des positions en octets, et non des numéros de ligne. En général, comme les longueurs de ligne varient, il n'est pas possible d'accéder à un numéro de ligne précis sans lire tout le fichier jusqu'à cet endroit, à moins que toutes vos lignes ne soient d'une longueur fixe ou que vous ayez construit un index convertissant les numéros de lignes en décalages d'octets. (Les mêmes restrictions s'appliquent aux positions des caractères dans un fichier avec des encodages de caractères de longueur variable : le système d'exploitation ne sait pas ce que sont les caractères, il ne connaît que des octets.)

HANDLE_FICHIER peut être une expression dont la valeur donne un véritable nom de handle de fichier ou une référence vers quoique ce soit ressemblant à un objet handle de fichier. La fonction renvoie vrai en cas de succès, faux sinon. Pour des raisons pratiques, la fonction peut calculer pour vous des décalages depuis diverses positions dans le fichier. La valeur de *DEPART* indique la position à partir de laquelle votre *DECALAGE* commencera : `0`, pour le début du fichier ; `1`, pour la position courante dans le fichier ; ou `2`, pour la fin du fichier. Le *DECALAGE* peut être négatif pour un *DEPART* de `1` ou de `2`. Si vous désirez employer des valeurs symboliques pour *DEPART*, vous pouvez utiliser `SEEK_SET`, `SEEK_CUR` et `SEEK_END` importée depuis les modules `IO::Seekable` ou `POSIX`, ou depuis la version 5.6 de Perl, depuis le module `Fcntl`.

Si vous voulez connaître la position dans le fichier pour `sysread` ou `syswrite`, n'employez pas `seek` ; les conséquences des tampons des entrées/sorties standards rendent la position système dans un fichier imprévisible et non portable. Utilisez `sysseek` à la place.

À cause des règles rigoureuses du C ANSI, vous devez sur certains systèmes faire un `seek` à chaque fois que vous basculez entre une lecture et une écriture. Parmi d'autres effets, ceci peut avoir comme conséquence d'appeler la fonction *clearerr*(3) de la bibliothèque standard d'entrée/sortie. Un *DEPART* de `1` (`SEEK_CUR`) avec un *DECALAGE* de `0` est utile pour ne pas bouger de la position courante dans le fichier :

```
seek(TEST,0,1);
```

Cette fonction s'avère intéressante notamment pour suivre la croissance d'un fichier, comme ceci :

```
for (;;) {
    while (<JOURNAL>) {
        gnoquer($_)           # Traiter la ligne courante
    }
    sleep 15;
    seek LOG,0,1;             # RAZ erreur de fin de fichier
}
```

Le `seek` final réinitialise l'erreur de fin de fichier sans bouger le pointeur. Si cela ne fonctionne pas, selon l'implémentation de la bibliothèque C d'entrées/sorties standards, alors il vous faut plutôt ce genre de code :

```
for (;;) {
    for ($pos_cour = tell FILE; <FILE>; $pos_cour = tell FILE) {
        gnoquer($_)           # Traiter la ligne courante
    }
    sleep $un_moment;
```

```
    seek FILE, $pos_cour, 0;   # RAZ erreur de fin de fichier
}
```

Des stratégies similaires peuvent être employées pour conserver les adresses de `seek` pour chaque ligne dans tableau.

seekdir

$! X ARG X U

```
seekdir HANDLE_REPERTOIRE, POSITION
```

Cette fonction fixe la position courante pour le prochain appel à `readdir` sur *`HANDLE_REPERTOIRE`*. *`POSITION`* doit être une valeur renvoyée par `telldir`. Cette fonction comporte les mêmes dangers relatifs à un possible compactage du répertoire que la routine de la bibliothèque système correspondante. La fonction peut ne pas être implémentée partout où l'est `readdir`. Elle ne l'est certainement pas si `readdir` est absente.

select (handle de fichier de sortie)

X ARG

```
select HANDLE_FICHIER
select
```

Pour des raisons historiques, il existe deux opérateurs `select` totalement distincts l'un de l'autre. La section suivante décrit l'autre. Cette version de l'opérateur `select` renvoie le handle de fichier de sortie actuellement sélectionné et, si *`HANDLE_FICHIER`* est fourni, fixe le handle par défaut courant pour la sortie. Cela a deux conséquences : d'abord, un `write` ou un `print` sans handle de fichier s'adressera à ce handle par défaut. Ensuite, les variables spéciales relatives à la sortie se réfèreront à ce handle de fichier de sortie. Par exemple, s'il faut fixer un format d'en-tête identique pour plus d'un handle de fichier de sortie, il vous faudrait faire ceci :

```
select RAPPORT1;
$^ = 'mon_en_tete';
select RAPPORT2;
$^ = 'mon_en_tete';
```

Mais remarquez que cela laisse `RAPPORT2` comme handle de fichier actuellement sélectionné. Ce qui est un comportement antisocial condamnable puisque cela peut réellement démolir les routines `print` ou `write` de quelqu'un d'autre. Les routines de bibliothèque bien écrites laissent le handle de fichier sélectionné courant dans l'état où elles l'ont trouvé en entrant. À cet effet, *`HANDLE_FICHIER`* peut être une expression dont la valeur donne le nom du véritable handle de fichier. Il vous est donc possible de sauvegarder et de restaurer le handle de fichier actuellement sélectionné :

```
my $ancien_hf = select STDERR; $| = 1; select $ancien_hf;
```

ou de manière idiomatique (mais nettement plus absconse) :

```
select((select(STDERR), $| = 1)[0])
```

Cet exemple fonctionne en construisant une liste composée de la valeur renvoyée par `select(STDERR)` (qui sélectionne `STDERR` par effet de bord) et `$| = 1` (qui est toujours 1), mais vide le tampon de `STDERR`, maintenant sélectionné, par effet de bord. Le premier élément de cette liste (le handle de fichier précédemment sélectionné) est maintenant utilisé comme un argument vers le `select` extérieur. Étrange, n'est-ce pas ? C'est ce

que vous obtenez quand vous connaissez juste assez de Lisp pour être dangereux.

Vous pouvez également utiliser le module standard `SelectSaver` pour restaurer automatiquement le `select` précédent à la sortie de la portée.

Maintenant que nous avons expliqué tout cela, nous devons souligner que vous n'aurez que rarement besoin d'utiliser cette forme de `select`, car la plupart des variables spéciales auxquelles vous pouvez vouloir accéder possèdent des méthodes d'enrobage orientées objet. Donc, au lieu d'accéder directement à `$|`, vous pouvez écrire :

```
use IO::Handle;        # Malheureusement, ce *n'est pas* un petit module
STDOUT->autoflush(1);
```

Et l'exemple précédent de format peut être codé comme suit :

```
use IO::Handle;
RAPPORT1->format_top_name("mon_en_tete");
RAPPORT2->format_top_name("mon_en_tete");
```

select (descripteurs de fichiers prêts)

```
select BITS_LEC, BITS_ECR, BITS_ERR, MINUTEUR
```

La forme à quatre arguments de `select` est totalement indépendante de l'opérateur `select` précédemment décrit. Celui-ci sert à découvrir lesquels (s'il en existe) de vos descripteurs de fichiers sont prêts à effectuer une entrée ou une sortie, ou à informer d'une condition d'exception. (Ce qui vous évite une scrutation.) Elle appelle l'appel système *select*(2) avec les masques de bits que vous avez spécifiés, qui peuvent être construits grâce à `fileno` et `vec`, comme ceci :

```
$lec_entree = $ecr_entree = $err_entree = '';
vec($lec_entree, fileno(STDIN), 1) = 1;
vec($ecr_entree, fileno(STDOUT), 1) = 1;
$err_entree = $lec_entree | $ecr_entree;
```

Si vous voulez faire un `select` sur plusieurs handles de fichier, il peut être préférable d'écrire une routine :

```
sub bits_hf {
    my @liste_hf = @_;
    my $bits;
    for (@liste_hf) {
        vec($bits, fileno($_), 1) = 1;
    }
    return $bits;
}
$lec_entree = bits_hf(qw(STDIN TTY MA_SOCK));
```

Si vous désirez utiliser les mêmes masques de bits de façon répétitive (et c'est plus efficace), la manière la plus courante est :

```
($nb_trouves, $temps_restant) =
    select( $lec_sortie=$lec_entree, $ecr_sortie=$ecr_entree,
            $err_sortie=$err_entree, $minuteur);
```

Ou pour bloquer jusqu'à ce qu'un descripteur de fichiers soit prêt :

```
$nb_trouves = select( $lec_sortie=$lec_entree, $ecr_sortie=$ecr_entree,
            $err_sortie=$err_entree, undef);
```

Comme vous pouvez le constater, l'appel de `select` dans un contexte scalaire renvoie seulement `$nb_trouves`, soit le nombre de descripteurs de fichiers prêts trouvés.

L'astuce `$ecr_sortie=$ecr_entree` fonctionne car la valeur d'une affectation est le côté gauche, ce qui fait que `$ecr_sortie` est d'abord modifié par l'affectation, puis par le `select`, alors que `$ecr_entree` reste inchangé.

Chaque argument peut également valoir `undef`. Le *`MINUTEUR`*, s'il est spécifié, est en secondes[12], et peut éventuellement comporter une virgule. (Un délai de `0` effectue une scrutation.) Toutes les implémentations ne sont pas capables de renvoyer le `$temps_restant`. Dans ce cas, elles renverront toujours un `$temps_restant` égal au `$minuteur` fourni.

Le module standard `IO::Select` fournit une interface plus amicale à `select`, surtout parce qu'il effectue pour vous toutes les opérations sur les masques de bits.

Un des emplois possibles de `select` consiste à endormir un processus avec une résolution plus fine que ce qu'admet `sleep`, ceci en spécifiant `undef` pour tous les masques de bits. Ainsi pour vous endormir pendant (au moins) 4,75 secondes, utilisez :

```
select undef, undef, undef, 4.75;
```

(Le triple `undef` peut ne pas fonctionner sur certains systèmes non-UNIX, et il vous faut alors mettre fallacieusement au moins un masque de bits à un descripteur de fichier qui ne sera jamais prêt.)

Il est dangereux de mélanger des entrées/sorties bufferisées (comme `read` ou `<HANDLE>`) avec `select`, à l'exception de ce que permet POSIX, et même dans ce cas, uniquement sur les systèmes réellement POSIX. Employez plutôt `sysread`.

semctl

$! [x 0]

```
semctl CLEF, NUM_SEM, CMD, ARG
```

Cette fonction invoque l'appel système *semctl*(2) des IPC System V. Vous devrez probablement d'abord faire `use IPC::SysV` pour obtenir les définitions de constantes correctes. Si *`CMD`* vaut `IPC_STAT` ou `GETALL`, *`ARG`* doit alors être une variable qui contiendra la structure `semid_ds` ou le tableau des valeurs de sémaphore qui sont renvoyés. Comme pour `ioctl` et `fcntl`, les valeurs renvoyées sont `undef` en cas d'erreur, « `0 but true` » (*N.d.T.* : 0 mais vrai) pour zéro ou sinon la valeur réellement renvoyée.

Voir également le module `IPC::Semaphore`. Cette fonction n'est disponible que sur les systèmes supportant les IPC System V.

semget

$! [x 0]

```
semget CLEF, NB_SEMS, TAILLE, FLAGS
```

12. *N.d.T.* : les anglophones ont la chance de dire *timeout*, ce qui évite toute confusion dans les unités de temps !

Cette fonction invoque l'appel système *semget*(2) des IPC System V. Vous devrez probablement d'abord faire `use IPC::SysV` pour obtenir les définitions de constantes correctes. La fonction renvoie l'ID du sémaphore ou `undef` en cas d'erreur.

Voir également le module `IPC::Semaphore`. Cette fonction n'est disponible que sur les systèmes supportant les IPC System V.

semop

$! X U

```
semop CLEF, CHAINE_OP
```

Cette fonction invoque l'appel système *semop*(2) des IPC System V pour réaliser des opérations sur les sémaphores, comme signaler ou attendre la libération. Vous devrez probablement d'abord faire `use IPC::SysV` pour obtenir les définitions de constantes correctes.

CHAINE_OP doit être un tableau empaqueté de structures `semop`. Vous pouvez construire chaque structure `semop` en écrivant `pack("s*", $num_sem, $op_sem, $flags_sem)`. Le nombre d'opérations de sémaphores est donné par la longueur de *CHAINE_OP*. La fonction renvoie vrai si elle a réussi, faux en cas d'erreur.

Le code qui suit attend sur le sémaphore `$num_sem` de l'ensemble de sémaphores identifié par `$id_sem` :

```
$op_sem = pack "s*", $semnum, -1, 0;
semop $id_sem, $op_sem or die "Problème de sémaphore : $!\n";
```

Pour envoyer un signal au sémaphore, il vous suffit de remplacer `-1` par `1`.

Voir également le module `IPC::Semaphore`. Cette fonction n'est disponible que sur les systèmes supportant les IPC System V.

send

$! X ARG X U

```
send SOCKET, MSG, FLAGS, VERS
send SOCKET, MSG, FLAGS
```

Cette fonction envoie un message sur une socket. Elle prend les mêmes flags que l'appel système du même nom — voir *send*(2). Sur des sockets non connectées, vous devez spécifier une destination *VERS* laquelle envoyer les données, ce qui fait que le `send` de Perl fonctionne comme *sendto*(2). La fonction `send` renvoie le nombre d'octets envoyés, ou `undef` en cas d'erreur.

(Certains systèmes traitent improprement les sockets comme des objets différents de simples descripteurs de fichiers, ce qui vous oblige à toujours utiliser `send` et `recv` sur les sockets au lieu des opérateurs d'entrée/sortie standards.)

Une erreur qu'au moins l'un de nous fait fréquemment est de confondre le `send` de Perl avec le `send` de C en écrivant :

```
send SOCK, $tampon, $length $tampon # FAUX
```

Ceci échouera mystérieusement selon le rapport exigé par le système entre la longueur de la chaîne et les bits de *FLAGS*. Voir la section *Passage de message* au chapitre 16 pour plus d'exemples.

setpgrp

$! X T X U

```
setpgrp PID, GRP_P
```

Cette fonction modifie le groupe de processus courant (*GRP_P*) pour le *PID* spécifié (employez un *PID* égal à `0` pour le processus courant). L'invocation de `setpgrp` lèvera une exception sur une machine n'implémentant pas *setpgrp*(2). Attention : certains systèmes ignoreront les arguments fournis et feront toujours `setpgrp(0, $$)`}. Ce sont heureusement les arguments que l'on fournit généralement. Si les arguments sont omis, ils vaudront `0,0` par défaut. La version BSD 4.2 de `setpgrp` n'acceptait aucun argument, mais pour BSD 4.4, c'est un synonyme de la fonction `setpgid`. Pour une meilleure portabilité (du moins théorique), utilisez directement la fonction `setpgid` du module `POSIX`. Ou, si vous voulez changer votre script en démon, la fonction `POSIX::setsid()` peut aussi bien faire l'affaire. Remarquez que la version `POSIX` de `setpgrp` n'accepte pas d'arguments, seul `setpgrp(0,0)` est donc portable.

setpriority

$! X T X U

```
setpriority QUOI, QUI, PRIORITE
```

Cette fonction modifie la *PRIORITE* courante d'un processus, d'un groupe de processus ou d'un utilisateur, tel que spécifiés par *QUOI* et *QUI*. Voir *setpriority*(2). Son invocation lèvera une exception sur une machine où *setpriority*(2) n'est pas implémentée. Pour faire un *nice* sur votre processus de quatre unités vers le bas (la même chose que d'exécuter votre programme avec *nice*(1)), essayez :

```
setpriority 0, 0, getpriority(0, 0) + 4;
```

L'interprétation d'une priorité donnée peut varier d'un système d'exploitation à l'autre. Certaine priorités ne sont pas accessibles aux utilisateurs sans privilèges.

Voir également le module `BSD::Resource` de CPAN.

setsockopt

$! X ARG X U

```
setsockopt SOCKET, NIVEAU, NOM_OPTION, VALEUR_OPTION
```

Cette fonction applique l'option de socket requise. La fonction renvoie `undef` en cas d'échec. *NIVEAU* spécifie quelle couche de protocole est visée par l'appel, ou `SOL_SOCKET` pour la socket elle-même qui se trouve au sommet de toutes les couches. *VALEUR_OPTION* peut être `undef` si vous ne voulez pas passer d'argument. Il est courant d'appliquer l'option `SO_REUSEADDR`, pour contourner le fait que l'on ne peut pas faire de `bind` vers une adresse donnée alors que la connexion TCP précédente sur ce port est toujours en train de se demander s'il faut en finir. Cela peut ressembler à ceci :

```
use Socket; socket(SOCK, ...)
    or die "Impossible de créer une socket : $!\n";
setsockopt(SOCK, SOL_SOCKET, SO_REUSEADDR, 1)
    or warn "setsockopt impossible : $!\n";
```

Voir *setsockopt*(2) pour les autres valeurs possibles.

shift

```
shift TABLEAU
shift
```

Cette fonction enlève la première valeur du tableau et la renvoie, réduisant ainsi le tableau d'un élément et déplaçant toutes les autres d'une unité vers le bas. (Ou vers le haut, ou vers la gauche, selon votre vision du tableau. Nous aimons bien vers la gauche.) S'il n'y a pas d'éléments dans le tableau, la fonction renvoie `undef`.

Si *TABLEAU* est omis, la fonction décale `@_` dans la portée lexicale des sous-programmes et des formats ; elle décale `@ARGV` dans les portées de fichiers (généralement, le programme principal) ou à l'intérieur de la portée lexicale établie par les constructions `eval` *CHAINE*, `BEGIN {}`, `CHECK {}`, `INIT {}` et `END {}`.

Les sous-programmes commencent souvent par recopier leurs arguments dans des variables lexicales et peuvent utiliser `shift` pour ce faire :

```
sub marine {
    my $brasse    = shift;      # profondeur [13]
    my $poissons  = shift;      # nombre de poissons
    my $o2        = shift;      # concentration en oxygène
    # ...
}
```

`shift` est également employé pour traiter les arguments au début de votre programme :

```
while (defined($_ = shift)) {
    /^[^-]/       && do { unshift @ARGV, $_;   last };
    /^-w/         && do { $AVERTISSEMENTS = 1; next };
    /^-r/         && do { $RECURSION = 1;      last };
    die "Argument inconnu $_\n";
}
```

Vous pouvez également considérer les modules `Getopt::Std` et `GetOpt::Long` pour traiter les argument de votre programme.

Voir également `unshift`, `push`, `pop` et `splice`. Les fonctions `shift` et `unshift` font la même chose à l'extrémité gauche d'un tableau que `push` et `pop` à l'extrémité droite.

shmctl

```
shmctl ID, CMD, ARG
```

Cette fonction invoque l'appel système *shmctl*(2) des IPC System V. Vous devrez probablement d'abord faire `use IPC::SysV` pour obtenir les définitions de constantes correctes.

Si *CMD* vaut `IPC_STAT`, *ARG* doit alors être une variable qui contiendra la structure `shmid_ds` renvoyée. Comme pour `ioctl` et `fcntl`, les valeurs renvoyées sont `undef` en cas d'erreur, « `0 but true` » (*N.d.T.* : 0 mais vrai) pour zéro ou sinon la valeur réellement renvoyée.

Cette fonction n'est disponible que sur les systèmes supportant les IPC System V.

13. *N.d.T.* : unité égale à 1828 mm.

shmget

$! X U

```
shmget CLEF, TAILLE, FLAGS
```

Cette fonction invoque l'appel système *shmget*(2) des IPC System V. La fonction renvoie l'ID du segment de mémoire partagée ou `undef` en cas d'erreur. Avant l'appel, faites `use IPC::SysV`.

Cette fonction n'est disponible que sur les systèmes supportant les IPC System V.

shmread

$! X U

```
shmread ID, VAR, POS, TAILLE
```

Cette fonction lit depuis l'*ID* de segment de mémoire partagée en commençant à la position *POS* sur la longueur *TAILLE* (en s'y attachant, puis en copiant les données qui s'y trouvent et enfin en s'en détachant). *VAR* doit être une variable qui contiendra les données lues. La fonction renvoie vrai en cas de réussite et faux en cas d'erreur.

Cette fonction n'est disponible que sur les systèmes supportant les IPC System V.

shmwrite

$! X U

```
shmwrite ID, CHAINE, POS, TAILLE
```

Cette fonction écrit sur l'*ID* de segment de mémoire partagée en commençant à la position *POS* sur la longueur *TAILLE* (en s'y attachant, puis en y copiant des données et enfin en s'en détachant). Si *CHAINE* est trop longue, seulement *TAILLE* octets seront utilisés ; si *TAILLE* est trop courte, des caractères nuls sont écrit pour remplir *TAILLE* octets. La fonction renvoie vrai en cas de réussite et faux en cas d'erreur.

Cette fonction n'est disponible que sur les systèmes supportant les IPC System V. (Vous êtes certainement fatigués de lire ceci — nous commençons à être fatigués de l'écrire.)

shutdown

$! X ARG X U

```
shutdown SOCKET, COMMENT
```

Cette fonction arrête une connexion socket de la manière indiquée par *COMMENT*. Si *COMMENT* vaut `0`, toute réception ultérieure est interdite. Si *COMMENT* vaut `1`, toute émission ultérieure est interdite. Si *COMMENT* vaut `2`, tout est interdit.

```
shutdown(SOCK, 0);    # plus de lectures
shutdown(SOCK, 1);    # plus d'écritures
shutdown(SOCK, 2);    # plus du tout d'entrées/sorties
```

Ceci est utile avec les sockets lorsque vous voulez prévenir l'autre côté que vous avez fini d'écrire mais pas de lire, ou vice versa. C'est également une forme plus déterminée de fermeture car elle empêche toute copie de ces descripteurs détenus par des processus forkés.

Imaginez un serveur qui veut lire la requête de son client jusqu'à la fin de fichier, puis envoyer une réponse. Si le client appelle `close`, cette socket n'est maintenant plus valide pour les entrées/sorties, aucune réponse ne pourra donc être renvoyée. À la place, le

client doit utiliser `shutdown` pour fermer la moitié de la connexion :

```
print SERVEUR, "ma requete\n";   # envoi de données
shutdown(SERVEUR, 1);            # envoi de fin de ligne; plus d'écritures
$reponse = <SERVEUR>;            # mais vous pouvez encore lire
```

(Si vous êtes arrivé ici pour savoir comment arrêter un système, vous devrez exécuter un programme externe à cet effet. Voir `system`.)

sin

```
sin EXPR
sin
```

Désolé, même si cet opérateur évoque des choses malicieuses aux anglophones[14], il ne fait que renvoyer le sinus de *EXPR* (exprimée en radians).

Pour l'opération inverse,vous pouvez utiliser la fonction `asin` des modules `Math::Trig` ou `POSIX`, ou utiliser cette relation :

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

sleep

```
sleep EXPR
sleep
```

Cette fonction endort le script pendant *EXPR* secondes, ou éternellement si *EXPR* est omise. Elle renvoie le nombre de secondes réellement passées à dormir. Elle peut être interrompue en envoyant un signal `SIGALRM` au processus. Sur certains systèmes anciens, le repos peut durer jusqu'à une seconde de moins que ce que demandiez, selon la manière de compter les secondes. La plupart des systèmes modernes dorment toujours durant la période entière. Ils peuvent cependant sembler dormir plus que cela, car l'ordonnanceur d'un système multitâches peut ne pas donner tout de suite la main à votre processus. S'il est disponible, l'appel `select` (descripteurs de fichiers prêts) peut vous donner une meilleure résolution. Vous pouvez également employer `syscall` pour appeler les routines *getitimer*(2) et *setitimer*(2) supportée par certains systèmes Unix. Vous ne pouvez probablement pas mélanger des appels à `alarm` avec ceux à `sleep`, car `sleep` est souvent implémenté en utilisant `alarm`.

Voir également la fonction `sigpause` du module `POSIX`.

socket

```
socket SOCKET, DOMAINE, TYPE, PROTOCOLE
```

Cette fonction ouvre une socket du type spécifié et l'attache au handle de fichier *SOCKET*. *DOMAIN*, *TYPE* et *PROTOCOLE* sont spécifiés comme pour *socket*(2). Si le handle de fichier *SOCKET* est indéfini, il sera autovivifié. Avant d'utiliser cette fonction, votre programme doit contenir la ligne :

```
use Socket;
```

14. *N.d.T.* : en anglais « sin » signifie « péché ».

Cela vous donne accès aux constantes appropriées. La fonction renvoie vrai en cas de réussite. Vois les exemple dans la section « Sockets » au chapitre 16.

Sur les systèmes qui supportent le flag close-on-exec pour les fichiers, ce flag sera positionné pour le nouveau descripteur de fichier ouvert, comme le détermine la valeur de `$^F`. Voir la variable `$^F` (`$SYSTEM_FD_MAX`) au chapitre 28.

socketpair

$! X ARG X T X U

```
socketpair SOCKET1, SOCKET2, DOMAINE, TYPE, PROTOCOLE
```

Cette fonction crée une paire de sockets non nommées dans le domaine spécifié, d'un type spécifié. *DOMAINE*, *TYPE* et *PROTOCOLE* sont spécifiés de la même manière que pour *socketpair*(2). Si l'un des arguments sockets (*SOCKET1* ou *SOCKET2*) est indéfini, il sera autovivifié. La fonction renvoie vrai en cas de succès, sinon faux. Sur un système où *socketpair*(2) n'est pas implémenté, l'appel à cette fonction lève une exception.

Cette fonction est généralement utilisée juste avant un `fork`. L'un des processus résultants doit fermer *SOCKET1* et l'autre *SOCKET2*. Vous pouvez utiliser ces sockets de manière bidirectionnelle, au contraire des handles de fichiers créés par la fonction `pipe`. Certains systèmes définissent `pipe` en termes de `socketpair`, auquel cas un appel à `pipe(Lec, Ecr)` est équivalent pour l'essentiel à :

```
use Socket;
socketpair(Lec, Ecr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Lec, 1);        # plus d'écritures pour le lecteur
shutdown(Ecr, 0);        # plus de lectures pour l'écrivain
```

Sur les systèmes qui supportent le flag close-on-exec pour les fichiers, ce flag sera positionné pour le nouveau descripteur de fichier ouvert, comme le détermine la valeur de `$^F`. Voir la variable `$^F` (`$SYSTEM_FD_MAX`) au chapitre 28. Voir également l'exemple à la fin de la section *Communication bidirectionnelle* au chapitre 16.

sort

$@

```
sort SUB_UTILISATEUR LISTE
sort BLOC LISTE
sort LISTE
```

Cette fonction trie la *LISTE* et renvoie la valeur liste triée. Elle trie par défaut dans le sens de comparaison standard des chaînes (les valeurs indéfinies étant triées avant les chaînes vides définies, qui sont triées avant tout le reste). Lorsque le pragma `use locale` est actif, `sort` *LISTE* trie *LISTE* selon la locale courante *LC_COLLATE*.

SUB_UTILISATEUR, s'il est spécifié, est le nom d'un sous-programme qui renvoie un entier inférieur, égal ou supérieur à 0, selon la manière dont les éléments du tableau doivent être ordonnés. (Les opérateurs `<=>` et `cmp` peuvent être utilisés pour effectuer facilement des comparaisons de nombres et de chaînes pouvant donner trois résultats.) Si un *SUB_UTILISATEUR* est donné mais que cette fonction est indéfinie, `sort` lève une exception.

Pour des raisons d'efficacité, le code d'appel normal des sous-programmes est court-circuité, ce qui a pour conséquences que : le sous-programme ne peut pas être récursif (ni

contenir d'opérateur de contrôle de boucle qui vous ferait sortir du bloc ou de la routine) et que les deux éléments à comparer lui sont passés non *via* `@_`, mais en positionnant temporairement les variables globales par `$a` et `$b` dans le paquetage où `sort` a été compilé (voir les exemples ci-dessous). Les variables `$a` et `$b` étant passées en tant qu'alias des valeurs réelles, ne les modifiez pas dans le sous-programme.

Le sous-programme de comparaison doit bien se comporter. S'il renvoie des résultats inconsistants (en disant un jour que `$x[1]` est inférieur à `$x[2]` et en prétendant le contraire le lendemain), les conséquences sont aléatoires. (C'est une autre raison de ne pas modifier `$a` et `$b`.)

`SUB_UTILISATEUR` peut être un nom de variable scalaire (non indicée), dont la valeur fournit le nom du sous-programme à utiliser. (Un nom symbolique plutôt qu'une référence en dur est autorisé même lorsque le pragma `use strict 'refs'` est actif.) En lieu et place de *`SUB_UTILISATEUR`*, on peut fournir un *`BLOC`*, sorte de sous-programme anonyme en ligne.

Pour un tri numérique ordinaire, écrivez ceci :

```
sub numerique { $a <=> $b }
@tri_par_nombre = sort numerique 53,29,11,32,7;
```

Pour trier dans l'ordre décroissant, il vous suffit d'appliquer `reverse` après le `sort`, ou d'inverser l'ordre de `$a` et `$b` dans le sous-programme de tri :

```
@decroissant = reverse sort numerique 53,29,11,32,7;

sub inverse_numerique { $b <=> $a }
@decroissant = sort inverse_numerique 53,29,11,32,7;
```

Pour trier des chaînes sans tenir compte de la casse (majuscules/minuscules), passez `$a` et `$b` à `lc` avant la comparaison :

```
@desordonnee = qw/moineau Autruche ALOUETTE moqueurchat geaiBLEU/;
@ordonnee = sort { lc($a) cmp lc($b) } @desordonnee;
```

(Avec Unicode, l'emploi de `lc` pour la canonisation de la casse est plus ou moins préférable à l'utilisation de `uc` car certaines langues différencient les majuscules pour les titres des majuscules ordinaires. Mais cela ne compte pas pour un tri ASCII basique, et si vous voulez trier convenablement de l'Unicode, vos routines de canonisation seront bien plus imaginatives que `lc`.)

Le tri des valeurs d'un hachage est une utilisation courante de la fonction `sort`. Par exemple, si un hachage `%montant_ventes` enregistre les ventes de différents services, une recherche dans le hachage à l'intérieur de la routine de tri permet aux clefs du hachage d'être triées selon leurs valeurs correspondantes :

```
# tri dans l'ordre décroissant des ventes par service
sub par_vente { $montant_ventes{$b} <=> $montant_ventes{$a} }

for $serv (sort par_vente keys %montant_ventes) {
    print "$serv => $montant_ventes{$serv}\n";
}
```

Vous pouvez effectuer des niveaux supplémentaires de tri en faisant des comparaisons en cascade à l'aide des opérateurs `||` ou `or`. Cela marche parfaitement car les opérateurs

de comparaison renvoient convenablement 0 pour une équivalence, ce qui les fait entrer dans la comparaison suivante. Ici les clefs de hachage sont triées tout d'abord selon les montants des ventes associées puis selon les clefs elles-mêmes (dans le cas où deux départements ou plus auraient le même montant de ventes) :

```
sub par_vente_puis_serv {
    $montant_ventes{$b} <=> $montant_ventes{$a}
        ||
    $a cmp $b
}

for $serv (sort par_vente_puis_serv keys %montant_ventes) {
    print "$serv => $montant_ventes{$serv}\n";
}
```

En supposant que @enregs est un tableau de références sur des hachages, dans lequel chaque hachage comporte des champs comme PRENOM, NOM, AGE, TAILLE et SALAIRE, la routine suivante met au début de la liste triée les enregistrements pour les personnes les plus riches, puis les plus grandes, ensuite les plus jeunes et enfin les premières dans l'ordre alphabétique :

```
sub prospecte {
    $b->{SALAIRE} <=>  $a->{SALAIRE}
        ||
    $b->{TAILLE}  <=>  $a->{TAILLE}
        ||
    $a->{AGE}     <=>  $b->{AGE}
        ||
    $a->{NOM}     cmp  $b->{NOM}
        ||
    $a->{PRENOM}  cmp  $b->{PRENOM}
}
@triee = sort prospecte @enregs;
```

Toute information utile qui dérive de $a et de $b peut servir de base à une comparaison dans une routine de tri. Par exemple, si les lignes d'un texte doivent être triées selon des champs particuliers, split peut être utilisé dans la routine de tri pour dériver les champs :

```
@lignes_triees = sort {
    @champs_a = split /:/, $a;      # champs séparés par deux points
    @champs_b = split /:/, $b;

    $champs_a[3] <=> $champs_b[3]   # tri numérique sur le 4ème champ, puis
        ||
    $champs_a[0] cmp $champs_b[0]   # tri de chaînes sur le 1er champ, puis
        ||
    $champs_b[2] <=> $champs_a[2]   # tri numérique décroissant sur le 3ème
        ||                          # champ
    ...                             # etc.
} @lignes;
```

Toutefois, puisque sort exécute la routine de tri plusieurs fois en utilisant différentes

paires de valeurs pour $a et $b, l'exemple précédent redécoupera chaque ligne avec split plus souvent que nécessaire.

Pour éviter le coût de dérivations répétées comme l'éclatement de lignes pour comparer leurs champs, exécutez la dérivation une seule fois par valeur avant le tri et sauvegardez l'information dérivée. Ici, on crée des tableaux anonymes pour encapsuler chaque ligne ainsi que le résultat de son éclatement :

```
@temp = map { [$_, split /:/] } @lignes;
```

Puis, on trie les références du tableau :

```
@temp = sort {
    @champs_a = @$a[1..$#$a];
    @champs_b = @$b[1..$#$b];

    $champs_a[3] <=> $champs_b[3]   # tri numérique sur le 4ème champ, puis
        ||
    $champs_a[0] cmp $champs_b[0]   # tri de chaînes sur le 1er champ, puis
        ||
    $champs_b[2] <=> $champs_a[2]   # tri numérique décroissant sur le 3ème
        ||                          # champ
    ...                             # etc.
} @temp;
```

Maintenant que les références du tableau sont triées, on peut récupérer les lignes d'origine à partir des tableaux anonymes :

```
@lignes_triees = map { $_->[0] } @temp;
```

Si l'on rassemble tout cela, cette technique *map-sort-map*, souvent appelée transformation schwartzienne[15], peut être accomplie en une seule instruction :

```
@lignes_triees= map { $_->[0] }
                sort {
                    @champs_a = @$a[1..$#$a];
                    @champs_b = @$b[1..$#$b];

                    $champs_a[3] <=> $champs_b[3]
                        ||
                    $champs_a[0] cmp $champs_b[0]
                        ||
                    $champs_b[2] <=> $champs_a[2]
                        ||
                    ...
                }
                map { [$_, split /:/] } @lignes;
```

Ne déclarez pas $a et $b en tant que variables lexicales (avec my). Elles sont globales au paquetage (bien qu'exemptes des restrictions habituelles concernant les variables globales lorsque vous employez use strict). Vous devez cependant vous assurer que la rou-

15. *N.d.T.* : du nom de l'illustre co-auteur des précédentes éditions de cet ouvrage : Randal L. Schwartz.

tine de tri se trouve bien dans le même package, ou sinon qualifier `$a` et `$b` avec le nom du paquetage de l'appelant (voir la fonction `caller`).

Ceci étant dit, dans la version 5.6, vous *pouvez* écrire des sous-programmes de tri avec la méthode standard de passage d'arguments (et , sans coïncidence aucune, utiliser des sous-programmes XS comme routines de tri), pourvu que vous déclariez le sous-programme de tri avec un prototype valant `($$)`. Et dans ce cas, vous pouvez en fait déclarer `$a` et `$b` comme variables lexicales :

```
sub numerique ($$) {
    my ($a, $b) = @_;
    $a <=> $b;
}
```

Et un jour ou l'autre, lorsque des prototypes complets auront été implémentés, vous écrirez seulement :

```
sub numerique ($a, $b) { $a <=> $b }
```

et nous serons alors plus ou moins revenus à notre point de départ.

splice

$@

```
splice TABLEAU, DECALAGE, LONGUEUR, LISTE
splice TABLEAU, DECALAGE, LONGUEUR
splice TABLEAU, DECALAGE
splice TABLEAU
```

Cette fonction enlève les éléments désignés par *DECALAGE* et *LONGUEUR* dans un *TABLEAU*, et les remplace par les éléments de *LISTE*, s'il en existe. Si *DECALAGE* est négatif, la fonction compte à rebours depuis la fin du tableau, mais si cela vous faisait atterrir avant le début du tableau, une exception est levée. Dans un contexte de liste, `splice` renvoie les éléments supprimés du tableau. Dans un contexte scalaire, elle renvoie le dernier élément enlevé ou `undef` s'il n'y en a pas. Si le nombre de nouveaux éléments n'est pas égal au nombre d'anciens éléments, le tableau augmente ou diminue en taille selon les besoins, et la position des éléments qui suivent cette tranche ajoutée ou enlevée est décalée en conséquence. Si *LONGUEUR* est omise, la fonction enlève tout à partir de *DECALAGE*. Si *DECALAGE* est omis, le tableau est vidé au fur et à mesure qu'il est lu. Les colonnes suivantes sont équivalentes (à supposer que `$[` vaille 0) :

Méthode directe	**Équivalent splice**
`push(@a, $x, $y)`	`splice(@a, @a, 0, $x, $y)`
`pop(@a)`	`splice(@a, -1)`
`shift(@a)`	`splice(@a, 0, 1)`
`unshift(@a, $x, $y)`	`splice(@a, 0, 0, $x, $y)`
`$a[$x] = $y`	`splice(@a, $x, 1, $y)`
`(@a, @a =())`	`splice(@a)`

La fonction `splice` peut également servir à découper la liste d'arguments passée à un sous-programme. En supposant par exemple que les longueurs de liste sont passées

avant les listes elles-mêmes :

```
sub eq_liste {        # compare deux valeurs de liste
    my @a = splice(@_, 0, shift);
    my @b = splice(@_, 0, shift);
    return 0 unless @a == @b;     # même longueur ?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (eq_liste($long, @machin[1..$long], scalar(@truc), @truc)) { }
```

Toutefois, il serait probablement plus propre de se contenter d'utiliser des références sur des tableaux pour faire cela.

split

$_ T

```
split /MOTIF/, EXPR, LIMITE  split /MOTIF/, EXPR
split /MOTIF/
split
```

Cette fonction recherche des délimiteurs dans une chaîne *EXPR* et l'éclate en une liste de sous-chaînes, en renvoyant la valeur liste résultante dans un contexte de liste, ou le nombre de sous-chaînes dans un contexte scalaire.[16] Les délimiteurs sont déterminés par la correspondance répétée d'un motif, en utilisant l'expression régulière donnée dans *MOTIF*. Les délimiteurs peuvent donc être de n'importe quelle taille et ne sont pas forcément la même chaîne à chaque recherche. (Les délimiteurs ne sont généralement pas renvoyés, mais voyez les exceptions ci-dessous.) Si le *MOTIF* ne correspond à rien, `split` renvoie la chaîne d'origine comme une sous-chaîne unique. S'il trouve une occurrence, vous obtenez deux sous-chaînes, et ainsi de suite. Vous pouvez donner au *MOTIF* des modificateurs d'expressions régulières, comme `/`*MOTIF*`/i`, `/`*MOTIF*`/x`, etc. Lorsque vous éclatez avec un motif valant `/^/`, le modificateur `//m` est implicite.

Si *LIMITE* est spécifiée et positive, la fonction éclate la chaîne en, au plus, ce nombre de champs (bien qu'elle puisse l'éclater en moins si elle ne trouve pas assez de délimiteurs). Si *LIMITE* est négative, elle est traitée comme si une *LIMITE* arbitrairement grande avait été spécifiée. Si *LIMITE* est omise, les champs nuls qui restent sont éliminés du résultat (ce dont les utilisateurs potentiels de `pop` feraient bien de se souvenir). Si *EXPR* est omise, la fonction éclate la chaîne `$_`. Si *MOTIF* est également omis ou s'il s'agit d'un espace littéral, `" "`, la fonction éclate sur les caractères d'espacement, `/s+/`, après avoir éliminé tout espacement en début de chaîne.

On peut éclater des chaînes de n'importe quelle longueur :

```
@cars   = split //,  $mot;
@champs = split /:/, $ligne;
@mots   = split " ", $paragraphe;
@lignes = split /^/, $tampon;
```

16. Un contexte scalaire entraîne également le fait que `split` écrive son résultat dans `@_`, mais cet usage est déprécié.

Un motif capable de correspondre soit à la chaîne vide, soit à quelque chose de plus long que la chaîne vide (par exemple, un motif composé d'un seul caractère modifié par un * ou un ?) éclatera la valeur de *EXPR* en caractères séparés partout où il correspond à la chaîne vide entre les caractères ; les correspondances non-nulles passeront outre les occurrences du délimiteur comme habituellement. (En d'autres termes, un motif ne trouvera pas de correspondance en plus d'un endroit, même avec une longueur de zéro.) Par exemple :

```
print join ':', split / */, 'à la vôtre';
```

produira « `à:l:a:v:o:t:r:e` ». Les espaces ont disparu car ils ont provoqué une correspondance non-nulle. Dans un cas trivial, le motif nul `//` se contente d'éclater la chaîne en caractères séparés, et les espaces ne disparaissent pas. (Pour une correspondance de motif normale, un motif `//` répéterait le dernier motif de correspondance réussie, mais le motif de `split` ne tombe pas dans cette crevasse.)

Le paramètre *LIMITE* peut être utilisé pour n'éclater qu'une partie d'une chaîne :

```
($login, $mot_de_passe, $reste) = split /:/, $_, 3;
```

Nous vous encourageons à éclater des listes de noms comme ceci afin d'autodocumenter votre code. (Pour des raisons de vérification d'erreurs, remarquez que `$reste` serait indéfini s'il y avait moins de 3 champs.) Quand une liste se voit assigner une valeur, si *LIMITE* est omise, Perl fournit une *LIMITE* supérieure de 1 au nombre de variables de la liste pour éviter un travail inutile. Pour la liste ci-dessus, *LIMITE* aurait été de 4 par défaut, et `$reste` n'aurait reçu que le troisième champ et non le reste de la ligne. Dans des applications où le temps compte, il vaut mieux ne pas éclater en plus de champs que ce dont vous avez vraiment besoin. (Le problème avec les langages puissants, c'est qu'ils vous laissent parfois être des idiots en puissance.)

Nous avons dit plus haut que les délimiteurs n'étaient pas renvoyés, mais si le *MOTIF* contient des parenthèses, les champs habituellement renvoyés viennent s'intercaler entre chaque occurrence des délimiteurs entre parenthèses qui est également inclue dans la liste résultante. Voici un exemple simple :

```
split /([-,])/, "1-10,20";
```

produit la valeur de liste :

```
(1, '-', 10, ',', 20)
```

S'il y a plus de parenthèses, un champ est renvoyé pour chaque paire, même si certaines ne donnent pas de correspondances, auquel cas des valeurs indéfinies sont renvoyées à ces positions. Ainsi, si vous écrivez :

```
split /(-)|(,)/, "1-10,20";
```

vous obtiendrez la valeur :

```
(1, '-', undef, 10, undef, ',', 20)
```

L'argument */MOTIF/* peut être remplacé par une expression spécifiant les motifs qui varient à l'exécution. Comme pour les motifs ordinaires, pour ne compiler à l'exécution qu'une seule fois, utilisez `/$variable/o`.

Il existe un cas spécial où la spécification d'un espace unique (`" "`) éclate sur les caractères d'espacement tout comme `split` sans argument. `split(" ")` peut donc être utilisé pour émuler le comportement de *awk* par défaut. Alors qu'au contraire `split(/ /)`

vous donnera autant de champs nuls initiaux qu'il existe d'espaces en en-tête. (Par ailleurs, si l'on fournit une chaîne au lieu d'une expression rationnelle, elle sera de toute façon interprétée comme une expression rationnelle.) Vous pouvez utiliser cette propriété pour supprimer les espaces au début et à la fin d'une chaîne et pour rassembler les séquences de caractères d'espacement consécutifs en un seul espace :

```
$chaine = join(' ', split(' ', $chaine));
```

L'exemple suivant éclate un en-tête de message RFC-822 en un hachage contenant `$tete{$Date}`, `$tete{Subject}`, et ainsi de suite. Il utilise l'astuce d'assigner une liste de paire à un hachage, en se basant sur le fait que les délimiteurs alternent avec des champs délimités. Il emploie les parenthèses pour renvoyer les parties de chaque délimiteur comme faisant partie de la valeur de liste renvoyée. Comme le motif de `split` garantit le renvoi des éléments en paires par la vertu de contenir un seul jeu de parenthèses, l'affectation du hachage est sûr de recevoir une liste composée de paires clef/valeur, où chaque clef est le nom d'un champ d'en-tête. (Cette technique fait malheureusement perdre des informations sur plusieurs lignes avec le même champ de clef, comme les lignes Received-By. Ah, et puis zut. . . .)

```
$entete =~ s/\n\s+/ /g; # Réunit les lignes de continuation.
%tete = ('PREAMBULE', split /^(\S*?):\s*/m, $entete);
```

L'exemple suivant traite les entrées d'un fichier Unix *passwd*(5). Vous pourriez éviter le `chomp`, auquel cas `$shell` se terminerait par un saut de ligne.

```
open MOT_DE_PASSE, '/etc/passwd';
while (<MOT_DE_PASSE>) {
    chomp;      # supprime le saut de ligne final
    ($login, $mot_de_passe, $uid, $gid, $nom_reel, $rep, $shell) =
        split /:/;
    ...
}
```

Voici comment traiter chaque mot de chaque ligne de chaque fichier d'entrée pour créer un hachage des fréquences de mots :

```
while (<>) {
    foreach $mot (split) {
        $compteur{$mot}++;
    }
}
```

L'inverse de `split` est effectué par `join` (sauf que `join` ne peut joindre qu'avec le même délimiteur entre tous les champs). Pour éclater une chaîne sur des champs de longueur fixe, employez `unpack`.

sprintf

```
sprintf FORMAT, LISTE
```

Cette fonction renvoie une chaîne formatée suivant les conventions habituelles de `printf` gouvernées par la fonction *sprintf* de la bibliothèque C. Voir *sprintf* (3) ou *printf* (3) sur votre système pour une explication des principes généraux. La chaîne *FORMAT* contient du texte comprenant des spécificateurs de champs auxquels les éléments de *LISTE* sont substitués, à raison d'un élément par champ.

Perl réalise son propre formatage pour `sprintf` — il émule la fonction C *sprintf*, mais il ne l'utilise pas.[17] Il en résulte que toutes les extensions non-standard dans votre fonction *sprintf* (3) locale ne sont pas accessibles depuis Perl.

Le `sprintf` de Perl permet d'utiliser les conversions universellement connues qui sont montrées dans le *tableau 29-4*.

Tableau 29-4. Formats pour sprintf

Champ	Signification
`%%`	Un signe pourcent.
`%c`	Un caractère avec le nombre donné.
`%s`	Une chaîne.
`%d`	Un entier signé, en décimal.
`%u`	Un entier non-signé, en décimal.
`%o`	Un entier non-signé, en octal.
`%x`	Un entier non-signé, en hexadécimal.
`%e`	Un nombre à virgule flottante, en notation scientifique.
`%f`	Un nombre à virgule flottante, en notation décimale fixe.
`%g`	Un nombre à virgule flottante, en notation `%e` ou `%f`.

De plus, Perl permet d'utiliser les conversion suivantes, largement supportées :

Champ	Signification
`%X`	Comme `%x`, mais avec des lettres majuscules.
`%E`	Comme `%e`, mais avec un « E » majuscule.
`%G`	Comme `%g`, mais avec un « E » majuscule (si c'est applicable).
`%b`	Un entier non-signé, en binaire.
`%p`	Un pointeur (affiche l'adresse de la valeur Perl en hexadécimal).
`%n`	Spécial : *sauvegarde* le nombre de caractères de sortie jusque là dans la prochaine variable de la liste d'arguments.

Enfin, pour une compatibilité antérieure (et nous voulons vraiment dire « antérieure »), Perl permet d'utiliser ces conversions inutiles mais largement supportées :

Champ	Signification
`%i`	Un synonyme pour `%d`.
`%D`	Un synonyme pour `%ld`.

17. Mise à part pour les nombres à virgule flottante, et même dans ce cas, seulement les modificateurs standards sont autorisés.

Champ	Signification
%U	Un synonyme pour %lu.
%O	Un synonyme pour %lo.
%F	Un synonyme pour %f.

Perl permet d'utiliser les flags suivants universellement connus entre le % et le caractère de conversion :

Flag	Signification
espace	Préfixe un nombre positif par un espace.
+	Préfixe un nombre positif par un signe plus.
-	Justifie le champ vers la gauche.
0	Utilise des zéros, et non des espaces, pour justifier vers la droite.
#	Préfixe un octal non nul par "0" et un hexadécimal non nul par "0x".
nombre	Largeur minimale du champ.
.nombre	« Précision » : chiffres après la virgule pour les nombres à virgule flottante, longueur maximale pour les chaînes, longueur minimale pour les entiers.
l	Interprète l'entier comme un type C long ou unsigned long.
h	Interprète l'entier comme un type C short ou unsigned short (si aucun flag n'est fourni, l'entier est interprété comme un type C int ou unsigned).

Il existe également ces deux flags spécifiques à Perl :

Flag	Signification
V	Interprète l'entier comme le type entier standard de Perl.
v	Interprète la chaîne comme un vecteur d'entiers, affichés comme des nombres séparés soit par des points, soit par une chaîne arbitraire reçue depuis la liste d'arguments lorsque le flag est précédé par *.

Si votre Perl connaît les « quads » (entiers de 64 bits) soit parce que la plate-forme les supporte nativement, soir parce que Perl a été spécialement compilé avec cette possibilité, alors les caractères d u o x X b i D U O affichent des quads et ils peuvent optionnellement être précédés par ll, L ou q. Par exemple, %lld %16LX %qo. Si Perl connaît les « longs doubles » (ceci requiert que la plate-forme supporte les longs doubles), les flags e f g E F G peuvent optionnellement être précédé par ll ou L. Par exemple, %llf %Lf.

Là où un nombre pourrait apparaître dans les flags, un astérisque (« * ») peut être utilisé en lieu et place, auquel cas Perl utilise la prochaine entité dans la liste d'arguments comme le nombre donné (c'est-à-dire, comme le champ de largeur ou de précision). Si un champ de largeur, obtenu avec « * » est négatif, il entraîne les mêmes effets qu'un flag « - » : une justification vers la gauche.

Le flag v est utilisé pour afficher les valeurs ordinales de caractères dans des chaînes arbitraires :

```
sprintf "la version est v%vd\n", $^V;         # version de Perl
sprintf "l'adresse est %vd\n", $adr;          # adresse IPv4
sprintf "l'adresse est %*vX\n", $adr;         # adresse IPv6
sprintf "les bits sont %*vb\n", " ", $bits;   # chaînes de bits aléatoires
```

sqrt

$_ $@

```
sqrt EXPR
sqrt
```

Cette fonction renvoie la racine carrée de *EXPR*. Pour les autre racines, comme la racine cubique, vous pouvez utiliser l'opérateur ** pour élever quelque chose à une puissance fractionnaire. N'essayez par l'une de ces approches avec des nombres négatifs, car cela devient un problème légèrement plus complexe (et lève une exception). Mais il existe un module standard pour s'occuper de cela aussi :

```
use Math::Complex;
print sqrt(-2);      # affiche 1.41421356237731i
```

srand

```
srand EXPR
srand
```

Cette fonction positionne la « graine » aléatoire pour l'opérateur rand. Si *EXPR* est omise, elle utilise une valeur semi-aléatoire fournie par le noyau (si il supporte le périphérique */dev/urandom*) ou basée sur l'heure et la date courante et le PID, entre autres choses. Il n'est généralement pas du tout nécessaire d'appeler srand, car si on ne l'appelle pas explicitement, l'opérateur rand l'invoque implicitement. Toutefois, ce n'était pas vrai dans les version de Perl antérieures à 5.004, donc si votre script a besoin de tourner avec des versions de Perl plus anciennes, il doit appeler srand.

Les programmes appelés fréquemment (comme les scripts CGI) qui n'utilisent que time ^ $$ pour générer une graine aléatoire peuvent les deux tiers du temps être victimes de la propriété mathématique a^b == (a+1)^(b+1). Alors ne faites pas cela. À la place, utilisez :

```
srand( time() ^ ($$ + ($$ << 15)) );
```

Vous aurez besoin de plus d'une graine plus aléatoire que celle par défaut pour faire de la cryptographie. Sur certains systèmes le périphérique */dev/urandom* est approprié. Sinon, la méthode habituelle est de faire une somme de contrôle (*N.d.T.* : *checksum*) sur la sortie compressée d'un ou plusieurs programmes donnant l'état du système d'exploitation en tenant compte du changement rapide de ces états. Par exemple :

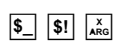
```
srand (time ^ $$ ^ unpack "%L*", 'ps wwaxl | gzip');
```

Si vous êtes particulièrement concernés par ce problème, regardez le module Math::TrulyRandom dans CPAN.

N'appelez *pas* srand plusieurs fois dans un programme à moins de savoir exactement ce que vous faites et pourquoi vous le faites. Le but de cette fonction est d'initialiser une

« graine » pour la fonction `rand` pour qu'elle puisse produire un résultat différent à chaque lancement du programme. Ne l'appelez qu'une seule fois au début, faute de quoi `rand` ne générera *pas* de nombres aléatoires !

stat



```
stat HANDLE_FICHIER
stat EXPR
stat
```

Dans un contexte scalaire, cette fonction renvoie un booléen indiquant si l'appel a réussi. Dans un contexte de liste, elle renvoie une liste de 13 éléments donnant l'état d'un fichier, qu'il soit ouvert *via* `HANDLE_FICHIER` ou nommé par `EXPR`. Elle est généralement utilisée comme suit :

```
($periph,$inode,$mode,$nb_liens,$uid,$gid,$type_periph,$taille,
    $date_a,$date_m,$date_c, $taille_bloc,$blocs)
        = stat $nom_fichier;
```

Tous les champs ne sont pas supportés par tous les systèmes d'exploitation, les champs non-supportés renvoient 0. Voici les significations des champs :

Tableau 29-5. Champs renvoyés par stat

Index	Champ	Signification
0	`$periph`	Numéro de périphérique du système de fichier.
1	`$inode`	Numéro d'inode.
2	`$mode`	Mode du fichier (type et permissions).
3	`$nb_liens`	Nombre de liens (en dur) vers le fichier.
4	`$uid`	UID numérique du propriétaire du fichier.
5	`$gid`	GID numérique du groupe désigné du fichier.
6	`$type_periph`	Identificateur du périphérique (pour les fichiers spéciaux seulement).
7	`$taille`	Taille totale du fichier, en octets.
8	`$date_a`	Dernière date d'accès au fichier en secondes depuis l'origine.
9	`$date_m`	Dernière date de modification du fichier en secondes depuis l'origine.
10	`$date_c`	Date de changement de l'inode (et *non* de création !) en secondes depuis l'origine.
11	`$taille_bloc`	Taille de bloc préféré pour les entrées/sorties sur le système de fichier.
12	`$blocs`	Nombre de blocs réellement alloués.

`$periph` et `$inode`, pris ensemble, identifient un fichier de manière unique. Les `$taille_bloc` et `$blocs` ne sont définis que sur les systèmes de fichiers dérivés de BSD. Le champ `$blocs` (s'il est défini) est donné en blocs de 512 octets. La valeur de

`$blocs*512` peut énormément différer de `$taille` pour des fichiers contenant des blocs non alloués, ou « trous », qui ne sont pas comptés dans `$blocs`. Si on passe à `stat` le handle de fichier spécial composé d'un souligné, aucune collection d'informations n'a lieu, mais le contenu actuel de la structure datant du dernier `stat`, `lstat` ou du dernier test de fichier en dépendant (comme `-r`, `-w` ou `-x`) sont renvoyés.

Comme le mode comprend à la fois le type du fichier et ses permissions, vous devez masquer la partie du type et faire `printf` ou `sprintf` avec `"%o"` si vous voulez voir les permissions réelles :

```
$mode = (stat($nom_fichier))[2];
printf "Les permissions sont %04o\n", $mode & 07777;
```

Le module `File::stat` fournit un mécanisme d'accès simplifié, orienté « par nom » :

```
use File::stat; $sb = stat($nom_fichier);
printf "Le fichier est %s, sa taille %s, ses permission %04o, date_m %s\n",
    $nom_fichier, $sb->size, $sb->mode & 07777,
    scalar localtime $sb->mtime;
```

Vous pouvez également importer les définitions symboliques des divers bits de mode depuis le module `Fcntl`. Voir la documentation en ligne pour plus de détail.

Astuce : si vous n'avez seulement besoin que de la taille du fichier, essayez l'opérateur de test de fichier `-s`, qui renvoie directement la taille en octets. Il existe également des tests de fichiers qui renvoie l'âge du fichier en nombre de jours.

study

```
study SCALAIRE
study
```

Cette fonction prend du temps supplémentaire pour étudier *SCALAIRE* afin d'anticiper de nombreuses recherches de motifs sur la chaîne avant qu'elle ne soit ensuite modifiée. Ceci peut ou non faire gagner du temps, selon la nature et le nombre de motifs de vos recherches, et selon la distribution des fréquences de caractères dans la chaîne recherchée — il vaut mieux que vous compariez les temps d'exécution avec et sans `study` pour voir ce qui tourne le plus vite. Les boucles qui cherchent de nombreuses chaînes courtes et constantes (y compris les parties constantes de motifs plus complexes) sont celles qui bénéficieront le plus de `study`. Si tous vos motifs sont des chaînes constantes, ancrées au début, `study` ne vous sera d'aucun secours car aucune scrutation ne sera effectuée. Vous ne pouvez avoir qu'un seul `study` (*N.d.T.* : étude) actif à la fois ; si vous étudiez un scalaire différent, le premier est « dé-étudié ».

`study` fonctionne de la manière suivante : une liste chaînée de chacun des caractères de la chaîne est construite de façon à savoir, par exemple, où se trouvent tous les caractères « `k` ». Pour chaque chaîne, le caractère le plus rare est sélectionné, en se fondant sur des tables de fréquence statiques construites par des programmes C sur du texte anglais. Seuls les endroits contenant ce caractère rare sont examinés.

Par exemple, voici une boucle qui insère des entrées d'index avant toute ligne contenant un certain motif :

```
while (<>) {
    study;
```

```
    print ".IX machin\n"  if /\bmachin\b/;
    print ".IX truc\n"    if /\btruc\b/;
    print ".IX bidule\n"  if /\bbidule\b/;
    ...
    print;
}
```

Pendant cette recherche sur `/\bmachin\b/`, seuls les endroits de `$_` contenant « h » seront examinées car « h » est plus rare que les autres lettres. En général, le gain est indiscutable, sauf dans des cas pathologiques. La seule question est de savoir si cela fait gagner plus de temps que de construire d'abord la liste chaînée.

Si vous devez rechercher des chaînes que vous ne connaissez qu'au moment de l'exécution, vous pouvez construire un boucle complète sous forme de chaîne et l'`eval`uer pour éviter la recompilation systématique de tous vos motifs. Ceci, associé à `$/` pour entrer des fichiers complets en un seul enregistrement, peut être extrêmement rapide, souvent plus rapide que des programmes spécialisés comme *fgrep*(1). La routine suivante scrute une liste de fichiers (`@fichiers`) pour trouver une liste de mots `@mots`), et affiche les noms des fichiers contenant une correspondance sans tenir compte de la casse :

```
$recherche = 'while (<>) { study;';
foreach $mot (@mots) {
    $recherche .= "++\$vu{\$ARGV} if /\\b$mot\\b/i;\n";
}
$recherche .= "}";
@ARGV = @fichiers;
undef $/;             # gobe chaque fichier en entier
eval $recherche;      # pied au plancher
die $@ if $@          # en cas d'échec de eval
$/ = "\n";            # remet le délimiteur d'entrée normal
foreach $fichier (sort keys(%vu)) {
    print $fichier, "\n";
}
```

Maintenant que nous avons l'opérateur `qr//`, les `eval`s compliqués à l'exécution comme celui ci-dessus sont moins indispensables. Ceci fait la même chose :

```
@motifs = ();
foreach $mot (@mots) {
    push @motifs, qr/\b$mot\b/i;
}
@ARGV = @fichiers;
undef $/;             # gobe chaque fichier en entier
while (<>) {
    for $motif (@motifs) {
        $vu{$ARGV}++ if /$motif/;
    }
}
$/ = "\n";            # remet le délimiteur d'entrée normal
foreach $fichier (sort keys(%vu)) {
    print $fichier, "\n";
}
```

sub

Déclarations nommées :

```
sub NOM PROTO ATTRS
sub NOM ATTRS
sub NOM PROTO
sub NOM
```

Définitions nommée :

```
sub NOM PROTO ATTRS BLOC
sub NOM ATTRS BLOC
sub NOM PROTO BLOC
sub NOM BLOC
```

Définitions anonymes :

```
sub PROTO ATTRS BLOC
sub ATTRS BLOC
sub PROTO BLOC
sub BLOC
```

La syntaxe des déclarations et définitions de sous-programmes paraît compliquée, mais elle est vraiment assez simple en pratique. Tout est basé sur la syntaxe :

```
sub NOM PROTO ATTRS BLOC
```

Les quatre arguments sont tous optionnels ; la seule restriction est que les champs présents doivent être présents dans cet ordre ; et que vous devez utiliser au moins un des champs *NOM* ou *BLOC*. Pour le moment nous ignorerons les champs *PROTO* et *ATTRS* ; ce sont seulement des modificateurs de la syntaxe de base. Les champs *NOM* et *BLOC* sont les parties importantes pour faire court :

- Si vous n'avez qu'un *NOM* mais pas de *BLOC*, il s'agit d'une déclaration de ce nom (et si jamais vous voulez appelez le sous-programme, vous devrez fournir plus tard une définition avec à la fois *NOM* et *BLOC*). Les déclarations nommées sont spécialement utiles pour que l'analyseur syntaxique traite un nom de manière spéciale s'il sait qu'il s'agit d'un sous-programme défini par l'utilisateur. Vous pouvez appelez ceci un sous-programme ou une fonction ou encore un opérateur, exactement comme les fonctions internes. Ces déclarations sont parfois appelées déclarations *anticipées*.
- Si vous avez à la fois un *NOM* et un *BLOC*, il s'agit d'une définition standard d'un sous-programme nommé (et également une déclaration, si vous n'avez pas déclaré le nom précédemment). Les définitions nommées sont spécialement utiles car le *BLOC* associe un véritable sens (le corps du sous-programme) à la déclaration. C'est exactement ce que cela signifie lorsque nous disons qu'elle définit le sous-programme au lieu de simplement le déclarer. La définition ressemble toutefois à la déclaration, en ce que le code l'encadrant ne la voit pas, et qu'elle ne renvoie pas de valeur en ligne avec laquelle vous pourriez référencer le sous-programme.
- Si vous n'avez qu'un *BLOC* mais pas de *NOM*, il s'agit d'une définition sans nom, c'est-à-dire, un sous-programme anonyme. Puisqu'elle n'a pas de nom, ce n'est absolument pas une déclaration, mais un opérateur réel qui renvoie à l'exécution une référence vers le corps du sous-programme anonyme. C'est extrêmement utile pour

traiter du code en tant que donnée. Cela vous permet de passer de menus morceaux de code en tant que fonctions de rappel (*N.d.T.* : *callbacks*), ou peut-être même en tant que fermetures si l'opérateur de définition `sub` se réfère à des variables lexicales qui se trouvent à l'extérieur de son propre code. Cela signifie que différents appels au même opérateur `sub` feront la comptabilité nécessaire pour conserver la visibilité de la « version » correcte de chacune de ces variables lexicales le temps où la fermeture existe, même si la portée d'origine des variables lexicales a été détruite.

Dans chacun de ces trois cas, soit l'un, soit les deux arguments *`PROTO`* et *`ATTRS`* peuvent apparaître après le *`NOM`* et avant le *`BLOC`*. Un prototype est une liste de caractères entre parenthèses qui indique à l'analyseur syntaxique comment traiter les arguments d'une fonction. Les attributs sont introduits par un deux-points et fournissent des informations supplémentaires à l'analyseur concernant la fonction. Voici une définition typique qui inclut tous les quatre champs :

```
num_ch_cmp ($$) : locked {
    my ($a, $b) = @_;
    return $a <=> $b || $a cmp $b;
}
```

pour plus de détails sur les listes d'attributs et leur manipulation, voir le pragma `attributes` au chapitre 31. Voir également le chapitre 6 et la section *Sous-programmes anonymes* au chapitre 8.

substr

```
subtsr EXPR, DECALAGE, LONGUEUR, REMPLACEMENT
subtsr EXPR, DECALAGE, LONGUEUR
subtsr EXPR, DECALAGE
```

Cette fonction extrait une sous-chaîne de la chaîne donnée par *`EXPR`* et la renvoie. La sous-chaîne est extraite en commençant à *`DECALAGE`* caractères depuis le premier. (Remarque : si vous avez fait le malin avec `$[`, le début de la chaîne n'est pas à 0, mais comme vous avez été sage (n'est-ce pas ?) il n'y a pas de problème). Si *`DECALAGE`* est négatif, la sous-chaîne démarre à cette distance de la fin de la chaîne. Si *`LONGUEUR`* est omise, toute la fin de la chaîne est renvoyée. Si *`LONGUEUR`* est négative, la longueur est calculée pour laisser ce nombre de caractères depuis la fin de la chaîne. Sinon, *`LONGUEUR`* indique la longueur de la sous-chaîne à extraire, ce à quoi vous pouviez vous attendre.

Vous pouvez utiliser `substr` comme une lvalue (quelque chose à laquelle on peut assigner une valeur), auquel cas *`EXPR`* doit aussi en être une. Si l'on assigne une valeur d'une longueur inférieure à *`LONGUEUR`*, la chaîne sera raccourcie et si l'on assigne une valeur d'une longueur supérieure à *`LONGUEUR`*, la chaîne s'allongera pour s'en accommoder. Il vous faudra peut-être compléter ou couper la valeur avec `sprintf` ou l'opérateur `x` pour conserver une chaîne de même longueur. Si vous tentez une affectation dans une zone non-allouée après la fin la chaîne, `substr` lève une exception.

Pour préfixer la valeur courante de `$var` par la chaîne `"Achille"` , écrivez :

```
substr($var, 0, 0) = "Achille";
```

Pour remplacer le premier caractère de `$var` par `"Talon"` , écrivez :

```
substr($var, 0, 1) = "Talon";
```

Et enfin, pour remplacer le dernier caractère de `$var` par `"Lefuneste"` , écrivez :

```
substr($var, -1, 1) = "Lefuneste";
```

Une alternative à l'utilisation de `substr` en tant que lvalue est l'emploi de la chaîne *`REMPLACEMENT`* comme quatrième argument. Ceci vous permet de remplacer des partie de l'*`EXPR`* et de renvoyer ce qu'il y avait avant en une seule opération, exactement comme vous pouvez le faire avec `splice`. Le prochain exemple remplace également le dernier caractère de `$var` par `"Lefuneste"` et met la chaîne remplacée dans `$avant` :

```
$avant = substr($var, -1, 1, "Lefuneste");
```

`substr` en tant que lvalue n'est pas réservée seulement aux affectations. Cet exemple remplace les espaces par des points, mais seulement dans les 10 derniers caractères de la chaîne :

```
substr($var, -10) =~ s/ /./g;
```


symlink

$! X T X U

```
symlink ANCIEN_NOM, NOUVEAU_NOM
```

Cette fonction crée un nouveau nom de fichier lié symboliquement à l'ancien nom de fichier. La fonction renvoie vrai en cas de succès, sinon faux. Sur les systèmes ne supportant pas les liens symboliques, elle lève une exception à l'exécution. Pour éviter cela, utilisez `eval` pour intercepter l'erreur potentielle :

```
$symlink_existe = eval { symlink("", ""); 1 };
```

Ou utilisez le module `Config`. Soyez prudent si vous fournissez un lien symbolique relatif, car cela sera interprété relativement à la position du lien symbolique lui-même, et non à votre répertoire de travail courant.

Voir également `link` et `readlink` plus haut dans ce chapitre.

syscall

$! X RO X T X U

```
syscall LISTE
```

Cette fonction invoque l'appel système (nous voulons dire un appel système, et pas une commande shell) spécifié dans le premier argument de la liste en passant les éléments restants comme arguments à l'appel système. (Beaucoup d'entre eux sont maintenant plus directement disponibles par le biais de modules comme `POSIX`). La fonction lève une exception si *syscall*(2) n'est pas implémenté.

Les arguments sont interprétés comme suit : si un argument donné est numérique, l'argument est passé comme un entier C. Sinon, c'est un pointeur sur la valeur chaîne qui est passé. Vous devez vous assurer que la chaîne est assez longue pour recevoir tout résultat pouvant y être écrit ; sinon vous chercher la copie du cœur du programme (*core dump*). Vous ne pouvez pas utiliser une chaîne littérale (ou toute autre chaîne en lecture seule) comme argument de syscall car Perl doit prévoir que l'on puisse écrire dans tout pointeur de chaîne. Si vos arguments entiers ne sont pas des littéraux et n'ont jamais été interprétés dans un contexte numérique, vous devrez peut-être leur ajouter 0 pour les forcer à ressembler à des nombres.

`syscall` renvoie toute valeur retournée par l'appel système invoqué. D'après les con-

ventions de codage C, si l'appel système échoue, `syscall` renvoie `-1` et positionne `$!` (errno). Certains appels système renvoient légitimement `-1` en cas de réussite. La manière appropriée de gérer de tels appels système et d'assigner `$!=0;` avant l'appel et de vérifier la valeur de `$!` si `syscall` renvoie `-1`.

Tous les appels système ne sont pas accessibles de cette manière. Par exemple, Perl peut passer jusqu'à 14 arguments à votre appel système, ce qui doit normalement suffire en pratique. Il existe cependant un problème avec les appels système qui renvoient des valeurs multiples. Prenez `syscall(&SYS_pipe)` : il renvoie le numéro de fichier de l'extrémité de lecture du pipe qu'il crée. Il n'y a aucun de moyen de récupérer le numéro de fichier de l'autre extrémité. Vous pouvez éviter ce type de problème en utilisant `pipe` à la place. Pour résoudre le problème générique, écrivez des XSUB (modules de sous-programmes externes, un dialecte de C) pour accéder aux appels système directement. Puis mettez votre nouveau module sur CPAN et devenez mondialement populaire.

Le sous-programme suivant renvoie l'heure courante en tant que nombre à virgule flottante plutôt que le nombre entier de secondes renvoyé par `time`. (Il ne fonctionne que sur les machines qui supporte l'appel système *gettimeofday*(2).)

```
sub heure_precise() {
    package main;   # pour le prochain require
    require 'syscall.ph';
    # prépare le tampon pour qu'il soit de la taille de deux entiers
    # longs de 32-bits
    my $tv = pack("LL", ());
    syscall(&SYS_gettimeofday, $tv, undef) >=0
        or die "gettimeofday : $!";
    my ($secondes, $microsecondes) = unpack("LL", $tv);
    return $secondes + ($microsecondes /1_000_000);
}
```

Imaginez que Perl ne supporte pas l'appel système *setgroups*(2)[18], mais que votre noyau l'implémente. Vous pourriez toujours l'atteindre de cette manière :

```
require 'syscall.ph';
syscall(&SYS_setgroups; scalar @nouveaux_gids, pack("i*", @nouveaux_gids))
        or die "setgroups : $!";
```

Vous devrez peut-être lancer `h2ph`, comme l'indiquent les instructions d'installations de Perl, pour créer *syscall.ph*. Certains systèmes peuvent nécessiter à la place un canevas pour `pack` de `"II"`. Encore plus perturbant, la fonction `syscall` suppose une équivalence de taille entre les types C `int`, `long` et `char*`. Essayez de ne pas considérer `syscall` comme le parangon de la portabilité.

Voir le module `Time::HiRes` de CPAN pour une approche plus rigoureuse des problèmes de dates et d'heures d'une granularité plus fine.

sysopen

$! X ARG

```
sysopen HANDLE_FICHIER, NOM_FICHIER, MODE, MASQUE
sysopen HANDLE_FICHIER, NOM_FICHIER, MODE
```

18. Bien qu'il le supporte *via* `$(`.

La fonction sysopen ouvre le fichier dont le nom est donné par *NOM_FICHIER* et l'associe avec *HANDLE_FICHIER*. Si *HANDLE_FICHIER* est une expression, sa valeur est employée comme le nom du handle de fichier ou comme une référence sur ce dernier. Si *HANDLE_FICHIER* est une variable dont la valeur est indéfinie, une valeur sera créée pour vous. La valeur de retour est vrai si l'appel réussit, sinon faux.

Cette fonction est une interface directe à l'appel *open*(2) de votre système d'exploitation suivi par un appel de bibliothèque *fdopen*(3). En tant que tel, vous devrez ici prétendre être un peu un programmeur C. Les valeurs possibles et les flags de bits du paramètre *MODE* sont disponibles *via* le module Fcntl. Puisque des systèmes différents supportent des flags différents, ne comptez pas sur la disponibilité de tous les flags sur votre système. Consultez votre page de manuel *open*(2) ou son équivalent local pour plus de détails. Néanmoins, les flags suivants devrait être présents sur tout système disposant d'une bibliothèque C raisonnable :

Flag	Signification
O_RDONLY	Lecture seule.
O_WRONLY	Écriture seule.
O_RDWR	Lecture et écriture.
O_CREAT	Crée le fichier s'il n'existe pas.
O_EXCL	Échoue si le fichier existe déjà.
O_APPEND	Ajoute à la fin du fichier.
O_TRUNC	Tronque le fichier.
O_NONBLOCK	Accès non-bloquant.

Beaucoup d'autres options sont toutefois possibles. Voici quelques flags moins communs :

Flag	Signification
O_NDELAY	Ancien synonyme pour O_NONBLOCK
O_SYNC	Écrit un bloc jusqu'à ce que la donnée soit physiquement écrite vers le matériel sous-jacent. On peut également voir ici O_ASYNC, O_DSYNC et O_RSYNC.
O_EXLOCK	flock avec LOCK_EX (à titre consultatif seulement).
O_SHLOCK	flock avec LOCK_SH (à titre consultatif seulement).
O_DIRECTORY	Échoue si le fichier n'est *pas* un répertoire.
O_NOFOLLOW	Échoue si le dernier composant du chemin est un lien symbolique.
O_BINARY	Effectue un binmode pour les systèmes Microsoft. Un O_TEXT peut également parfois exister pour obtenir le comportement opposé.
O_LARGEFILE	Certains systèmes en ont besoin pour les fichiers de plus de 2 GO.
O_NOCTTY	L'ouverture d'un fichier de terminal ne le fera pas devenir le terminal de contrôle du processus si vous n'en aviez pas encore. Habituellement ce flag n'est plus nécessaire.

Le flag O_EXCL ne sert *pas* pour les verrous : ici, l'exclusivité signifie que si le fichier existe déjà, sysopen échoue. Si le fichier nommé par *NOM_FICHIER* n'existe pas et que le *MODE* inclue le flag O_CREAT, alors sysopen crée le fichier avec les permissions initiales déterminées par l'argument *MASQUE* (ou 0666 s'il est omis) et modifié par le umask courant de votre processus. Cette valeur par défaut est raisonnable : voir la définition de umask pour une explication.

Les handles de fichiers ouverts avec open et sysopen peuvent être utilisés de manière interchangeable. Vous n'avez pas besoin d'utiliser sysread et ses amis simplement parce que vous avez ouvert le fichier avec sysopen, et rien ne vous en empêche si vous l'avez ouvert avec open. Les deux sont capables de choses que l'autre ne peut pas faire. Un open traditionnel peut ouvrir des pipes, forker des processus, positionner des disciplines, dupliquer des handles de fichiers et convertir un numéro descripteur de fichier vers un handle de fichier. De plus, il ignore les caractères d'espacement au début ou à la fin du nom de fichier et respecte la particularité de « - » en tant que nom de fichier spécial. Mais lorsqu'il s'agit d'ouvrir de véritables fichiers, sysopen peut faire tout ce qu'open fait.

Les exemples suivants montrent les appels équivalents à ces deux fonctions. Nous avons omis les vérifications or die $! pour plus de clarté, mais assurez-vous de toujours vérifier les valeurs de retour dans vos programmes. Nous nous restreindrons à n'employer que les flags disponibles sur virtuellement tous les systèmes d'exploitations. Il s'agit simplement d'un moyen de contrôler les valeurs que vous rassemblez par un OU en utilisant l'opérateur de bit | pour les passer dans l'argument *MODE* .

- Ouvrir un fichier en lecture :

```
open(HF, "<", $chemin);
sysopen(HF, $chemin, O_RDONLY);
```

- Ouvrir un fichier en écriture, en créant un nouveau fichier si besoin ou en tronquant un fichier existant :

```
open(HF, ">", $chemin);
sysopen(HF, $chemin, O_WRONLY | O_TRUNC | O_CREAT);
```

- Ouvrir un fichier en ajout, en le créant si besoin :

```
open(HF, ">>", $chemin);
sysopen(HF, $chemin, O_WRONLY | O_APPEND | O_CREAT);
```

- Ouvrir un fichier en mise à jour, le fichier devant déjà exister :

```
open(HF, "+<", $chemin);
sysopen(HF, $chemin, O_RDWR);
```

Et voici ce que vous pouvez faire avec sysopen mais *pas* avec un open traditionnel :

- Ouvrir et créer un fichier en écriture, le fichier ne devant pas déjà exister :

```
sysopen(HF, $chemin, O_WRONLY | O_EXCL | O_CREAT);
```

- Ouvrir un fichier en ajout, le fichier devant déjà exister :

```
sysopen(HF, $chemin, O_WRONLY | O_APPEND);
```

- Ouvrir un fichier en mise à jour, en créant un nouveau fichier si besoin :

```
sysopen(HF, $chemin, O_RDWR | O_CREAT);
```

- Ouvrir un fichier en mise à jour, le fichier ne devant pas déjà exister :

```
sysopen(HF, $chemin, O_RDWR | O_EXCL | O_CREAT);
```

- Ouvrir un fichier en écriture seule sans bloquer, mais sans créer le fichier s'il n'existe pas :

```
sysopen(HF, $chemin, O_WRONLY | O_NONBLOCK);
```

Le module `FileHandle`, décrit au chapitre 32, offre un ensemble de synonymes orientés objet (plus quelques nouvelles fonctionnalités) pour ouvrir des fichiers. Vous êtes cordialement invités à utiliser les méthodes appropriées de `FileHandle`[19] sur tout handle créé avec `open`, `sysopen`, `pipe`, `socket` ou `accept`, même si vous n'avez pas employé le module pour initialiser ces handles.

sysread

$! $@ T X ARG X RO

```
sysread HANDLE_FICHIER, SCALAIRE, LONGUEUR, DECALAGE
sysread HANDLE_FICHIER, SCALAIRE, LONGUEUR
```

Cette fonction tente de lire *LONGUEUR* octets de données dans la variable *SCALAIRE* depuis le *HANDLE_FICHIER* spécifié en utilisant l'appel système de bas niveau *read*(2). La fonction renvoie le nombre d'octets réellement lus, ou 0 en fin de fichier. Elle renvoie `undef` en cas d'erreur. La taille de *SCALAIRE* sera ajustée à la longueur effectivement lue. Le *DECALAGE*, s'il est spécifié, indique à partir d'où lire dans la chaîne, afin de pouvoir lire au milieu d'une chaîne utilisée comme tampon. Voir `syswrite` pour un exemple d'utilisation de *DECALAGE*. Une exception est levée si *LONGUEUR* est négative ou si *DECALAGE* pointe à l'extérieur de la chaîne.

Préparez-vous à gérer vous-même les problèmes (comme les appels système interrompus) que les entrées/sorties standards gèrent habituellement pour vous. Comme `sysread` passe outre les entrées/sorties standards, ne mélangez pas `sysread` avec d'autres type de lectures, ni avec `print`, `printf`, `write`, `seek`, `tell` ou `eof` à moins d'être adepte des sciences occultes (ou des soirées cuir et chaînes). De même, soyez s'il vous plaît prévenus que lors de la lecture dans un fichier contenant de l'Unicode ou tout autre encodage sur plusieurs octets, les frontières du tampon peuvent tomber au milieu d'un caractère.

sysseek

$! X ARG

```
sysseek HANDLE_FICHIER, POSITION, DEPART
```

Cette fonction affecte la position système du *HANDLE_FICHIER* en utilisant l'appel système *lseek*(2). Elle passe outre les entrées/sorties standard, ainsi les mélanges avec des lectures (autres que par `sysread`), ou avec `print`, `printf`, `write`, `seek`, `tell` ou `eof` peuvent semer la confusion. *HANDLE_FICHIER* peut être une expression dont la valeur donne le nom du handle de fichier. Les valeurs de *DEPART* sont `0` pour mettre se positionner exactement à *POSITION*, `1` pour se positionner à la position courante plus *POSITION* et `2` pour se positionner à EOF plus *POSITION* (en général, négative). Pour *DEPART*, vous pouvez utiliser les constantes `SEEK_SET`, `SEEK_CUR` et `SEEK_END` des modules standards `IO::See-`

19. En fait, les méthodes de `IO::File` ou `IO::Handle`.

kable et POSIX — ou, depuis la version 5.6 de Perl, du module Fcntl qui est plus portable et plus pratique.

La fonction renvoie la nouvelle position ou undef en cas d'échec. Une position de zéro est renvoyée comme étant la chaîne spéciale « 0 but true » (*N.d.T.* : 0 mais vrai), qui peut être utilisé numériquement sans produire d'avertissements.

system

$! $? X T

```
system CHEMIN  LISTE
system CHEMIN
```

Cette fonction exécute pour vous tout programme existant sur le système et renvoie le code de retour de ce programme — et non sa sortie. Pour capturer la sortie d'une commande, utilisez les apostrophes inverses ou qx// à la place. La fonction system fonctionne exactement comme exec, hormis le fait qu'elle effectue d'abord un fork puis, qu'après l'exec, elle attende que le programme exécuté se termine. C'est-à-dire qu'elle lance le programme et revient quand ce dernier a fini, au contraire d'exec, qui remplace le programme en train de tourner par le nouveau et qui donc ne revient jamais si le remplacement réussit.

Le traitement des paramètres varie selon le nombre d'arguments, comme dans la description d'exec, y compris le fait de déterminer si le shell sera appelé et si vous avez menti au programme à propos de son nom en spécifiant un *CHEMIN* séparé.

Étant donné que system et les apostrophes inverses bloquent les signaux SIGINT et SIGQUIT, l'envoi de l'un de ces signaux (provenant par exemple d'un Control-C) au programme en train de tourner n'interrompt pas votre programme principal. Mais l'autre programme qui est en train de tourner *reçoit bien* le signal. Vérifiez la valeur de retour de system pour voir si le programme que vous avez lancé s'est terminé proprement ou pas.

```
@args = ("commande", "arg1", "arg2");
system(@args) == 0
    or die "system @args a échoué : $?";
```

La valeur de retour est le statut de sortie du programme tel que renvoyé par l'appel système *wait*(2). Avec les sémantiques traditionnelles, pour obtenir la véritable valeur de sortie, divisez par 256 ou faites un décalage de 8 bits vers la gauche. Ceci car l'octet de poids faible contient quelque chose d'autre. (Deux choses en fait.) Les sept bits de poids faible indiquent le numéro du signal qui a tué le processus (s'il y en a un) et le huitième bit indique si le processus a fait une copie du cœur du programme (*core dump*). Vous pouvez vérifier toutes les possibilités d'échec, y compris les signaux et les *core dumps*, en inspectant $? ($CHILD_ERROR) :

```
$valeur_sortie = $? >> 8;
$num_signal    = $? & 127;   # ou 0x7f, ou 0177, ou 0b0111_1111
$coeur_copie   = $? & 128;   # ou 0x80, ou 0200, ou 0b1000_0000
```

Lorsque le programme a été lancé à travers un shell[20] car vous aviez un seul argument

20. C'est-à-dire */bin/sh* par définition, ou tout ce qui a un sens sur votre plate-forme, mais pas le shell que l'utilisateur utilise à ce moment.

et que cet argument contenait des métacaractères du shell, les codes de retours normaux sont sujets aux caprices et aptitudes supplémentaires du shell. En d'autres termes, dans ces circonstances, vous pouvez être incapable de retrouver l'information détaillée décrite précédemment.

syswrite

$! $@ X ARG

```
syswrite HANDLE_FICHIER, SCALAIRE, LONGUEUR, DECALAGE
syswrite HANDLE_FICHIER, SCALAIRE, LONGUEUR
syswrite HANDLE_FICHIER, SCALAIRE
```

Cette fonction tente d'écrire *LONGUEUR* octets de données depuis la variable *SCALAIRE* vers le *HANDLE_FICHIER* spécifié en utilisant l'appel système *write*(2). La fonction renvoie le nombre d'octets réellement écrits, ou `undef` en cas d'erreur. Le *DECALAGE*, s'il est spécifié, indique à partir de quel endroit de la chaîne il faut commencer à écrire. (Si par exemple la chaîne vous sert par exemple de tampon, ou qu'il vous faut reprendre une écriture partielle.) Un *DECALAGE* négatif spécifie que l'écriture doit partir de ce nombre d'octets depuis la fin de la chaîne. Si *SCALAIRE* est vide, le seul *DECALAGE* permis est 0. Une exception est levée si *LONGUEUR* est négative ou si *DECALAGE* pointe à l'extérieur de la chaîne.

Pour copier des données du handle de fichier *DEPUIS* vers le handle de fichier *VERS*, vous pouvez utiliser quelque chose comme :

```
use Errno qw/EINTR/;
$taille_bloc = (stat FROM)[11] ||16384;   # taille de bloc préférée ?
while ($lng = sysread DEPUIS, $tamp, $taille_bloc) {
    if (!defined $lng) {
        next if $! == EINTR;
        die "Erreur système sur read : $!\n";
    }
    $decalage = 0;
    while ($lng) {     # Gère les écritures partielles.
        $ecrit = syswrite VERS, $tamp, $lng, $decalage;
        die "Erreur système sur write : $!\n" unless defined $ecrit;
        $decalage += $ecrit;
        $lng -= $ecrit;
    }
}
```

Préparez-vous à gérer par vous-même les problèmes que les entrées/sorties standards traitent habituellement pour vous, telles que les écritures partielles. Comme `syswrite` passe outre les entrées/sorties standards de la bibliothèque C, ne mélangez pas d'appels à `syswrite` avec des lectures (autres que `sysread`), des écritures (comme `print`, `printf` ou `write`) ou d'autres opérations d'entrées/sorties comme `seek`, `tell` ou `eof` à moins d'être adepte des sciences occultes.[21]

21. Ou des soirées cuir et chaînes.

tell

X ARG

```
tell HANDLE_FICHIER
tell
```

Cette fonction renvoie la position actuelle dans le fichier (en octets, sur une base de zéro) pour *HANDLE_FICHIER*. Généralement, cette valeur est passée à la fonction `seek` à un moment ultérieur pour revenir à la position courante. *HANDLE_FICHIER* peut être une expression donnant le nom du handle de fichier réel ou une référence sur un objet handle de fichier. Si *HANDLE_FICHIER* est omis, la fonction renvoie la position dans le dernier fichier lu. Les positions ne sont significatives que sur les fichiers ordinaires. Les périphériques, les pipes et les sockets ne connaissent pas les positions.

Il n'existe pas de fonction `systell`. Utilisez `sysseek(HF, 0, 1)` pour cela. Voir `seek` pour un exemple expliquant comment utiliser `tell`.

telldir

X ARG X U

```
telldir HANDLE_REPERTOIRE
```

Cette fonction renvoie la position courante des routines `readdir` sur *HANDLE_REPERTOIRE*. Cette valeur peut être fournie à `seekdir` pour accéder à un endroit particulier d'un répertoire. La fonction comporte les mêmes dangers relatifs à un possible compactage du répertoire que la routine de la bibliothèque système correspondante. Cette fonction peut ne pas être implémentée partout où l'est `readdir`. Même si c'est le cas, aucun calcul ne peut être fait avec la valeur de retour. Il s'agit seulement d'une valeur opaque, qui n'a de signification que pour `seekdir`.

tie

$@

```
tie VARIABLE, NOM_CLASSE, LISTE
```

Cette fonction relie une variable à une classe de paquetage qui fournit l'implémentation de la variable. *VARIABLE* est la variable (scalaire, tableau ou hachage) ou le type glob (représentant un handle de fichier) qui doit être relié. *NOM_CLASSE* est le nom d'une classe implémentant des objets du type approprié.

Tous les arguments additionnels sont passés à la méthode constructeur approprié, soit l'un des constructeurs parmi `TIESCALAR`, `TIEARRAY` ou `TIEHASH`. (Si la méthode appropriée n'est pas trouvée, une exception est levée.) Ces arguments sont généralement similaires à ceux que l'on enverrait à la fonction *dbm_open*(3) du C, mais leur signification dépend du paquetage. L'objet renvoyé par le constructeur est à son tour renvoyé par la fonction `tie`, ce qui s'avère utile si vous voulez accéder à d'autres méthodes de *NOM_CLASSE*. (On peut aussi accéder à l'objet par le biais de la fonction `tied`). Ainsi, une classe reliant un hachage à une implémentation ISAM peut fournir une méthode supplémentaire pour traverser séquentiellement un ensemble de clefs (le « S » de ISAM), puisque vos implémentations habituelles de DBM ne peuvent pas le faire.

Les fonctions comme `keys` et `values` peuvent renvoyer d'énormes valeurs de listes quand elles sont appliquées à de grands objets comme des fichiers DBM. Vous pouvez préférer utiliser la fonction `each` pour parcourir ces derniers. Par exemple :

```
use NDBM_File;
tie %ALIAS, "NDBM_File", "/etc/aliases", 1, 0
    or die "Impossible d'ouvrir les alias : $!\n";
while (($clef,$val) = each %ALIAS) {
    print $clef, ' = ', $val, "\n";
}
untie %ALIAS;
```

Une classe implémentant un hachage doit comprendre les méthodes suivantes :

```
TIEHASH CLASSE, LISTE
FETCH SOI, CLEF
STORE SOI, CLEF, VALEUR
DELETE SOI, CLEF
CLEAR SOI
EXISTS SOI, CLEF
FIRSTKEY SOI
NEXTKEY SOI, CLEF_PRECEDENTE
DESTROY SOI
```

Une classe implémentant un tableau ordinaire doit comprendre les méthodes suivantes :

```
TIEARRAY CLASSE, LISTE
FETCH SOI, INDICE
STORE SOI, INDICE, VALEUR
FETCHSIZE SOI
STORESIZE SOI, COMPTEUR
CLEAR SOI
PUSH SOI, LISTE
POP SOI
SHIFT SOI
UNSHIFT SOI  LISTE
SPLICE SOI, DECALAGE, LONGUEUR, LISTE
EXTEND SOI, COMPTEUR
DESTROY SOI
```

Une classe implémentant un scalaire doit comprendre les méthodes suivantes :

```
TIESCALAR CLASSE, LISTE
FETCH SOI
STORE SOI, VALEUR
DESTROY SOI
```

Une classe implémentant un handle de fichier doit comprendre les méthodes suivantes :

```
TIEHANDLE CLASSE, LISTE
READ SOI, SCALAIRE, LONGUEUR, DECALAGE
READLINE SOI
GETC SOI
WRITE SOI, SCALAIRE, LONGUEUR, DECALAGE
PRINT SOI, LISTE
PRINTF SOI, FORMAT, LISTE
CLOSE SOI
DESTROY SOI
```

Les méthodes indiquées ci-dessus n'ont pas toutes besoin d'être implémentées: Les modules `Tie::Hash`, `Tie::Array`, `Tie::Scalar` et `Tie::Handle` fournissent des classes de bases qui ont des méthodes par défaut raisonnables. Voir chapitre 14, *Variables attachées*, pour un commentaire détaillé de ces méthodes. À l'inverse de `dbmopen`, la fonction `tie` ne fera pas automatiquement un `use` ou un `require` — vous devez le faire vous-même explicitement. Voir les modules `DB_File` et `Config` qui fournissent des implémentations intéressantes de `tie`.

tied

```
tied VARIABLE
```

Cette fonction renvoie une référence vers l'objet qui sous-tend le scalaire, le tableau, le hachage ou le typeglob contenu dans *`VARIABLE`* (la même valeur qui a été renvoyée à l'origine par l'appel à `tie`, qui a lié la variable au paquetage). Elle renvoie la valeur indéfinie si *`VARIABLE`* n'est pas attachée à un paquetage. Vous pouvez par exemple utiliser :

```
ref tied %hachage
```

pour savoir à quel paquetage votre hachage est attaché. (À supposer que vous l'ayez oublié).

time

```
time
```

Cette fonction renvoie le nombre de secondes écoulées depuis le 1er janvier 1970, UTC, à 00::00::00.[22] La valeur renvoyée est acceptée par `gmtime` et `localtime` pour servir aux comparaisons avec les dates de modification et d'accès renvoyées par `stat` et pour être passée à `utime`.

```
$debut = time(); system("une commande lente");
$fin = time();
if ($fin - $debut > 1) {
    print "Le programme a commencé à : ", scalar localtime($debut), "\n";
    print "Le programme a fini à :     ", scalar localtime($fin), "\n";
}
```

times

```
times
```

Dans un contexte de liste, cette fonction renvoie un tableau à quatre éléments donnant les temps utilisateur et système, en secondes (probablement fractionnaires), pour ce processus et pour les fils de ce processus qui sont terminés.

```
($utilisateur, $systeme, $cutilisateur, $csysteme) = times();
printf "Ce pid et ses fils ont consommé %.3f secondes\n",
            $utilisateur + $systeme + $cutilisateur + $csysteme;
```

22. Que l'on appelle aussi « epoch », origine — à ne pas confondre avec l'« épopée », qui se rapporte à la conception d'UNIX. (D'autres systèmes d'exploitation ont une origine différente, pour ne pas parler de leur épopée.)

Dans un contexte scalaire, elle ne renvoie que le temps utilisateur. Par exemple, pour mesurer la vitesse d'exécution d'une section de code Perl :

```
$debut = times();
...
$fin = times();
printf "Nous avons mis %.2f secondes CPU en temps utilisateur\n",
        $fin - $debut;
```

tr///


```
tr///
y///
```

Il s'agit de l'opérateur de traduction (appelé aussi translation), qui ressemble à l'opérateur `y///` du programme *sed* d'UNIX, en mieux, selon l'humble opinion de tout le monde. Voir le chapitre 5.

truncate

```
truncate HANDLE_FICHIER, LONGUEUR
truncate EXPR, LONGUEUR
```

Cette fonction tronque le fichier ouvert sur *HANDLE_FICHIER*, ou nommé par *EXPR*, à la longueur spécifiée. La fonction lève une exception si *truncate*(2) ou un équivalent n'est pas implémenté sur le système. (Il est toujours possible de tronquer un fichier en n'en recopiant que le début, si vous disposez de l'espace disque nécessaire.) La fonction renvoie vrai en cas de succès, `undef` sinon.

uc


```
uc EXPR
uc
```

Cette fonction renvoie une version en majuscules de *EXPR*. Il s'agit de la fonction interne implémentant l'échappement `\U` dans les chaînes entre guillemets. Perl essayera de faire de qu'il faut pour respecter les valeurs actuelles de votre locale, mais nous travaillons toujours sur les interactions avec Unicode. Voir la page de manuel *perllocale* pour les dernières avancées. Dans tous les cas, lorsque Perl utilise les tables Unicode, `uc` convertit en majuscules plutôt qu'en capitales de titres. Voir `ucfirst` pour les conversions en capitales de titres.

ucfirst


```
ucfirst EXPR
ucfirst
```

Cette fonction renvoie une version de *EXPR* avec la première lettre en majuscule (*title-cased*, mise en capitale de titre dans le jargon Unicode). Il s'agit de la fonction interne implémentant l'échappement `\u` dans les chaînes entre guillemets. Votre locale `LC_CTYPE` actuelle peut être respectée si vous faites un `use locale` et si vos données ne ressemblent pas à de l'Unicode, mais nous ne garantissons rien pour le moment.

Pour forcer la première lettre en capitale et tout le reste en minuscules, utilisez :

```
ucfirst lc $mot
```

ce qui est équivalent à `"\u\L$mot"`.

umask

```
umask EXPR
umask
```

Cette fonction fixe le *umask* (masque utilisateur) du processus et renvoie l'ancien en utilisant l'appel système *umask*(2). Votre *umask* indique au système d'exploitation quels bits de permissions *désactiver* au moment de créer un nouveau fichier, y compris les fichiers qui sont des répertoires. Si *EXPR* est omise, la fonction se contente de renvoyer le *umask* courant. Par exemple, pour s'assurer que les bits « utilisateur » sont activés et que les bits « autres » sont désactivés, essayez quelque chose comme :

```
umask((umask() & 077) | 7);   # ne change pas les bits "groupe"
```

Rappelez-vous qu'un *umask* est un nombre, habituellement donné en octal ; ce n'est *pas* une chaîne de chiffres octaux. Voir aussi `oct`, si tout ce que vous avez est une chaîne. Rappelez-vous également que les bits d'*umask* sont le complément des permissions ordinaires.

Les permissions Unix `rwxr-x---` sont représentées comme trois ensembles de trois bits, ou trois chiffres octaux : `0750` (le 0 du début indique un octal et ne compte pas comme un des chiffres). Puisque les bits d'*umask* sont complémentaires, ils représentent des bits de permissions désactivées. Les valeurs de permission (ou « mode ») que vous fournissez à `mkdir` ou `sysopen` sont modifiés par votre *umask*, ainsi même si vous dites à `sysopen` de créer un fichier avec les permissions `0777`, si votre *umask* vaut `0022`, le fichier est créé avec les permissions `0755`. Si votre *umask* valait `0027` (le groupe ne peux pas écrire, les autres ne peuvent ni lire, ni écrire, ni exécuter), alors passer à sysopen un *MASQUE* de `0666` créerait un fichier avec un mode de `0640` (puisque `0666 & ~0027` vaut `0640`).

Voici quelques conseils : donnez un mode de création valant `0666` pour les fichiers normaux (pour `sysopen`) et un de `0777` à la fois pour les répertoires (pour `mkdir`) et pour les fichiers exécutables. Cela donne aux utilisateurs la liberté de choisir : s'ils veulent des fichiers protégés, ils choisiront des *umasks* de processus de `022`, `027` ou même le masque particulièrement antisocial de `077`. Les programmes ne doivent que rarement, si ce n'est jamais, prendre des décisions stratégiques qu'il vaut mieux laisser à l'utilisateur. Les exceptions à cette règle sont les programmes qui écrivent des fichiers qui doivent rester privés : fichiers mails, cookies de navigateurs web, fichiers *.rhosts*, et ainsi de suite.

Si *umask*(2) n'est pas implémenté sur votre système et que vous essayez de restreindre votre *propre* accès (c'est-à-dire, si `(EXPR & 0700) > 0`), vous déclencherez une exception à l'exécution. Si *umask*(2) n'est pas implémenté sur votre système et que vous n'essayez pas de restreindre votre propre accès, la fonction renverra simplement `undef`.

undef

```
undef EXPR
undef
```

undef est le nom sous lequel nous nous référons à l'abstraction connue sous le nom de *la valeur indéfinie*. Il se trouve aussi qu'il s'agit d'une fonction bien nommée qui renvoie toujours la valeur indéfinie. Nous mélangeons les deux avec joie.

Par coïncidence, la fonction undef rend explicitement indéfinie une entité si vous passer son nom comme argument. L'argument *EXPR*, s'il est spécifié, soit être une lvalue. À partir de là, vous ne pouvez utilisez qu'une valeur scalaire, un tableau complet, un hachage complet, un nom de sous-programme (en utilisant le préfixe &) ou un typeglob. Toute la mémoire associée à l'objet sera récupérée (mais pas renvoyée au système, pour la plupart des système d'exploitation). La fonction undef ne fera probablement pas ce que vous attendez d'elle sur la plupart des variables spéciales. L'utiliser sur des variables en lecture seule, comme $1, lève une exception.

La fonction undef est un opérateur unaire, non un opérateur de liste, vous ne pouvez donc rendre indéfini qu'une seule chose à la fois. Voici quelques exemples d'utilisation d'undef en tant qu'opérateur unaire :

```
undef $bidule;
undef $machin{'truc'};    # Différent de delete $machin{'truc'};
undef @tableau
undef %hachage;
undef &monsp;
undef *xyz                # Détruit $xyz, @xyz, %xyz, &xyz, etc.
```

Sans son argument, undef n'est utilisé que pour sa valeur :

```
select(undef, undef, undef, $sieste);

return (wantarray ? () : undef) if $dans_les_choux;
return if $dans_les_choux;  # même chose
```

Vous pouvez utiliser undef comme un emplacement dans la partie gauche d'une affectation de liste, auquel cas la valeur correspondante de la partie droite est simplement rejetée. À part cela, vous ne pouvez pas utiliser undef en tant que lvalue.

```
($a, $b, undef, $c) = &truc;   # Ignore la troisième valeur renvoyée
```

De plus, n'essayez pas de comparer quoique ce soit avec undef — ça ne fait pas ce que vous pensez. Tout ce que ça fait, c'est comparer avec 0 ou la chaîne vide. Utilisez la fonction defined pour déterminer si une valeur est définie.

unlink

$_ $! x

```
unlink LISTE
unlink
```

Cette fonction supprime une liste de fichiers.[23] La fonction renvoie le nombre de fichiers effectivement supprimés. Quelques exemples simples :

23. En fait, sous un système de fichier POSIX, elle supprime les entrées de répertoire (fichiers) qui se réfèrent aux véritables fichiers. Comme un fichier peut être référencé (lié) depuis plus d'un répertoire, le fichier lui-même n'est pas détruit avant que la dernière référence pointant sur lui ne soit effacée.

```
$cpt = unlink 'a', 'b', 'c';
unlink @partants;
unlink glob("*.orig");
```

La fonction `unlink` ne supprime pas les répertoires à moins d'être super-utilisateur et que l'option de la ligne de commande **`-U`** soit donnée à Perl. Même dans ces conditions, vous êtes prévenus que l'effacement d'un répertoire est susceptible d'infliger des dommages sévères dans votre système de fichiers. Employez plutôt `rmdir`.

Voici une commande *rm* très simple avec une vérification d'erreurs qui ne l'est pas moins :

```
#!/usr/bin/perl
@impossible = grep {not unlink}, @ARGV;
die "$0: unlink de @impossible impossible\n" if @impossible;
```

unpack

$@

```
unpack CANEVAS, EXPR
```

Cette fonction fait l'inverse de `pack` : elle développe une chaîne (*EXPR*) représentant une structure de données et l'éclate selon le *CANEVAS* en une liste de valeurs qu'elle renvoie. Dans un contexte scalaire, elle peut être employée pour dépaqueter une seule valeur. Le *CANEVAS* est ici à peu près du même format que pour la fonction `pack` - il spécifie l'ordre et le type des valeurs à dépaqueter. Voir `pack` pour une description plus détaillée de *CANEVAS*. Un élément invalide dans le *CANEVAS* ou une tentative déplacement à l'extérieur de la chaîne avec les formats `x`, `X` ou `@`, lève une exception.

La chaîne est découpée en morceaux décrits dans le *CANEVAS*. Chaque morceau est converti séparément en une valeur. Généralement, les octets de la chaîne sont le résultat d'un `pack` ou ils représentent une structure C d'un certain type.

Si le compteur de répétitions d'un champ est plus grand que ce que le reste dans la chaîne en entrée le permet, le compteur de répétitions est décrémenté en douce. (Normalement, vous devriez de toute façon utiliser un compteur de répétitions valant ici *.) Si la chaîne en entrée est plus grande que ce que décrit le *CANEVAS*, le reste de la chaîne est ignoré.

La fonction `unpack` est également très utile pour les données de texte brut, pas seulement pour les données binaires. Supposez que vous ayez un fichier de données qui contiendrait des enregistrements ressemblant à ceci :

```
2000 Ingrid Caven                  Jean-Jacques Schuhl
1999 Première ligne                Jean-Marie Laclavetine
1998 Confidence pour confidence    Paule Constant
1997 La bataille                   Patrick Rambaud
1996 Le chasseur Zéro              Roze Pascale
1995 Le testament français         André Makine
1994 Un aller simple               Didier Van Cauwelaert
1993 Le rocher de Tanios           Amin Maalouf
1992 Texaco                        Patrick Chamoiseau
```

vous ne pouvez pas utiliser `split` pour décomposer les champs car ils n'ont pas de séparateur distinct. À la place, les champs sont déterminés par leur décalage d'octets dans

l'enregistrement. Ainsi même s'il s'agit d'un enregistrement de texte ordinaire, puisqu'il est composé dans un format fixe, vous pouvez utiliser `unpack` pour le décomposer :

```
while (<>) {
    ($annee, $titre, $auteur) = unpack("A4 x A26 A*", $_);
    print "$auteur a gagné le prix Goncourt $annee pour $titre\n";
}
```

Voici un programme *uudecode* complet :

```
#!/usr/bin/perl
$_ = <>
until ($mode,$fichier) = /^begin\s*(\d*)\s*(\S*)/;
open(SORTIE,"> $fichier") if $fichier ne "";
while (<>) {
    last if /^end/;
    next if /[a-z]/;
    next unless int((((ord() - 32) & 077) + 2) / 3) ==
                int(length() / 4);
    print SORTIE unpack "u", $_;
}
chmod oct $mode, $fichier;
```

En plus des champs autorisés dans `pack`, vous pouvez préfixer un champ par *%nombre* pour produire une somme de contrôle des entités sur *nombre* bits au lieu des entités elles-mêmes. La valeur par défaut est une somme de contrôle de 16 bits. La somme de contrôle est calculée en additionnant les valeurs numériques des valeurs développées (pour les champs de chaînes, on additionne `ord($car)` et pour les champs de bits, on additionne les zéros et les uns). Par exemple, ce qui suit calcule le même nombre que le programme *sum*(1) de System V :

```
undef $/;
$somme_controle = unpack ("%32C*", <>) % 65536;
```

L'exemple suivant compte de façon efficace le nombre de bits positionnés dans un vecteur de bits :

```
$bits_a_1 = unpack "%32b*", $masque_de_select;
```

Voici un décodeur simple MIME (BASE 64) :

```
while (<>) {
    tr#A-Za-z0-9+/##cd;                # enlever les caractères non-base 64
    tr#A-Za-z0-9+/# -_#;               # convertir au format uuencodé
    $lng = pack("c", 32 + 0.75*length); # calculer l'octet de longueur
    print unpack("u", $len . $_);      # uudécoder et afficher
}
```

unshift

```
unshift TABLEAU, LISTE
```

Cette fonction fait l'inverse de `shift`. (Ou l'inverse de `push`, selon le point de vue.) Elle ajoute une *LISTE* au début du tableau et renvoie le nombre actualisé d'éléments dans le tableau :

```
unshift @ARGV, '-e', $cmd unless $ARGV[0] =~ /^-/;
```

Remarquez que la *LISTE* est ajoutée en entier au début, et non un élément à la fois, les éléments ajoutés au début restent donc dans le même ordre. Utilisez `reverse` pour les inverser.

untie

```
untie VARIABLE
```

Brise le lien existant entre une variable ou un typeglob contenu dans *VARIABLE* et le paquetage auquel elle était liée. Voir `tie` et l'intégralité du chapitre 14, mais tout particulièrement la section *Un piège subtil du déliement*.

use $! $@

```
use MODULE VERSION LISTE
use MODULE VERSION ()
use MODULE VERSION
use MODULE LISTE
use MODULE ()
use MODULE
use VERSION
```

La déclaration `use` charge un module, s'il n'a pas déjà été chargé auparavant, et importe depuis ce module des sous-programmes et des variables dans le paquetage courant. (En langage technique, elle importe depuis le module donné des sémantiques dans votre paquetage.) La plupart des déclarations `use` ressemblent à :

```
use MODULE LISTE ;
```

Ce qui est exactement équivalent à :

```
BEGIN { require MODULE ; import MODULE LISTE ; }
```

Le *BEGIN* force le `require` et le `import` à se produire au moment de la compilation. Le `require` s'assure que le module est chargé en mémoire s'il ne l'est pas déjà. L'`import` n'est pas une fonction interne — c'est simplement un appel ordinaire de méthode dans le package nommé par *MODULE* pour demander à ce module de mettre la liste des fonctionnalités dans le package courant. Le module peut implémenter sa méthode d'importation de la manière qui lui plaît, bien que la plupart choisissent simplement de dériver leur méthode d'importation par héritage de la classe Exporter qui est définie dans le module `Exporter`. Voir le chapitre 11, *Modules* et le module `Exporter` pour plus d'informations. Si aucune méthode `import` ne peut être trouvée, l'appel est abandonné sans un mot.

Si vous ne voulez pas altérer votre espace de nom, fournissez explicitement une liste vide :

```
use MODULE ();
```

Ce qui est exactement équivalent à :

```
BEGIN { require MODULE ; }
```

Si le premier argument de `use` est un numéro de version comme 5.6.2, la version courante de Perl en train de s'exécuter doit être au moins aussi récente que la version spé-

cifiée. Si la version actuelle de Perl est inférieur à *VERSION*, un message d'erreur est affiché et l'interpréteur Perl se termine immédiatement. Ceci est très utile pour vérifier la version courante de Perl avant de charger des modules de bibliothèque qui dépendent des nouvelles versions, puisque de temps en temps, nous avons besoin de « casser » les mauvaises fonctionnalités des anciennes versions de Perl. (Nous n'essayons pas de casser plus de choses que nécessaire. En fait nous essayons souvent de casser moins de choses que nécessaire.)

Et puisque nous parlons de ne pas casser certaines choses, Perl accepte encore les anciens numéros de versions de la forme :

```
use 5.005_03;
```

Toutefois, pour mieux s'aligner avec les standards de l'industrie, Perl 5.6 accepte désormais (et nous préférons voir) la syntaxe sous forme d'un triplet :

```
use 5.6.0;     # Soit, version 5, sous-version 6, correctif 0.
```

Si l'argument *VERSION* est présent après *MODULE*, alors le `use` appellera la méthode `VERSION` de la classe `MODULE` avec la *VERSION* donnée comme argument. Remarquez qu'il n'y pas de virgule après *VERSION* ! La méthode `VERSION` par défaut, qui est héritée de la classe `UNIVERSAL`, meurt en gémissant si la version donnée est plus élevée que la valeur de la variable `$Module::VERSION`.

Voir le chapitre 32 pour une liste des modules standards.

Comme `use` fournit une interface largement ouverte, les pragmas (directives de compilation) sont également implémentés *via* des modules. Les pragmas actuellement implémentées comprennent :

```
use autouse 'Carp' => qw(carp croak);
use bytes;
use constant PI => 4 * atan2(1,1);
use diagnostics;
use integer;
use lib '/opt/projets/spectre/lib';
use locale;
use sigtrap qw (die INT QUIT);
use strict qw(subs vars refs);
use warnings "deprecated";
```

La plupart des ces modules pragmatiques importent des définitions dans la portée lexicale courante. (Au contraire des modules ordinaires, qui n'importent des symboles que dans le paquetage courant, qui a une rapport minime avec la portée lexicale courante autre que le fait que la portée lexicale est compilée avec ce paquetage à l'esprit. Ceci pour dire que. . . oh et puis zut, voyez le chapitre 11.)

Il existe une déclaration correspondante, `no`, qui « désimporte » toutes les significations importées à l'origine avec `use` qui sont devenues depuis, heu, sans importance :

```
no integer;
no strict 'refs';
no utf8;
no warnings "unsafe";
```

Voir le chapitre 31 pour une liste des pragmas standards.

utime

$! X T X U

```
utime LISTE
```

Cette fonction change les dates d'accès et de modification sur chacun des fichiers de la liste. Les deux premiers éléments de la liste doivent être les dates d'accès et de modification *en numérique*, dans cet ordre. La fonction renvoie le nombre de fichiers changés avec succès. La date de modification d'inode de chaque fichier est fixée à la date courante. Voici un exemple d'une commande *touch* qui met la date de modification du fichier (en supposant que vous en êtes le propriétaire) à environ un mois dans le futur :

```
#!/usr/bin/perl
# touch_mois - antidate les fichiers à maintenant + 1 mois
$jour = 24 * 60 * 60;              # 24 heures en secondes
$plus_tard = time() + 30 * $jour;  # 30 jours, c'est environ un mois
utime $plus_tard, $plus_tard, @ARGV;
```

et voici un exemple un peu plus sophistiqué d'une commande qui fait un *touch* avec quelques notions de contrôle d'erreurs :

```
#!/usr/bin/perl
# touch_mois - antidate les fichiers à maintenant + 1 mois
$plus_tard = time() + 30 * 24 * 60 * 60;
@impossibles = grep {not utime $plus_tard, $plus_tard, $_} @ARGV;
die "touch impossible de @impossibles.\n" if @impossibles;
```

Pour lire les dates de fichiers existants, utilisez `stat` et passez les champs appropriés à `localtime` ou `gmtime` pour l'affichage.

values

```
values HACHAGE
```

Cette fonction renvoie une liste constituée de toutes les valeurs du *HACHAGE* indiqué. Les valeurs sont renvoyées dans un ordre apparemment aléatoire, mais c'est le même que celui produit par les fonctions `keys` ou `each` pour le même hachage. Curieusement, pour trier un hachage par ses valeurs, vous aurez besoin habituellement de la fonction `keys`, regardez donc l'exemple dans la description de `keys`.

Vous pouvez modifier les valeurs d'un hachage en utilisant cette fonction car la liste renvoyée contient des alias des valeurs et pas simplement des copies. (Dans les versions précédentes, vous aviez besoin d'utiliser pour ce faire une portion de hachage.)

```
for (@hachage{keys %hachage}) { s/machin/truc/g } # ancienne manière
for (values %hachage)         { s/machin/truc/g } # modifie maintenant
                                                  # les valeurs
```

L'emploi de `values` sur un hachage lié à un fichier DBM important est voué à produire une liste toute aussi volumineuse, ce qui vous donnera un processus important. Il peut être préférable pour vous d'utiliser dans ce cas la fonction `each`, qui parcourra les entrées de hachage une à une, sans les absorber en une liste gargantuesque.

vec

vec *EXPR, DECALAGE, BITS*

La fonction vec fournit un stockage compact de listes d'entiers non-signés. Ces entiers sont empaquetés et compressés autant que possible dans une chaîne Perl ordinaire. La chaîne dans *EXPR* est traitée comme une chaîne de bits construite à partir d'un nombre arbitraire d'éléments selon la longueur de la chaîne.

DECALAGE spécifie l'indice de l'élément particulier qui vous intéresse. Les syntaxes pour lire et écrire l'élément sont les mêmes, puisque vec stocke la valeur de l'élément selon que vous utilisiez un contexte lvalue ou rvalue.

BITS spécifie l'étendue d'un élément en bits, ce qui doit être exprimé en puissance de 2 : 1, 2, 4, 8, 16 ou 32 (et également 64 sur certaines plates-formes). (Une exception est levée si une quelconque autre valeur est utilisée.) Chaque élément peut ainsi contenir un entier compris dans l'intervalle `0..(2**BITS)-1`. Pour les plus petites tailles, on empaquète dans chaque octet autant d'éléments que possible. Lorsque *BITS* vaut 1, il y a huit éléments par octets. Lorsque *BITS* vaut 2, il y a quatre éléments par octets. Lorsque *BITS* vaut 4, il y a deux éléments (habituellement appelés des quartets) par octets. Et ainsi de suite. Les entiers plus grands qu'un octet sont stockés dans l'ordre gros-boutise (*N.d.T.* : *big-endian*).

Une liste d'entiers non-signés peut être stockée dans une simple variable scalaire en les assignant individuellement à la fonction vec. (Si *EXPR* n'est pas une lvalue valide, une exception est levée.) Dans l'exemple suivant les éléments font tous 4 bits de large :

```
$chaine_bits = "";
$decalage = 0;

foreach $nb (0, 5, 5, 6, 2, 7, 12, 6) {
    vec($chaine_bits, $decalage++, 4) = $num;
}
```

Si un élément est écrit après la fin de la chaîne, Perl étendra d'abord la chaîne avec suffisamment d'octets à zéro.

Les vecteurs stockés dans la variable scalaire peuvent par la suite être atteints en spécifiant le *DECALAGE* correct :

```
$nb_elements = length($chaine_bits)*2;    # 2 éléments par octet

foreach $decalage (0 .. nb_elements-1) {
    print vec($chaine_bits, $decalage, 4), "\n";
}
```

Si l'élément sélectionné se situe après la fin de la chaîne, une valeur de 0 est renvoyée.

Les chaînes créées avec vec peuvent également être manipulées avec les opérateurs logiques |, &, ^ et ~. Ces opérateurs postulent qu'une opération de vecteurs de bits est désirée quand les deux opérandes sont des chaînes.Voir des exemples de ceci au chapitre 3, *Opérateurs unaires et binaires*, dans la section *Opérateurs sur les bits*.

Si *BITS* == 1, une chaîne de bits peut être créée pour stocker une série de bits dans un seul scalaire. L'ordre est tel que l'on garanti que `vec($chaine_bits,0,1)` part du bit le

plus faible du premier octet de la chaîne.

```
@bits = (0,0,1,0, 1,0,1,0, 1,1,0,0, 0,0,1,0);

$chaine_bits = "";
$decalage = 0;

foreach $bit (@bits) {
    vec($chaine_bits, $decalage++, 1) = $bit;
}
print "$chaine_bit\n";           # "TC", i.e. '0x54', '0x43'
```

Une chaîne de bits peut être convertie vers ou depuis une chaîne de `1` et de `0` en donnant un canevas de « `b*` » à `pack` ou `unpack`. Dans l'autre sens, `pack` peut être utilisé avec un canevas de « `b*` » pour créer la chaîne de bits depuis une chaîne de `1` et de 0. L'ordre est compatible avec celui que `vec` attend.

```
chaine_bits = pack "b*", join('', @bits);
print "$chaine_bits\n";          # "TC" comme précédemment
```

`unpack` peut être utilisé pour extraire la liste de `0` et de `1` de la chaîne de bits.

```
@bits = split (//, unpack("b*", $chaine_bits));
print "@bits\n";                 # 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0
```

Si vous connaissez la longueur exacte en bits, celle-ci peut être utilisée en lieu et place de « * ».

Voir `select` pour des exemples supplémentaires d'utilisation de bitmaps générées avec `vec`. Voir `pack` et `unpack` pour des manipulations de données binaires à un niveau supérieur.

wait

$! $? x u

```
wait
```

Cette fonction attend qu'un processus fils se termine et renvoie le PID du processus décédé, ou `-1` s'il n'y a pas de processus fils (ou sur certains systèmes, si les processus fils sont automatiquement enterrés). Le statut est renvoyé dans `$?` comme nous l'avons décrit pour la fonction `system`. Si vous obtenez des processus fils zombies, vous devrez appeler cette fonction, ou `waitpid`.

S'il n'y a pas de fils, contrairement à votre attente, il s'est probablement produit un appel à `system`, une fermeture de pipe ou des apostrophes inverses entre le `fork` et le `wait`. Ces constructions provoquent également une attente avec *wait*(2) et peuvent avoir récolté votre processus fils. Utilisez `waitpid` pour éviter ce problème.

waitpid

$! $? x u

```
waitpid PID, FLAGS
```

Cette fonction attend qu'un certain processus fils se termine et renvoie le PID quand le processus est mort, `-1` s'il n'y a pas de fils, ou `0` si les *`FLAGS`* spécifient un appel non-bloquant et que le processus n'est pas encore mort. Le statut du processus mort est renvoyé dans `$?` comme nous l'avons décrit pour la fonction `system`. Pour obtenir les valeurs va-

lides des flags, vous devrez importer le tag de groupe d'importation `":sys_wait.h"` depuis le module `POSIX`. Voici un exemple qui réalise une attente non-bloquante des processus zombies :

```
use POSIX ":queue";
do {
    $fils = waitpid(-1, &WNOHANG);
} until $fils == -1;
```

Sur les systèmes qui n'implémentent pas les appels système *waitpid*(2) et *wait4*(2), les *FLAGS* ne peuvent être spécifiés qu'à `0`. En d'autres termes, vous pouvez attendre là un *PID* spécifique, mais vous ne pouvez le faire en mode non-bloquant.

Sur certains systèmes, une valeur de retour de `-1` peut signifier que les processus fils sont automatiquement enterrés car vous avez fait `$SIG{CHLD} = 'IGNORE'`.

wantarray

```
wantarray
```

Cette fonction renvoie vrai si le contexte du sous-programme actuellement en train de s'exécuter demande une valeur tableau et faux sinon. La fonction renvoie une valeur définie fausse (`""`) si le contexte appelant demande un scalaire et la valeur indéfinie fausse (`undef`) si le contexte appelant ne demande rien de particulier, c'est-à-dire s'il s'agit d'un contexte vide.

Voici quelques exemples d'usages typiques :

```
return unless defined wantarray;    # ne se préoccupe pas d'en faire plus
my @a = calcul_complique();
return wantarray ? @a : \@a;
```

Voir également `caller`. Cette fonction aurait vraiment dû s'appeler « `wantlist` » mais nous l'avons baptisée alors que les contextes de listes s'appelaient encore contextes de tableaux.

warn $!

```
warn LISTE
warn
```

Cette fonction produit un message d'erreur en affichant *LISTE* sur `STDERR` exactement comme `die`, mais sans essayer de sortir ou de lancer une exception. Par exemple :

```
warn "Débogage actif" if $debug;
```

Si *LISTE* est vide et que `$@` contient déjà une valeur (généralement issue d'un `eval` précédent), la chaîne `"\t...caught"` est ajoutée à la fin de `$@` sur `STDERR`. (De manière similaire à celle dont `die` propage les erreurs, excepté le fait que `warn` ne propage (lève à nouveau) pas l'exception.) Si la chaîne de message fournie est vide, le message « `Warning: Something's wrong` »[24] est utilisé.

24. *N.d.T.* : Littéralement : « `Avertissement : Il y a quelque chose de mauvais` ».

Si la chaîne de message fournie ne se termine pas par un saut de ligne, les informations concernant le fichier et le numéro de ligne sont automatiquement ajoutées à la fin. La fonction `warn` n'a aucun rapport avec l'option de la ligne de commande de Perl `-w`, mais elle peut être utilisée en conjonction avec cette dernière, comme lorsque vous voulez émuler le comportement interne :

```
warn "Quelque chose de foncièrement mauvais\n" if $^W;
```

Aucun message n'est affiché si un gestionnaire pour `$SIG{__WARN__}` est installé. Il appartient au gestionnaire d'en décider si le message lui semble approprié. Une chose que vous pourriez vouloir faire est de promouvoir un simple avertissement en exception :

```
local $SIG{__WARN__} = sub {
    my $msg = shift;
    die $msg if $msg =~ /n'est pas numérique/;
}
```

La plupart des gestionnaires doivent donc faire prendre leurs dispositions pour afficher les avertissements qu'ils n'ont pas prévu de gérer, en appelant à nouveau `warn` dans le gestionnaire. Ceci est tout à fait sûr ; il n'y aura pas de boucles sans fin car les routines `__WARN__` ne sont pas appelées depuis des routines `__WARN__`. Ce comportement diffère un peu de celui des gestionnaires de `$SIG{__DIE__}` (qui ne suppriment pas le texte d'erreur mais qui peuvent à la place appeler une nouvelle fois `die` pour le changer).

L'emploi d'un gestionnaire `__WARN__` donne un moyen particulièrement puissant de taire tous les avertissements, même ceux qui sont pourtant qualifiés d'obligatoires. Parfois, vos aurez besoin d'encapsuler cela dans un bloc `BEGIN{}` pour que ce soit pris en compte dès la compilation :

```
# nettoie *tous* les avertissements à la compilation
BEGIN { $SIG{__WARN__} = sub { warn $_[0] if $AVERTISSEMENTS_ACTIFS } }
my $truc = 10;
my $truc = 20;                # pas d'avertissement à propos des my $truc
                              # dupliqués mais, bon, vous l'avez voulu !

# pas d'avertissements à la compilation ou à l'exécution jusqu'ici;
$AVERTISSEMENTS_ACTIFS = 1;   # avertissements à l'exécution activés
                              #à partir d'ici
warn "\$truc est vivant et vaut $truc !"; # dénonce tout
```

Voir le pragma `use warnings` pour le contrôle de la portée lexicale des avertissements. Voir les fonctions `carp` et `cluck` du module `Carp` pour d'autres moyens de produire des messages d'avertissement.

write

$! $@ X ARG

```
write HANDLE_FICHIER
write
```

Cette fonction écrit un enregistrement formaté (qui peut être multilignes) vers le handle de fichier spécifié, en utilisant le format associé à ce handle de fichier — voir la section *Variables de format* au chapitre 7. Par défaut, le format d'un fichier est celui qui a le même nom que le handle de fichier. Toutefois, le format d'un handle de fichier peut être modifié en changeant la variable `$~` après avoir fait un `select` sur ce handle :

```
$ancien_hf = select(HANDLE);
$~ = "NOUVEAU_NOM";
select($ancien_hf);
```

ou en écrivant :

```
use IO::Handle;
HANDLE->format_name("NOUVEAU_NOM");
```

Puisque les formats sont mis dans l'espace de nom d'un paquetage, il se peut que vous deviez qualifier le nom du format si le `format` a été déclaré dans un paquetage différent :

```
$~ = "Autre_Paquetage::NOUVEAU_NOM";
```

Le traitement de l'en-tête de formulaire est automatiquement géré : s'il n'y a plus assez de place sur la page courante pour l'enregistrement formaté, la page est avancée en écrivant un saut de page, un format de haut de page spécial est utilisé pour formater le nouvel en-tête de page, et l'enregistrement est enfin écrit. Le nombre de lignes restant sur la page courante se trouve dans la variable `$-`, qui peut être mise à 0 pour forcer qu'une nouvelle page soit prise au prochain `write`. (Vous devrez peut être `select`ionner d'abord le handle de fichier.) Par défaut, le nom du format de haut de page est celui du handle de fichier auquel est ajouté « `_TOP` », mais il peut être modifié en changeant la variable `$^` après avoir fait un `select` sur ce handle ou en écrivant :

```
use IO::Handle;
HANDLE->format_top_name("NOUVEAU_NOM_TOP");
```

Si *`HANDLE_FICHIER`* n'est pas spécifié, ce qui est écrit sort sur le handle de fichier de sortie par défaut, qui est `STDOUT` au départ mais peut être changé par la forme à un seul argument de l'opérateur `select`. Si *`HANDLE_FICHIER`* est une expression, celle-ci est évaluée pour déterminer le véritable *`HANDLE_FICHIER`* à l'exécution.

Si un `format` spécifié ou le `format` de l'en-tête de page courant n'existe pas, une exception est levée.

La fonction `write` n'est *pas* l'inverse de `read`. Utilisez `print` pour une simple écriture de chaîne. Si vous avez consulté cette entrée parce que vous voulez court-circuiter les entrées/sorties standard, regardez `syswrite`.

y///

```
y///
```

Il s'agit de l'opérateur de traduction (appelé aussi translation), également connu sous le nom de `tr///`. Voir le chapitre 5.

30

La bibliothèque standard Perl

La distribution standard de Perl contient bien plus que l'exécutable *perl* qui exécute vos scripts. Elle inclue également des centaines de modules remplis de code réutilisable. Comme les modules standards sont disponibles partout, si vous utilisez l'un d'entre eux dans votre programme, vous pouvez faire tourner votre programme partout où Perl est installé, sans étapes d'installation supplémentaires.

Bibliothèquologie

Avant d'énumérer ces modules dans les prochains chapitres, passons quelque peu en revue la terminologie dans laquelle nous nageons jusqu'au cou.

espace de noms

Les noms sont conservés dans un *espace de noms* afin de ne pas les confondre avec des noms identiques dans des espaces différents. Ainsi reste à ne pas confondre les espaces de noms entre eux. Il existe deux manières d'éviter cela : leur donner des noms uniques ou leur attribuer des emplacements uniques. Perl vous laisse faire les deux : les espaces de noms nommés s'appellent des *paquetages* et les espaces de noms anonymes sont appelés *portées lexicales*. Puisque les portées lexicales ne peuvent pas s'étendre au-delà d'un fichier et puisque les modules standards sont (au minimum) de la taille d'un fichier, il s'ensuit que toutes les interfaces de modules doivent employer des espaces de noms nommés (des paquetages) si quelqu'un doit les utiliser depuis l'extérieur du fichier.

paquetage

Un *paquetage* est un mécanisme standard de Perl pour déclarer un espace de noms nommé. Il s'agit d'un mécanisme très simple pour rassembler des fonctions et des variables qui ont des points communs. Tout comme deux répertoires peuvent contenir tous deux un fichier (différent) s'appelant *fred*, deux parties d'un programme Perl peuvent, chacune de leur côté, avoir leur propre variable `$fred` ou leur propre fonction `&fred`. Même si ces variables ou ces fonctions semblent avoir le même nom, ces noms résident dans des espaces de noms distincts gérés par la déclaration

`package`. Les noms de paquetages sont utilisés pour identifier à la fois des modules et des classes, comme nous l'avons décrit au chapitre 11, *Modules* et au chapitre 12, *Objets*.

bibliothèque

Le terme *bibliothèque* (*library*) est malheureusement suremployé dans la culture Perl. De nos jours, nous utilisons normalement le terme pour désigner l'ensemble des modules Perl installés sur votre système.

Historiquement, une bibliothèque Perl était également un simple fichier contenant une collection de sous-programmes partageant un objectif commun. Un tel fichier est souvent suffixé par *.pl*, pour « perl library ». Nous employons toujours cette extension pour des morceaux variés de code Perl que vous incorporez avec `do` *`FICHIER`* ou `require`. Bien qu'il ne s'agisse pas d'un module à part entière, un fichier de bibliothèque se déclare généralement lui-même comme étant dans un paquetage distinct, ainsi les variables et les sous-programmes qu'il contient peuvent être rassemblées et n'interfèrent pas accidentellement avec d'autres variables dans votre programme. Il n'y a pas d'extension de fichier obligatoire ; en outre, on rencontre souvent d'autres *.pl* comme nous le verrons plus loin dans ce chapitre. Ces fichiers de bibliothèques simples et sans structure ont été largement supplantés par les modules.

module

Un *module* Perl est un fichier de bibliothèque conforme à certaines conventions spécifiques qui autorisent un ou plusieurs fichiers implémentant ce module à être rentrés avec une seule déclaration `use` à la compilation. Les noms de fichier des modules doivent toujours se terminer par *.pm*, car la déclaration `use` le présuppose. La déclaration `use` convertira également le séparateur de paquetage `::` vers votre séparateur de répertoire quel qu'il soit afin que la structure des répertoires dans votre bibliothèque Perl puisse refléter la structure de votre paquetage. Le chapitre 11 décrit comment créer vos propres modules Perl.

classe

Une *classe* est juste un module qui implémente des méthodes pour les objets associés au nom du paquetage du module. Si les modules orientés objet vous intéressent, voyez le chapitre 12.

pragma

Un *pragma* est juste un module spécial qui fait joujou avec les boutons internes de Perl. Voir le chapitre 31, *Modules de pragmas*.

extension

Une *extension* est un module Perl qui, en plus de charger un fichier *.pm*, charge également un bibliothèque partagée implémentant en C ou en C++ la sémantique du module.

programme

Un *programme* Perl est un code conçu pour tourner en tant qu'entité indépendante ; il est également connu sous le nom de *script* lorsque vous ne voulez pas que quiconque en attende un peu trop, sous le nom d'*application* lorsqu'il est

volumineux et compliqué, sous le nom d'*exécutable* lorsque celui qui l'appelle ne se soucie pas du langage dans lequel il a été écrit, ou encore sous le nom de *solution industrielle* lorsqu'il coûte une fortune. Les programmes Perl existent en tant que code source, bytecode ou code machine natif. S'il s'agit de quelque chose que vous pouvez lancer depuis la ligne de commande, nous l'appellerons un *programme*.

Visite guidée de la bibliothèque Perl

Vous gagnerez énormément de temps si vous faites l'effort de vous familiariser avec la bibliothèque Perl standard, car il n'y a aucune raison de réinventer la roue. Nous devons toutefois vous avertir que cette collection contient un vaste champ de matières premières. Bien que certaines bibliothèques soit extrêmement utiles, d'autres peuvent être sans aucun rapport avec vos besoins. Par exemple, si vous n'écrivez que du code Perl pur à 100 %, les modules qui supportent le chargement dynamique d'extensions C et C++ ne vont pas beaucoup vous aider.

Perl s'attend à trouver les modules de bibliothèque quelque part dans son chemin d'inclusion (*N.d.T.* : « *include* ») de bibliothèque, `@INC`. Ce tableau spécifie la liste ordonnée des répertoires où Perl cherche lorsque vous chargez du code de bibliothèque en utilisant les mots clefs `do`, `require` ou `use`. Vous pouvez facilement afficher la liste de ces répertoires en appelant Perl avec l'option **`-V`**, pour Version Vraiment Verbeuse, ou avec ce simple code :

```
% perl -le "print foreach @INC"
/usr/libdata/perl5/sparc-openbsd/5.00503
/usr/local/libdata/perl5/sparc-openbsd/5.00503
/usr/libdata/perl5
/usr/local/libdata/perl5
/usr/local/libdata/perl5/site_perl/sparc-openbsd
/usr/libdata/perl5/site_perl/sparc-openbsd
/usr/local/libdata/perl5/site_perl
/usr/libdata/perl5/site_perl
.
```

Il ne s'agit que d'un exemple d'affichage possible. Chaque installation de Perl utilise ses propres chemins. Le point important est que, malgré la variation des contenus selon la politique de votre constructeur et de celle de l'installation sur votre site, vous pouvez compter sur toutes les bibliothèques standards qui sont installées avec Perl. Si vous voulez retrouver l'endroit depuis lequel un fichier a été réellement chargé, consultez la variable `%INC`. Pour un fichier de module, vous pouvez trouver exactement où Perl l'a trouvé avec cette commande :

```
% perldoc -l MODULE
```

Si vous regardez les répertoires de `@INC` et leurs sous-répertoires, vous trouverez différentes sortes de fichiers installés. La plupart ont des noms se terminant par *.pm*, mais certains se finissent par *.pl*, *.ph*, *.al* ou *.so*. Ceux qui vous intéressent le plus sont dans la première catégorie car un suffixe *.pm* indique que le fichier est un module Perl authentique. Nous en dirons plus sur ceux-là dans un instant.

Le peu de fichiers que vous verrez là finissant par *.pl* sont ces vieilles bibliothèques Perl que nous avons mentionnées plus haut. Ils sont inclus pour garder une compatibilité

avec d'anciennes versions de Perl datant des années 80 et du début des années 90. Ainsi, le code Perl qui marchait, disons en 1990, doit continuer à bien se comporter sans histoires même si vous avez installé une version moderne de Perl. Lorsque vous êtes en train d'écrire du code nouveau, vous devez toujours préférer, lorsque c'est possible, utiliser la version avec *.pm* plutôt que celle avec *.pl*. Ainsi, les modules ne pollueront pas votre espace de noms comme beaucoup des vieux fichiers *.pl* le font.

Une remarque à propos de l'emploi de l'extension *.pl* : elle signifie *Perl library* (bibliothèque Perl), et non pas programme Perl. Bien que l'extension *.pl* soit utilisée pour identifier les programmes Perl sur les serveurs web qui ont besoin de distinguer, dans le même répertoire, les programmes exécutables des contenus statiques, nous vous suggérons d'employer un suffixe de *.plx* à la place pour indiquer un programme Perl exécutable. (Le même conseil vaut pour les systèmes d'exploitation qui choisissent les interpréteurs en se basant sur les extensions des noms de fichiers.)

Les fichiers avec des extensions *.al* sont de petits morceaux de modules plus grands qui sont automatiquement chargés lorsque vous utiliser leur fichier parent *.pm*. Si vous construisez le plan de votre module en utilisant l'outil standard *h2xs* qui est fourni avec Perl (et si vous n'avez pas employer le flag Perl **-A**), la procédure `make install` utilisera le module `Autoloader` pour créer pour vous ces petits fichiers *.al*.

Les fichiers *.ph* sont construits par le programme *h2ph*, un outil quelque peu vieillissant mais parfois toujours utile pour convertir des directives de préprocesseur C vers du Perl. Les fichiers *.ph* résultants contiennent des constantes parfois nécessaires aux fonctions de bas niveau comme `ioctl`, `fnctl` ou `syscall`. (De nos jours la plupart de ces valeurs sont disponibles, plus facilement et de manière plus portable, dans des modules standards comme `POSIX`, `Errno`, `Fcntl` ou `Sockets`.) Voir *perlinstall* pour savoir comment installer ces composants optionnels mais parfois importants.

La dernière extension que vous pourriez rencontrer en errant ici et là est *.so* (ou ce que votre système emploie pour les bibliothèques partagées). Ces fichiers *.so* sont des portions de modules d'extension qui sont dépendantes de la plate-forme. Elles sont écrites en C ou en C++ et il faut les compiler pour obtenir le code objet relogeable dynamiquement. L'utilisateur final n'a de toute façon pas besoin d'être au courant de leur existence car elles sont masquées par l'interface du module. Lorsque le code de l'utilisateur dit `require Module` ou `use Module`, Perl charge *Module.pm* et l'exécute, ce qui laisse le module attirer à lui toute autre partie nécessaire, comme *Module.so* ou tous les composant *.al* autochargés. En fait, le module peut charger tout ce qui lui plaît, y compris 582 autres modules. Il peut télécharger l'intégralité de CPAN s'il le veut et pourquoi pas les archives des deux dernières années de *freshmeat.net*.

Un module n'est pas simplement un morceau de code statique en Perl. C'est un agent actif qui arrive à comprendre comment implémenter pour vous une interface. Il peut suivre toutes les conventions standards, ou pas. Il lui est permis de tout faire pour pervertir la signification du reste de votre programme, y compris jusqu'à convertir le reste de votre programme en SPITBOL. Ce genre de canular est considéré comme étant parfaitement loyal pour autant qu'il soit bien documenté. Lorsque vous utilisez un tel module Perl, vous acceptez *son* contrat, non un contrat standard écrit par Perl.

Vous feriez donc mieux de lire les petits caractères de la notice.

31

Modules de pragmas

Un *pragma* est un type particulier de module qui affecte la phase de compilation de votre programme. Certains modules de pragmas (ou *pragmata*, pour faire court, ou *pragmas*, pour faire encore plus court) peuvent également affecter la phase d'exécution de votre programme. Voyez-les comme des indications données au compilateur. Comme ils ont besoin d'être visibles à la compilation, ils ne fonctionnent que lorsqu'ils sont invoqués par un `use` ou un `no`, car lorsqu'un `require` ou un `do` s'exécute, la compilation est finie depuis longtemps.

Par convention, les noms de pragmas sont écrit en minuscules car les noms de modules en minuscules sont réservés pour la distribution Perl elle-même. Lorsque vous écrivez vos propres modules, utilisez au moins une lettre majuscule pour éviter d'entrer en conflit avec les noms de pragmas.

Au contraire des modules ordinaires, la plupart des pragmas limitent leurs effets au reste du bloc le plus interne encadrant l'endroit d'où ils ont été appelés. En d'autres termes, ils sont limités à une portée lexicale, exactement comme les variables `my`. Habituellement, la portée lexicale d'un bloc extérieur couvre tout bloc intérieur inclus dans le premier, mais un bloc intérieur peut contredire un pragma de portée lexicale provenant d'un bloc extérieur avec l'instruction `no` :

```
use strict;
use integer;
{
    no strict 'refs';      # permet les références symboliques
    no integer;            # reprend l'arithmétique en virgule flottante
    # ...
}
```

Plus encore que les autres modules inclus dans la distribution Perl, les pragmas forment une partie intégrale et essentielle de l'environnement de compilation en Perl. Il est difficile de bien utiliser le compilateur si vous ne savez pas comment lui passer des indications, nous produirons donc un effort supplémentaire à décrire les pragmas.

Un autre point auquel il faut prêter attention est le fait que nous utilisons souvent les pragmas pour prototyper des fonctionnalités qui seront plus tard encodées dans une syntaxe « réelle ». Ainsi vous verrez dans quelques programmes des pragmas dépréciés comme `use attrs` dont la fonctionnalité est maintenant directement supportée par la syntaxe de déclaration des sous-programmes. De même, `use vars` est en cours de remplacement par les déclarations `our`. Et `use subs` peut un jour être remplacé par un attribut `override` dans les déclarations de sous-programmes ordinaires. Nous ne sommes pas extrêmement pressés de casser les anciennes pratiques mais nous pensons assurément que les nouvelles pratiques sont plus jolies.

use attributes

```
sub une_fonct : method;
my $fermeture = sub : method { };

use attributes;
@liste_attrs = attributes::get(\&une_fonct);
```

Le pragma `attributes` a deux objectifs. Le premier est de fournir un mécanisme interne pour déclarer *des listes d'attributs*, qui sont des propriétés optionnelles associées aux déclarations de sous-programmes et (dans un futur proche) aux déclarations de variables. (Puisqu'il s'agit d'un mécanisme interne, vous ne devriez pas en général utiliser ce pragma directement.) Le second objectif est de fournir un moyen de récupérer ces listes d'attributs à l'exécution en appelant la fonction `attribute::get`. Dans cette fonctionnalité, `attributes` est juste un module standard et non un pragma.

Perl ne gère actuellement qu'un petit nombre d'attributs en interne. Les attributs spécifiques à un paquetage sont autorisés par le biais d'un mécanisme d'extension expérimental décrit dans la section *Package-specific Attribute Handling* (*N.d.T.* : gestion d'attributs spécifiques à un paquetage) de la page de man *attributes*(3).

Le positionnement des attributs intervient à la compilation ; la tentative de positionner un attribut inconnu provoque une erreur de compilation. (L'erreur peut être intercepée avec `eval`, mais elle continuera d'arrêter la compilation à l'intérieur du bloc `eval`.)

Il n'existe actuellement que trois attributs internes implémentés pour les sous-programmes : `locked`, `method` et `lvalue`. Voir le chapitre 6, *Sous-programmes* et le chapitre 7, *Threads*, pour une discussion approfondie sur ces attributs. Il n'existe pas actuellement d'attributs internes pour les variables comme il y en a pour les sous-programmes mais nous en avons plusieurs en tête que nous aimons bien, comme `constant`.

Le pragma `attributes` fournit deux sous-programmes pour un usage générique. Ils peuvent être importés si vous les demandez :

get

Cette fonction renvoie une liste (qui peut être vide) des attributs lorsqu'on lui donne un seul paramètre en entrée, lequel est une référence à un sous-programme ou à une variable. La fonction lève une exception en invoquant `Carp::croak` si on lui passe des arguments invalides.

reftype

Cette fonction se comporte plus ou moins comme la fonction interne `ref` mais elle renvoie toujours le type de donnée sous-jacent intrinsèque à Perl ou la valeur référencée, en ignorant tout paquetage dans lequel elle pourrait avoir été consacrée.

Des détails plus précis sur la gestion des attributs sont toujours en cours de discussion, vous feriez donc mieux de consulter la documentation en ligne de votre distribution de Perl pour savoir quel est l'état en cours.

use autouse

```
use autouse 'Carp' => qw(carp croak);
carp "Cet appel à carp a été prédéclaré avec autouse\n";
```

Ce pragma fournit un mécanisme pour une demande de chargement à l'exécution d'un module particulier si et seulement si une fonction de ce module est réellement appelée. Il accomplit ceci en donnant une souche (*stub*) de fonction qui est remplacée par l'appel réel lorsqu'on la déclenche. Dans l'esprit, ce mécanisme est similaire au comportement des modules standards `AutoLoader` et `SelfLoader`. Pour faire court, c'est une astuce de performance qui facilite le démarrage plus rapide (en moyenne) de votre programme Perl en évitant la compilation de modules qui pourraient ne jamais être appelés durant une exécution donnée du programme.

Le comportement de `autouse` est lié au fait que le module soit déjà chargé ou non. Par exemple, si le module `Module` est déjà chargé, alors la déclaration :

```
use autouse 'Module' => qw(fonc1 fonc2($;$) Module::fonc3);
```

est équivalente à la simple importation de deux fonctions :

```
use Module qw(fonc1 fonc2);
```

Cela suppose que `Module` définisse `fonc2` avec le prototype `($;$)` et que `fonc1()` et `fonc3()` n'aient pas de prototype. (Plus généralement, cela suppose également que `Module` utilise la méthode `import` du module standard `Exporter` ; sinon, une exception est levée.) Dans tous les cas, la déclaration `Module::fonc3` est ignorée puisqu'elle est certainement déjà déclarée.

Si, d'un autre côté, `Module` n'a pas encore été chargé lorsque le pragma `autouse` est analysé, ce dernier déclare les fonctions `fonc1` et `fonc2` dans le paquetage courant. Il déclare également une fonction `Module::fonc3` (ce que l'on pourrait analyser comme étant légèrement anti-social, si ce n'est que l'absence du module `Module` aurait des conséquence encore plus anti-sociales). Lorsque ces fonctions sont appelées, elles s'assurent que le `Module` en question est chargé et elles sont alors remplacées par les appels aux fonctions réelles qui viennent d'être chargées.

Puisque le pragma `autouse` déplace des portions de votre programme depuis la compilation vers l'exécution, cela peut avoir des ramifications déplaisantes. Par exemple, si le module sur lequel vous avez appliqué `autouse` possède une phase d'initialisation qui s'attend à être passée au plus tôt, celle-ci pourra ne pas arriver assez tôt. L'emploi de `autouse` peut également masquer des bogues dans votre code lorsque d'importantes vérifications sont déplacées de la compilation vers l'exécution.

En particulier, si le prototype que vous avez spécifié dans la ligne `autouse` est faux, vous

ne le découvrirez pas avant que la fonction correspondante soit exécutée (ce qui peut se produire des mois ou des années plus tard, pour une fonction rarement appelée). Pour résoudre partiellement ce problème, vous pouvez écrire votre code ainsi durant votre phase de développement :

```
use Chasse;
use autouse Chasse => qw(tolle($) clameur($&));
clameur "Cette clameur a été prédéclarée avec autouse\n";
```

La première ligne s'assure que les erreurs dans votre spécification des arguments soient trouvées assez tôt. Lorsque votre programme passe de la phase de développement à la phase de production, vous pouvez mettre en commentaire le chargement ordinaire du module `Chasse` et laisser seulement l'appel à `autouse`. De cette manière, vous obtenez de la sécurité pendant le développement et des performances durant la production.

use base

```
use base qw(Mere Pere);
```

Ce pragma permet à un programmeur de déclarer aisément une classe dérivée basée sur les classes parentes listées. La déclaration ci-dessus est à peu près équivalente à :

```
BEGIN {
    require Mere;
    require Pere;
    push @ISA, qw(Mere Pere)
}
```

Le pragma `use base` s'occupe de chaque `require` nécessaire. Lorsque vous êtes dans la portée du pragma `strict 'vars'`, `use base` vous laisse faire (effectivement) un assignement de `@ISA` sans avoir à déclarer d'abord `our @ISA`. (Puisque le pragma `use base` intervient lors de la compilation, il vaut mieux éviter de titiller `@ISA` dans votre coin lors de l'exécution.)

Mais au-delà de ceci, `use base` possède une autre propriété. Si l'une des classes de base nommées utilise les facilités de `use fields` décrites plus loin dans ce chapitre, alors le pragma initialise les attributs spéciaux de champs du paquetage depuis la classe de base. (L'héritage multiple des classes de champs n'est *pas* supporté. Le pragma `use base` lève une exception si plus d'une des classes de base nommées possède des champs.)

Toute classe de base qui n'a pas encore été chargée l'est automatiquement *via* `require`. Toutefois, le fait que l'on doive ou non charger un paquetage d'une classe de base avec `require` est déterminé, non pas par l'inspection habituelle de `%INC`, mais par l'absence d'une variable globale `$VERSION` dans le paquetage de base. Cette astuce évite à Perl d'essayer en boucle (et d'échouer) de charger une classe de base qui ne se trouve pas dans le fichier que l'on appelle par `require` (car, par exemple, elle est chargée en tant que faisant partie du fichier d'un autre module). Si `$VERSION` n'est pas détectée après avoir réussi à charger le fichier, `use base` définira `$VERSION` dans le paquetage de base, en la positionnant à la chaîne « `-1, defined by base.pm` ».

use blib

Depuis la ligne de commande :

```
% perl -Mblib programme [args...]
% perl -Mblib=REP  programme [args...]
```

Depuis votre programme Perl :

```
use blib;
use blib 'REP ';
```

Ce pragma est en premier lieu un moyen de tester arbitrairement les programmes Perl pour découvrir une version non-installée d'un paquetage grâce à l'option de la ligne de commande de Perl `-M`. Il suppose que la structure de vos répertoires a été produite par le module standard `ExtUtils::MakeMaker`.

Le pragma cherche une structure de répertoire *blib* commençant depuis un répertoire nommé *REP* (ou le répertoire courant si l'on n'en a spécifié aucun) et s'il n'y trouve pas de répertoire *blib*, il revient en arrière à travers vos répertoires « `..` », en remontant jusqu'au cinquième niveau dans les répertoires parents.

use bytes

```
use bytes;
no bytes;
```

Le pragma `use bytes` désactive la sémantique des caractères pour le reste de la portée lexicale dans lequel il apparaît. Le pragma `no bytes` peut être employé pour inverser les effets de `use bytes` à l'intérieur de la portée lexicale courante.

Perl présume normalement une sémantique de caractères en présence d'une donnée sous forme de caractères (c'est-à-dire, une donnée notée comme étant dans un encodage de caractère particulier).

Pour comprendre les implications et les différences entres les sémantiques de caractères et les sémantiques d'octets, voir le chapitre 15, *Unicode*. Une visite à Tokyo peut également être utile.

use charnames

```
use charnames COMMENT;
print "\N{SPEC_CAR} est un drôle de caractère" ;
```

Ce pragma de portée lexicale permet aux caractères nommés d'être interpolés vers des chaînes en utilisant la notation `\N{`*SPEC_CAR*`}` :

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} est appelée sigma.\n";

use charnames ':short';
print "\N{greek:Sigma} est appelée sigma.\n";

use charnames qw(cyrillic greek);
print "\N{sigma} est sigma en grec et \N{be} est b en cyrillique.\n";
```

Le pragma supporte les valeurs `:full` et `:short` pour le paramètre *COMMENT*, ainsi que des noms de scripts spécifiques.[1] L'argument *COMMENT* détermine comment rechercher le caractère spécifié par *SPEC_CAR* dans `\N{`*SPEC_CAR*`}`. Si `:full` est présent, *SPEC_CAR* est d'abord cherché dans les tables de caractères Unicode en tant que nom complet de caractère Unicode. Si `:short` est présent et que *SPEC_CAR* est de la forme *NOM_SCRIPT*`:`*NOM_CAR*, *NOM_CAR* est recherché en tant que lettre dans le script *NOM_SCRIPT*. Si *COMMENT* contient des noms de scripts spécifiques, *SPEC_CAR* est recherché en tant que *NOM_CAR* dans chacun des scripts donnés et dans l'ordre spécifié.

Pour la recherche de *NOM_CAR* dans un script *NOM_SCRIPT* donné, le pragma regarde dans la table des noms standards Unicode s'il peut trouver des motifs ayant la forme de :

NOM_SCRIPT CAPITAL LETTER *NOM_CAR*
NOM_SCRIPT SMALL LETTER *NOM_CAR*
NOM_SCRIPT LETTER *NOM_CAR*

Si *NOM_CAR* est entièrement en minuscules (comme dans `\N{sigma}`), la variante `CAPITAL` est ignorée.

Vous pouvez écrire votre propre module fonctionnant comme le pragma `charnames` mais définissant différemment les noms de caractères. Toutefois, l'interface pour cela est toujours expérimentale, consultez donc son état courant dans la page de man.

use constant

```
use constant TAILLE_TAMPON       => 4096;
use constant UNE_ANNEE           => 365.2425 * 24 * 60 * 60;
use constant PI                  => 4 * atan2 1, 1;
use constant DEBOGAGE            => 0;
use constant ORACLE              => 'oracle@cs.indiana.edu';
use constant NOM_UTILISATEUR     => scalar getpwuid($<);
use constant INFOS_UTILISATEUR   => getpwuid($<);

sub deg_rad { PI * $_[0] / 180 }

print "Cette ligne ne fait rien"      unless DEBOGAGE;
# Les références peuvent être déclarées en tant que constantes
use constant C_HACHAGE           => { truc => 42 };
use constant C_TABLEAU           => [ 1,2,3,4 ];
use constant C_PSEUDO_HACH       => [ { truc => 1 }, 42];
use constant C_CODE              => sub { "mord $_[0]\n" };

print C_HACHAGE->{truc};
print C_TABLEAU->{$i];
print C_PSEUDO_HACH->{truc};
```

1. Nous ne voulons pas dire par là des scripts Perls. Nous voulons dire des « scripts » comme dans certains styles particuliers de lettres écrites, comme le romain ou le grec ou le cyrillique. Malheureusement, « script » est le terme technique pour cela et nous n'avons pas beaucoup de chance de persuader le Consortium Unicode d'adopter un terme différent.

```
print C_CODE->{"moi"};
print C_HACHAGE->[10];          # erreur à la compilation
```

Ce pragma déclare le symbole nommé comme étant une constante immuable[2] avec la valeur scalaire ou de liste donnée. Vous devez faire une déclaration séparée pour chaque symbole. Les valeurs sont évaluées dans un contexte de liste. Vous pouvez outrepasser cela avec `scalar` comme nous l'avons fait ci-dessus.

Puisque les constantes ne commencent pas par un `$`, vous ne pouvez pas les interpoler directement dans les chaînes entre guillemets, bien que ce soit possible indirectement :

```
print "La valeur de PI est @{[ PI ]}.\n";
```

Comme les constantes de liste sont renvoyées en tant que listes, et non en tant que tableaux, vous devez indicer une constante représentant un valeur de liste en utilisant des parenthèses supplémentaires comme vous le feriez avec toute autre expression de liste :

```
$rep_maison = INFO_UTILISATEUR[7];     # FAUX
$rep_maison = (INFO_UTILISATEUR)[7];   # ok
```

Bien que l'on recommande d'écrire les constantes entièrement en majuscules pour les faire ressortir et pour aider à éviter les collisions potentielles avec d'autres mots-clefs et noms de sous-programmes, il ne s'agit que d'une convention. Les noms de constantes doivent commencer par une lettre mais celle-ci n'a pas besoin d'être en majuscule.

Les constantes ne sont pas privées dans la portée lexicale dans laquelle elles apparaissent. Elle sont plutôt de simples sous-programmes sans argument dans la table de symboles du paquetage où a lieu la déclaration. Vous pouvez vous référez à une constante *`CONST`* depuis un paquetage `Autre` en écrivant `Autre::`*`CONST`*. Approfondissez la notion d'insertion en ligne (*N.d.T.* : *inlining*) de tels sous-programmes à la compilation en lisant la section *Substitution en ligne de fonctions constantes* au chapitre 6.

Comme dans toutes les directives `use`, `use constant` intervient à la compilation. Ainsi, placer une déclaration de constante à l'intérieur d'une instruction conditionnelle comme `if ($truc) { use constant ... }` est une incompréhension totale.

Si l'on omet la valeur pour un symbole, ce dernier prendra la valeur `undef` dans un contexte scalaire ou la liste vide dans un contexte de liste. Mais il vaut probablement mieux déclarer ceux-ci explicitement :

```
use constant CAMELIDE        => ();
use constant MAISON_CHAMEAU  => undef;
```

Restrictions de use constant

Les listes constantes ne sont pas actuellement insérées en ligne de la même manière que les scalaires. Et il n'est pas possible d'avoir un sous-programme ou un mot clef du même nom qu'une constante. C'est probablement une Bonne Chose.

Vous ne pouvez pas déclarer plus d'une constante nommée à la fois :

```
use constant TRUC => 4, MACHIN => 5;     # FAUX
```

2. Implémentée comme un sous-programme sans argument et renvoyant toujours la même constante.

Ceci défini une constante nommée TRUC qui renvoie une liste (4, "MACHIN", 5). Vous devez plutôt écrire :

```
use constant TRUC => 4;
use constant MACHIN => 5;
```

Vous allez au-devant des problèmes si vous utilisez une constante dans un contexte qui met automatiquement les mots bruts entre guillemets. (Ceci est pas vrai pour tout appel de sous-programme, pas seulement pour les constantes.) Par exemple vous ne pouvez pas écrire $hachage{*CONSTANTE*} car *CONSTANTE* serait interprétée comme une chaîne. Utilisez $hachage{*CONSTANTE*()} ou $hachage{+*CONSTANTE*} pour empêcher le mécanisme de mise entre guillemets de se mettre en branle. De même, puisque l'opérateur => met entre guillemets son opérande de gauche si cette dernière est un nom brut, vous devez écrire *CONSTANTE*() => 'valeur' au lieu de *CONSTANTE* => 'valeur'.

Un jour, vous serez capables d'utiliser un attribut constant dans les déclarations de variables :

```
my $PI : constant = 4 * atan2(1,1);
```

Ceci possède tous les avantages d'être une variable plutôt qu'un sous-programme, mais tous les inconvénients de ne pas être encore implémenté.

use diagnostics

```
use diagnostics;            # activé à la compilation
use diagnostics -verbose;

enable diagnostics;         # activé à l'exécution
disable diagnostics;        # désactivé à l'exécution
```

Ce pragma étend les diagnostics ordinaires, laconiques et supprime les avertissements dupliqués. Il augmente les versions abrégées avec les descriptions plus explicites et plus agréables que l'on peut trouver au chapitre 33, *Messages de diagnosti*c. Comme les autres pragmas, celui-ci affecte également la phase de compilation de votre programme, pas seulement la phase d'exécution.

Lorsque vous utilisez use diagnostics au début de votre programme, l'option Perl **-w** de la ligne de commande est automatiquement activée en mettant $^W à 1. La suite de votre entière compilation sera alors sujette à des diagnostics améliorés. Ces derniers sont toujours envoyés vers STDERR.

En raison des interactions entre les résultats d'exécution et de compilation, et parce que ce n'est souvent pas une très bonne idée, vous ne devriez pas utiliser no diagnostics pour désactiver les diagnostics à la compilation. Vous pouvez néanmoins contrôler leur comportement à l'exécution en utilisant les méthodes enable et disable, pour respectivement les activer et les désactiver. (Assurez-vous d'avoir d'abord fait le use ou sinon vous serez incapables d'accéder aux méthodes.)

Le flag -verbose affiche d'abord l'introduction de la page man de *perldiag* avant tout autre diagnostic. La variable $diagnostics::PRETTY peut être positionnée (avant le use) pour générer de meilleures séquences d'échappement pour les pagineurs comme *less*(1) ou *more*(1) :

```
BEGIN { $diagnostics::PRETTY = 1 }
use diagnostics;
```

Les avertissements envoyés par Perl et détectés par ce pragma ne sont affichés qu'une seule fois chacun. Ceci est utile lorsque vous êtes pris dans une boucle qui génère encore et toujours le même avertissement (comme une valeur non-initialisée). Les avertissements générés manuellement, comme ceux découlant des appels à `warn` ou `carp`, ne sont pas affectés par ce mécanisme de détection des duplications.

Voici quelques exemples d'utilisation du pragma `diagnostics`. Le fichier suivant déclenche à coup sûr quelques erreurs tant à la compilation qu'à l'exécution :

```
use diagnostics;
print NULLE_PART "rien\n";
print STDERR "\n\nCe message devrait rester tel quel.\n";
warn "\tAvertissement utilisateur";
print "\nTESTEUR DE DIAGNOSTIC : Entrez un <RETOUR_CHARIOT> ici : ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

ce qui affiche :

```
Parenthesis missing around "my" list at diagtest line 6 (#1)

    (W parenthesis) You said something like

        my $foo, $bar = @_;

    when you meant

        my ($foo, $bar) = @_;

Remember that "my", "our", and "local" bind tighter than comma.

Name "main::NULLE_PART" used only once: possible typo at diagtest line 2 (#2)

(W once) Typographical errors often show up as unique variable
    names. If you had a good reason for having a unique name,
    then just mention it again somehow to suppress the message.
    The our declaration is provided for this purpose.

Name "main::b" used only once: possible typo at diagtest line 6 (#2)
Name "main::x" used only once: possible typo at diagtest line 8 (#2)
Name "main::y" used only once: possible typo at diagtest line 8 (#2)

Filehandle main::NULLE_PART never opened at diagtest line 2 (#3)

(W unopened) An I/O operation was attempted on a filehandle that
    was never initialized. You need to do an open() or a socket()
    call, or call a constructor from the FileHandle package.

        Ce message devrait rester tel quel.
```

```
        Avertissement utilisateur at diagtest line 4.

TESTEUR DE DIAGNOSTIC : Entrez un <RETOUR_CHARIOT> ICI :
Use of uninitialized value in division (/) at diagtest line 8 (#4)

(W uninitialized) An undefined value was used as if it were
    already defined. It was interpreted as a "" or a 0, but maybe
    it was a mistake. To suppress this warning assign a defined
    value to your variables.

Illegal division by zero at diagtest line 8 (#5)

(F) You tried to divide a number by 0. Either something was
    wrong in your logic, or you need to put a conditional in to
    guard against meaningless input.

Uncaught exception from user code:
        Illegal division by zero at diagtest line 8.
```

Les messages de diagnostic sont tirés du fichier *perldiag.pod*. Si un gestionnaire `$SIG{__WARN__}` existant est détecté, il sera toujours respecté, mais seulement après que la fonction `diagnostics::splainthis` (l'intercepteur de `$SIG{__WARN__}` pour le pragma) aura fait son travail pour vos avertissements. Perl ne supporte pas à l'heure actuelle les empilements de gestionnaires, c'est donc le mieux que nous puissions faire pour l'instant. Il existe une variable `$diagnostics::DEBUG` que vous pouvez positionner si vous êtes d'incurables curieux à propos des genres de choses qui sont interceptées.

```
BEGIN { $diagnostics::DEBUG = 1 }
use diagnostics;
```

use fields

Dans le module `Animal` :

```
package Animal;
use strict;
use fields qw(nom poids _Animal_id);
my $ID = 0;
sub new {
    my Animal $obj = shift;
    unless (ref $obj) {
        $obj = fields::new($obj);
        $obj->{_Animal_id} = "ceci est l'ID secret de l'Animal";
    }
    $obj->{nom} = "Hé, toi !";
    $obj->{poids} = 20;
    return $obj;
}
1;
```

Dans un programme séparé, *demo_animal* :

```
use Animal;
my Animal $rock = new Animal;     # lexicalement typé

$rock->{nom} = "quartz";
$rock->{poids} = "2kg";
$rock->{_Animal_id} = 1233;       # attribut privé

$rock->{couleur} = "bleu";        # génère une erreur de compilation
```

Dans le module `Chien` :

```
package Chien;
use strict;
use base 'Animal';                # hérite les champs et méthodes d'Animal
use fields qw(nom pedigree);      # surcharge l'attribut nom d'Animal
                                  # et ajoute un nouvel attribut pedigree
use fields qw(remue _Chien_prive) # non-partagé avec Animal
sub new {
    my $classe = shift;
    my $obj = fields::new{$classe};
    $obj_>SUPER::new();           # initialise les champs de base
    $obj->{pedigree} = "aucun";   # initialise ses propres champs
    return $obj;
}
```

Dans un programme séparé, *demo_chien* :

```
use Chien;

my Chien $medor = new Chien       # lexicalement typé
$medor->{nom} = "Theloneus"       # non-hérité
$medor->{poids} = "30lbs";        # hérité
$medor->{pedigree} = "bâtard";    # non-hérité

$medor->{couleur} = 'brun';       # génère une erreur de compilation
$medor->{_Animal_pid} = 3324;     # génère une erreur de compilation
```

Le pragma `fields` fournit une méthode pour déclarer les champs de classe dont le type peut être vérifié à la compilation. Ceci repose sur une fonctionnalité connue sous le nom de pseudo-hachages : si une variable lexicale typée (`my Animal $rock`) contient une référence (l'objet `Animal`) et est utilisée pour accéder à un élément de hachage (`$rock->{nom}`), s'il existe un paquetage du même nom que le type déclaré, et si ce paquetage a positionné des champs de classe en utilisant le pragma `fields`, alors l'opération se change en accès à un tableau lors de la compilation, pourvu que le champ spécifié soit valide.

Le pragma `base` apparenté combinera les champs des classes de base et ceux déclarés avec le pragma `fields`. Ceci rend permet à l'héritage de champ de bien fonctionner.

Les noms de champ qui commencent par un caractère souligné sont rendus privés pour la classe et invisibles dans les sous-classes. Les champs hérités peuvent être surchargés mais généreront un avertissement si les avertissements sont activés.

La conséquence de tout ceci est que vous pouvez avoir des objets avec des champs nommés qui soient aussi compacts que des tableaux avec un accès aussi rapide. Cependant,

ceci ne fonctionne que tant que l'on accède aux objets à travers des variables convenablement typées. Si les variables ne sont pas typées, l'accès est uniquement vérifié à l'exécution, votre programme tourne donc plus lentement car il doit accéder à la fois à un hachage et à un tableau. En plus des déclarations de champs, les fonctions suivantes sont supportées :

`new`

La fonction `fields::new` crée et consacre un pseudo-hachage dans la classe spécifiée (qui peut également être spécifiée en passant un objet de cette classe). L'objet est créé avec les champs déclarés auparavant pour cette classe avec le pragma `fields`. Ceci autorise l'écriture d'un constructeur comme ceci :

```
package Creature::Sons;
use fields qw(chat chien oiseau);

sub new {
    my Creature:Sons $obj = shift;
    $obj = fields::new($obj) unless ref $obj;
    $obj->{chat} = 'miaule';                          # élément scalaire
    @$obj{'chien', 'oiseau'} = ('aboie','siffle');  # tranche
    return $obj;
}
```

`phash`

La fonction `fields::phash` crée et initialise un pseudo-hachage brut (non-consacré). Vous devriez toujours utiliser cette fonction pour créer des pseudo-hachages au lieu de le faire directement, au cas où nous décidions de changer l'implémentation.

Si le premier argument de `phash` est une référence à un tableau, le pseudo-hachage sera créé avec des clefs provenant de ce tableau. Si le second argument est fourni, ce doit être également une référence à un tableau dont les éléments seront utilisés comme valeurs. Si le second tableau contient moins d'éléments que le premier, les éléments restants du pseudo-hachage ne seront pas initialisés. Ceci est particulièrement utile pour créer un pseudo-hachage à partir des arguments d'un sous-programme :

```
sub chien_option {
    my $option = fields::phash([qw(nom grade matricule)], [@_]);
}
```

Vous pouvez aussi passer une liste de paires clef/valeur qui sera utilisée pour construire le pseudo-hachage :

```
my $option = fields::phash(nom => "Joe",
                           grade => "capitaine",
                           matricule => 42);

my $pseudo_hachage = fields::phash(%args);
```

Pour plus d'information sur les pseudo-hachages, voir la section « Pseudo-hachages » au chapitre 8, *Références*.

L'implémentation actuelle conserve les champs déclarés dans le hachage `%FIELDS` du paquetage appelant, mais cela peut changer dans les prochaines versions, il vaut donc mieux vous fier à l'interface de ce pragma pour gérer vos champs.

use filetest

```
$lecture_peut_etre_possible = -r "fichier"     # utilise les bits de mode
{
    use filetest 'access';
    $lecture_vraiment_possible = -r "fichier"  # dépiste plus sérieusement
}
$lecture_peut_etre_possible = -r "fichier';    # utilise à nouveau les
                                               #bits de mode
```

Ce pragma de portée lexicale indique au compilateur de changer le comportement des opérateurs unaires de test de fichier `-r`, `-w`, `-x`, `-R`, `-W` et `-X`, documentés au chapitre 3, *Opérateurs unaires et binaires*. Le comportement par défaut de ces tests de fichiers est d'utiliser les bits de mode renvoyés par les appels de la famille de `stat`. Toutefois, ce n'est peut-être pas toujours la façon correcte de faire, comme lorsqu'un système de fichiers comprends les ACL (*Access Control Lists*, listes de contrôle d'accès). Dans les environnement où cela compte, comme AFS, le pragma `use filetest` peut aider les opérateurs de permissions à renvoyer des résultats plus cohérents avec d'autres outils.

Il peut y avoir une légère baisse de performance pour les opérateurs de test de fichier affectés lorsqu'on les utilise avec `use filetest`, car sur certains systèmes les fonctionnalités étendues ont besoin d'être émulées.

Attention : toute notion d'utilisation de tests de fichiers dans des objectifs de sécurité est une cause perdue d'avance. La porte est grande ouverte pour des situations de concurrence (*race conditions*), car il n'y a aucun moyen de garantir que les permissions ne changeront pas entre le test et l'opération effective. Si vous êtes ne serait-ce qu'un peu sérieux avec la sécurité, vous n'utiliserez pas d'opérateurs de test de fichier pour décider si quelque chose *va* marcher ou pas. À la place, essayez de lancer l'opération réelle puis testez si l'opération a réussi ou non. (Vous devriez de toute façon faire cela.) Voir la section *Gestion des problèmes de synchronisation* au chapitre 23, *Sécurité*.

use filetest 'access'

Actuellement une seule importation, `access`, est implémentée. L'appel de `use filetest 'access'` active l'emploi d'*access*(2) ou d'appels systèmes équivalents lorsqu'on lance des tests de fichier ; symétriquement, `no filetest 'access'` le désactive. Cette fonctionnalité étendue de test de fichier n'est utilisée que lorsque l'opérande de l'opérateur (ou, si vous préférez, l'argument de la fonction unaire) est un véritable nom de fichier, et non un handle de fichier.

use integer

```
use integer;
$x = 10 / 3;
# $x vaut maintenant 3, et non 3.33333333333333333
```

Ce pragma de portée lexicale indique au compilateur d'utiliser des opérations entières à partir de ce point et jusqu'à la fin du bloc l'encadrant. Sur la plupart des machines, cela infuence peu la majorité des calculs mais sur celles qui ne disposent pas de coprocesseur arithmétique, cela peut entraîner une différence de performance dramatique.

Remarquez que ce pragma affecte certaines opérations numériques, pas les nombres eux-même. Par exemple, si vous lancez ce code :

```
use integer;
$x = 1.8;
$y = $x + 1;
$z = -1.8;
```

vous vous retrouverez avec `$x == 1.8`, `$y == 2` et `$z == -1`. Le cas de `$z` s'explique par le fait qu'un `-` unaire compte comme une opération, ainsi la valeur `1.8` est tronquée à un avant que son bit de signe soit inversé. De même, les fonctions qui attendent des nombres à virgule flottante, comme `sqrt` ou les fonctions trigonométriques, recevront et renverront toujours des nombres à virgule même sous `use integer`. Ainsi `sqrt(1.44)` vaut `1.2` mais `0 + sqrt(1.44)` vaut maintenant seulement `1`.

L'arithmétique entière native, telle qu'elle est fournie par votre compilateur C, est utilisée. Cela signifie que la propre sémantique de Perl pour les opérations arithmétiques peut ne pas être préservée. Une source d'ennuis fréquente est le modulo des nombres négatifs. Perl le calcule d'une façon mais votre matériel peut le faire d'une autre manière :

```
% perl -le 'print (4 % -3)'
-2
% perl -Minteger -le 'print (4 % -3)'
1
```

use less

```
use less;              # AUCUN N'EST IMPLÉMENTÉ

use less 'CPU';
use less 'memory';
use less 'time';
use less 'disk';
use less 'fat';        # géant avec "use locale";
```

Actuellement non-implémenté, ce pragma est destiné à donner un jour des indications au compilateur, au générateur de code ou à l'interpréteur pour activer certains équilibrages.

Il n'y a aucun mal à demander à utiliser moins (*use less*) de quelque chose alors que Perl ne sait pas comment réduire ce quelque chose à l'heure actuelle.

use lib

```
use lib "$ENV{HOME}/libperl";    # ajoute ~/libperl
no lib ".";                      # enlève cwd
```

Ce pragma simplifie la manipulation de `@INC` à la compilation. Il est généralement utilisé pour ajouter des répertoires supplémentaires dans le chemin de recherche de Perl pour qu'ensuite les instructions `do`, `require` et `use` trouvent les fichiers de bibliothèque qui ne se situent pas dans le chemin de recherche par défaut de Perl. Il est particulière-

ment important avec use, puisque ce dernier intervient également à la compilation et que l'affectation ordinaire de @INC (c'est-à-dire à l'exécution) arriverait trop tard.

Les paramètres de use lib sont ajoutés au début du chemin de recherche de Perl. Écrire use lib *LISTE* revient *à peu près* au même qu'écrire BEGIN { unshift(@INC, *LISTE*) }, mais use lib *LISTE* inclut le support des répertoires spécifiques à une plate-forme. Pour chaque répertoire $rep donné dans sa liste d'arguments, le pragma list vérifie également si un répertoire nommé *$rep/$nom_architecture/auto* existe ou non. Si c'est le cas, le répertoire *$rep/$nom_architecture/auto* est supposé correspondre à un répertoire spécifique à la plate-forme et il est donc ajouté à @INC (devant $rep).

Pour éviter les ajouts redondants qui ralentissent les temps d'accès et gaspillent une petite quantité de mémoire, les entrées dupliquées qui restent dans @INC sont supprimées lorsque des entrées sont ajoutées.

Normalement, vous ne devez qu'*ajouter* des répertoires dans @INC. Si vous avez vraiment besoin d'enlever des répertoires de @INC, prenez garde à ne supprimer que ceux que vous avez vous-même ajoutés ou ceux dont vous êtes certains qu'ils ne sont pas nécessaires à d'autres modules dans votre programme. D'autres modules peuvent avoir ajouté dans votre tableau @INC des répertoires dont ils avaient besoin pour que leurs opérations soient correctement effectuées.

Le pragma no lib supprime de @INC toutes les instances de chaque répertoire nommé. Il détruit également tous les répertoires correspondants spécifiques à la plate-forme, tels que nous les avons décrits plus haut.

Lorsque le pragma lib est chargé, il sauvegarde la valeur courante de @INC dans le tableau @lib::ORIG_INC, afin de copier ce tableau dans le véritable @INC pour restaurer la valeur originale.

Même si @INC inclut généralement le répertoire courant, point (« . »), Ce n'est vraiment pas aussi utile que vous pouvez le penser. Premièrement, l'entrée point vient à la fin, pas au début. Ainsi les modules installés dans le répertoire courant ne prennent pas soudainement le dessus sur les versions globales au système. Vous pouvez écrire use lib "." si c'est ce que vous voulez vraiment. Plus ennuyeux, il s'agit du répertoire courant du processus Perl, et non du répertoire où le script a été installé, ce qui enlève toute fiabilité à cet ajout de « . ». Si vous créez un programme, plus quelques modules utilisés par ce programme, il marchera tant que vous développez, mais cela cessera lorsque vous ne le lancerez pas depuis le répertoire où se trouvent les fichiers.

Une solution pour cela est d'utiliser le module standard FindBin :

```
use FindBin;              # où le script a-t-il été installé ?
use lib $FindBin::Bin;    # utilise également ce répertoire pour lib
```

Le module FindBin essaie de deviner le chemin complet vers le répertoire où a été installé le programme du processus en train de tourner. N'utilisez pas ceci dans des objectifs de sécurité car les programmes malveillants peuvent d'ordinaire le détourner en essayant suffisamment énergiquement. Mais à moins que vous n'essayez intentionnellement de casser le module, il devrait marcher comme prévu. Le module procure une variable $Findbin::Bin (que vous pouvez importer) qui contient l'emplacement où le programme a été installé, tel que le module l'a deviné. Vous pouvez alors utiliser le pragma lib pour ajouter ce répertoire à votre tableau @INC, produisant ainsi un chemin relatif à l'exécutable.

Certains programmes s'attendent à être installés dans un répertoire *bin* et trouvent alors leurs modules de bibliothèque dans les fichiers « cousins » installés dans un répertoire *lib* au même niveau que *bin*. Par exemple, des programmes peuvent se trouver dans */usr/local/apache/bin* ou */opt/perl/bin* et les bibliothèques dans */usr/local/apache/lib* et */opt/perl/lib*. Ce code en tient soigneusement compte :

```
use FindBin qw(Bin);
use lib "$Bin/../lib";
```

Si vous vous retrouvez à spécifier le même `use lib` dans plusieurs programmes sans rapport les uns avec les autres, vous pouvez envisager de positionner à la place la variable d'environnement `PERL5LIB`. Voir la description de la variable d'environnement `PERL5LIB` au chapitre 19, *L'interface de la ligne de commande*.

```
# syntaxe pour sh, bash, ksh ou zsh
$ PERLLIB=$HOME/libperl; export PERL5LIB

# syntaxe pour csh ou tcsh
$ setenv PERL5LIB ~/libperl
```

Si vous voulez utiliser des répertoires optionnels uniquement pour ce programme sans changer son code source, regardez du côté de l'option de la ligne de commande **`-I`** :

```
% perl -I ~/libperl chemin-programme args
```

Voir le chapitre 19 pour plus d'informations sur l'emploi de **`-I`** dans la ligne de commande.

use locale

```
@x = sort @y;          # tri dans l'ordre ASCII
{
    use locale;
    @x = sort @y;      # tri dans l'ordre défini par les locales
}
@x = sort @y;          # à nouveau tri dans l'ordre ASCII
```

Ce pragma de portée lexicale indique au compilateur d'activer (ou de désactiver, avec `no locale`) l'emploi des locales POSIX pour les opérations intrinsèques. L'activation des locales indique aux fonctions de conversions de casse (majuscule/minuscule) de Perl et au moteur de recherche de motifs de respecter l'environnement de votre langue, en autorisant les signes diacritiques[3], etc. Si ce pragma est actif et que votre bibliothèque C connaît les locales POSIX, Perl recherche la valeur de votre variable d'environnement `LC_CTYPE` pour les expressions régulières et `LC_COLLATE` pour les comparaisons de chaînes comme celles de `sort`.

Comme les locales sont plus une forme de nationalisation que d'internationalisation, l'emploi des locales peut interférer avec Unicode. Voir le chapitre 15 pour plus d'informations sur l'internationalisation.

3. *N.d.T.* : signe qui donne à un caractère de l'alphabet une valeur spéciale.

use open

```
use open IN => ":crlf", OUT => ":raw";
```

Le pragma open déclare une ou plusieurs disciplines par défaut pour les opérations d'entrées/sorties. Chaque opérateur open et readpipe (c'est-à-dire qx// ou les apostrophes inverses) découvert à l'intérieur de la portée lexicale de ce pragma et qui ne spécifie pas ses propres disciplines utilisera celles déclarées par défaut. Ni open avec un ensemble explicite de disciplines, ni sysopen en toute circonstance, ne sont influencés par ce pragma.

Seules les deux disciplines :raw et :crlf sont actuellement implémentées (cependant au moment où nous écrivons cela, nous espérons qu'une discipline :utf8 le soit également bientôt). Sur les systèmes ancestraux qui font la distinction entre ces deux modes de conversions lors de l'ouverture de fichiers, la discipline :raw correspond au « mode binaire » et :crlf au « mode texte ». (Ces deux disciplines n'ont actuellement aucun effet sur les plates-formes où binmode n'en a pas non plus, mais seulement pour le moment ; voir la fonction open au chapitre 29, *Fonctions*, pour une description plus détaillée des sémantiques que nous espérons en ce qui concerne les diverses disciplines.)

Le support exhaustif des disciplines d'entrées/sorties n'est pas encore implémenté à ce jour. Lorsqu'elles seront enfin supportées, ce pragma servira comme l'une des interfaces pour déclarer les disciplines par défaut pour toutes les entrées/sorties. À ce moment-là, les disciplines par défaut déclarées avec ce pragma seront accessibles par le biais de la discipline spéciale nommée « :DEFAULT » et exploitables à l'intérieur des constructeurs de handles qui autorisent la spécification de disciplines. Cela rendra possible l'empilement de nouvelles disciplines par-dessus celles par défaut.

```
open (HF, "<:para :DEFAULT", $fichier)
    or die "impossible d'ouvrir $fichier : $!";
```

Lorsqu'il sera complet, le support des disciplines d'entrées/sorties permettra à toutes les disciplines supportées de fonctionner sur toutes les plates-formes.

use overload

Dans le module Nombre :

```
package Nombre;
use overload "+" => \&mon_addition";
             "-" => \&ma_soustraction";
            "*=" => \&mon_produit_par";
```

Dans votre programme :

```
use Nombre;
$a = new Nombre 57;
$b = $a +5;
```

Les opérateurs internes fonctionnent bien avec des chaînes et des nombres mais n'ont que peu de sens lorsqu'on les applique à des références d'objets (puisque, contrairement à C ou à C++, Perl ne permet pas de faire de l'arithmétique de pointeur). Le pragma overload vous laisse redéfinir les significations de ces opérations internes lorsque vous

les appliquez à des objets de votre cru. Dans l'exemple précédent, l'appel au pragma redéfinit trois opération sur des objets `Nombre` : l'addition appellera la fonction `Nombre::mon_addition`, la soustraction appellera la fonction `Nombre::ma_soustraction` et l'opérateur d'assignement accompagné d'une multiplication appellera la méthode `mon_produit_par` de la classe `Nombre` (ou de l'une de ses classes de base). Nous disons que ces opérateurs sont *surchargés* (*overloaded*) car des significations supplémentaires les recouvrent (et non parce qu'ils ont trop de significations - quoique cela peut également être le cas).

Pour plus d'informations sur la surcharge, voir le chapitre 13, *Surcharge*.

use re

Ce pragma contrôle l'utilisation des expressions régulières. On peut l'invoquer de quatre façons : " `taint` " et " `eval` ", qui ont une portée lexicale, plus " `debug` " et " `debugcolor` " qui n'en ont pas.

```
use re 'taint';
# Le contenu de $corresp est marqué si $gribouille était aussi marqué.
($corresp) = ($gribouille =~ /(.*)$/s);

# Permet l'interpolation de code :
use re 'eval';
$motif = '(?{ $var = 1 })';     # exécution de code inséré
/alpha${motif}omega/;           # n'échouera pas à moins d'être lancé
                                # avec -T et que $motif soit marqué

use re 'debug';                 # comme "perl -Dr"
/^(.*)$/s;                      # affichage d'informations de débogage
                                #     durant la compilation et l'exécution

use re 'debugcolor'             # comme 'debug', mais avec un affichage
                                #     en couleur
```

Lorsque `use re 'taint'` est actif et qu'une chaîne marquée est la cible de l'expression régulière, les variables numérotées de l'expression régulière et les valeurs renvoyés par l'opérateur `m//` dans un contexte de liste sont également marquées. Ceci est très utile lorsque les opérations d'expressions régulières sur des données marquées ne sont pas vouées à extraire des sous-chaînes sécurisées mais à effectuer d'autres transformations. Voir la discussion sur le marquage au chapitre 23.

Lorsque `use re 'eval'` est actif, une expression régulière a le droit de contenir des assertions exécutant du code Perl, qui sont de la forme `(?{...})`, même si l'expression régulière contient des variables interpolées. L'exécution des segments de code résultant de l'interpolation de variables à l'intérieur des expressions régulières n'est pas habituellement permise pour des raisons de sécurité : vous ne voulez pas que des programmes qui lisent des motifs depuis des fichiers de configuration, des arguments de la ligne de commande ou des champs de formulaires CGI, se mettent subitement à exécuter un code arbitraire s'ils ne sont pas conçus pour s'attendre à cela. Cet emploi du pragma permet uniquement aux chaînes non-marquées d'être interpolées ; les données marquées causeront toujours la levée d'une exception (si vous activez la vérification des marquages).

Voir également le chapitre 5, *Correspondance de motifs* et le chapitre 23.

À l'intention de ce pragma, l'interpolation d'expressions régulières précompilées (produites par l'opérateur `qr//`) n'est pas considérée comme une interpolation de variable. Néanmoins, lorsque vous construisez le motif `qr//`, `use re 'eval'` doit être actif si l'une de ses chaînes interpolées contient des assertions de code. Par exemple :

```
$code = '(?{ $n++ })';          # assertion de code
$chaine = '\b\w+\b' . $code;    # construit la chaîne à interpoler
$ligne =~ /$chaine/;            # ceci a besoin de use re 'eval'
$motif = qr/$chaine/;           # ceci également
$ligne =~ /$motif/;             # mais pas ceci
```

Avec `use re 'debug'`, Perl émet des message de débogage à la compilation et à l'exécution d'expressions régulières. L'affichage est identique à celui obtenu en lançant un « débogueur Perl » (l'un de ceux compilés en passant **`-DDEBUGGING`** au compilateur C) et en exécutant alors votre programme Perl avec l'option de la ligne de commande de Perl **`-Dr`**. Selon la complexité de votre motif, l'affichage résultant peut vous noyer. L'appel de `use re 'debugcolor'` permet un affichage plus coloré qui peut s'avérer utile, pourvu que votre terminal comprenne les séquences de couleurs. Positionner votre variable d'environnement `PERL_RE_TC` avec une liste de propriétés *termcap*(5) pertinentes, séparées par des virgules, pour faire ressortir l'affichage. Pour plus de détails, voir le chapitre 20, *Le débogueur Perl*.

use sigtrap

```
use sigtrap;
use sigtrap qw(stack-trace old-interface-signals);   # idem
use sigtrap qw(BUS SEGV PIPE ABRT);
use sigtrap qw(die INT QUIT);
use sigtrap qw(die normal-signals);
use sigtrap qw(die untrapped normal-signals);
use sigtrap qw(die untrapped normal-signals stack-trace any error-signals);
```

Le pragma `sigtrap` installe pour vous quelques gestionnaires de signaux simples pour vous éviter de vous en inquiéter. Ceci est très utile dans les situations où les signaux non-interceptés causeraient des dysfonctionnement dans votre programme; comme lorsque vous avez des blocs `END {}`, des destructeurs d'objets ou d'autres traitements à lancer à la fin de votre programme quelle que soit la façon dont il se termine.

Le pragma `sigtrap` fournit deux gestionnaires de signaux simples que vous pouvez utiliser. L'un apporte à Perl une trace de la pile et l'autre lance une exception ordinaire *via* `die`. Vous pouvez aussi fournir votre propre gestionnaire au pragma pour qu'il l'installe. Vous pouvez spécifier des ensembles de signaux prédéfinis à intercepter ; vous pouvez également donner votre propre liste explicite de signaux. Le pragma peut optionnellement installer des gestionnaires uniquement pour les signaux qui n'auraient sinon pas été gérés.

Les arguments passés à `use sigtrap` sont traités dans l'ordre. Lorsqu'un nom de signal fourni par l'utilisateur ou le nom d'un des ensembles de signaux prédéfinis de `sigtrap` est rencontré, un gestionnaire est immédiatement installé. Lorsqu'il y a une option, celle-ci n'affecte que les gestionnaires installés ensuite en traitant la liste d'arguments.

Gestionnaires de signaux

Ces options ont pour effet de décider quel gestionnaire sera utilisé pour les signaux installés ensuite :

`stack-trace`

Ce gestionnaire fourni par le pragma affiche une trace de la pile d'exécution Perl vers `STDERR` et tente alors de faire une copie du cœur du programme (*core dump*). Il s'agit du gestionnaire de signaux par défaut.

`die`

Ce gestionnaire fourni par le pragma appelle `die` *via* `Carp::croak` avec un message indiquant le signal reçu.

`handler` *`VOTRE_GESTIONNAIRE`*

`VOTRE_GESTIONNAIRE` sera utilisé comme gestionnaire pour les signaux installés ensuite. *`VOTRE_GESTIONNAIRE`* peut être n'importe quelle valeur valide pour un assignement de `%SIG`. Souvenez-vous que le fonctionnement correct de beaucoup d'appels de la bibliothèque C (en particulier les appels d'entrées/sorties standards) n'est pas garanti à l'intérieur d'un gestionnaire de signaux. Pis encore, il est difficile de deviner quels morceaux du code de la bibliothèque C sont appelés depuis tel ou tel morceau du code Perl. (D'un autre côté, nombre de signaux que `sigtrap` intercepte sont un peu vicieux — ils vous feront tourner en bourrique de toute façon, alors ça ne coûte pas grand chose d'*essayer* de faire quelque chose, n'est-ce pas?)

Listes prédéfinies de signaux

Le pragma `sigtrap` inclut quelques listes prédéfinies de signaux à intercepter :

`normal-signals`

Il s'agit des signaux qu'un programme peut normalement s'attendre à rencontrer et qui le font, par défaut, se terminer. Ce sont les signaux `HUP`, `INT`, `PIPE` et `TERM`.

`error-signals`

Il s'agit des signaux qui d'ordinaire indiquent un sérieux problème avec l'interpréteur Perl ou avec votre programme. Ce sont les signaux `ABRT`, `BUS`, `EMT`, `FPE`, `KILL`, `QUIT`, `SEGV`, `SYS` et `TRAP`.

`old-interface-signals`

Il s'agit des signaux qui sont interceptés par défaut sous une ancienne version de l'interface de `sigtrap`. Ce sont les signaux `ABRT`, `BUS`, `EMT`, `FPE`, `KILL`, `PIPE`, `QUIT`, `SEGV`, `SYS`, `TERM` et `TRAP`. Si aucun signal ou aucune liste de signaux n'est passé à `use sigtrap`, on utilise cette liste.

Si votre plate-forme n'implémente pas un signal particulier cité dans les listes prédéfinies, le nom de ce signal sera discrètement ignoré. (Le signal lui-même ne peut pas être ignoré car il n'existe pas.)

Autres arguments de sigtrap

`untrapped`

Ce token supprime les gestionnaires pour tous les signaux listés ultérieurement s'ils ont déjà été interceptés ou ignorés.

`any`

Ce token installe des gestionnaires pour tous les signaux listés ultérieurement. Il s'agit du comportement par défaut.

`signal`

Tout argument qui ressemble à un nom de signal (c'est-à-dire qui correspond au motif `/^[A-Z][A-Z0-9]*$/` enjoint à `sigtrap` de gérer ce signal.

`nombre`

Un argument numérique requiert que le numéro de version du pragma `sigtrap` soit au moins *nombre*. Ceci fonctionne exactement comme la plupart des modules ordinaires qui ont une variable de paquetage `$VERSION`.

```
% perl -Msigtrap -le 'print $sigtrap::VERSION'
1.02
```

Exemples de sigtrap

Fournit une trace de la pile pour les signaux utilisant l'ancienne interface :

```
use sigtrap;
```

Idem, mais plus explicite :

```
use sigtrap qw(stack-trace old-interface-signals);
```

Fournit une trace de la pile uniquement pour les quatre signaux listés :

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Meurt (*via* `die`) sur réception du signal `INT` ou `QUIT` :

```
use sigtrap qw(die INT QUIT);
```

Meurt (*via* `die`) sur réception de l'un des signaux `HUP`, `INT`, `PIPE` ou `TERM` :

```
use sigtrap qw(die normal-signals);
```

Meurt (*via* `die`) sur réception de `HUP`, `INT`, `PIPE` ou `TERM` — à cela près qu'on ne change pas le comportement pour les signaux qui ont déjà été interceptés ou ignorés ailleurs dans le programme :

```
use sigtrap qw(die untrapped normal-signals);
```

Meurt (*via* `die`) sur réception de n'importe quel signal « `normal-signals` » non-intercepté actuellement ; de plus, fournit une trace arrière de la pile d'exécution sur réception de l'un des signaux « `error-signals` » :

```
use sigtrap qw(die untrapped normal-signals stack-trace any error-signals);
```

Installe la routine `mon_gestionnaire` comme gestionnaire pour les signaux « `normal-signals` » :

```
use sigtrap 'handler' => \&mon_gestionnaire, 'normal-signals';
```

Installe la routine mon_gestionnaire comme gestionnaire pour les signaux « normal-signals » ; de plus, fournit une trace arrière de la pile d'exécution sur réception de l'un des signaux « error-signals » :

```
use sigtrap qw(handler mon_gestionnaire normal-signals stack-trace
               error-signals);
```

use strict

```
use strict;            # Installe les trois restrictions.

use strict "vars";     # Les variables doivent être prédéclarées.
use strict "refs";     # Les références symboliques sont interdites.
use strict "subs";     # Les chaînes brutes doivent être entre guillemets.

use strict;            # Installe toutes les restrictions
no strict "vars";      # puis renonce à l'une d'entre elles.
```

Ce pragma de portée lexicale modifie quelques règles basiques à propos de ce que Perl considère comme étant du code légal. Quelquefois, ces restrictions semblent trop strictes pour de la programmation occasionnelle, comme lorsque vous ne faites qu'essayer de programmer à la hâte un filtre de cinq lignes. Plus votre programme est gros, plus vous devez être strict avec lui.

Actuellement, vous devez être strict avec trois choses : subs, vars et refs. Si aucune liste d'importation n'est fournie, les trois restrictions sont prises en compte.

strict 'refs'

Cet exemple génère une erreur à l'exécution si vous utilisez des références symboliques, intentionnellement ou pas. Voir le chapitre 8, *Références* pour plus d'informations.

```
use strict 'refs';

$ref = \$truc;   # Enregistre une référence "réelle" (en dur).
print $$ref;     # Déréférencement ok.

$ref = "truc";   # Enregistre le nom d'une variable globale (de paquetage).
print $$ref;     # FAUX, erreur à l'exécution avec strict refs.
```

Les références symboliques sont suspectes pour diverses raisons. Il est étonnamment aisé, même pour des programmeurs bien intentionnés, de les invoquer accidentellement ; strict 'refs' l'empêche. Au contraire des références réelles, les références symboliques ne peuvent se référer qu'à des variables globales. Elles ne possèdent pas de compteur de références. Il existe souvent un meilleur moyen d'accomplir ce que vous faîtes : au lieu de référencer un symbole dans une table de symboles globale, utilisez un hachage pour sa propre mini-table de symboles. C'est plus efficace, plus lisible et moins sujet à engendrer des erreurs.

Néanmoins, certains types de manipulations valides nécessitent vraiment un accès direct à la table de symboles globale du paquetage pour les noms de variables et de fonctions. Par exemple, il se peut que vous vouliez examiner la liste @EXPORT de la superclasse @ISA d'un paquetage donné dont vous ne connaissez pas le nom à l'avance. Ou

vous pourriez vouloir installer toute une kyrielle d'appels de fonctions qui seraient toutes des alias de la même fermeture. C'est seulement là que les références symboliques sont les plus utiles, mais pour les utiliser alors que `use strict` est actif, vous devez d'abord annuler la restriction " `ref` " :

```
# construit un groupe d'accesseurs pour les attributs for my $nom_meth (qw/
nom grade matricule/) {
no strict 'refs';
*$nom_meth = sub { $_[0]->{ __PACKAGE__ . $nom_meth };
}
```

strict 'vars'

Dans cette restriction, une erreur de compilation est déclenchée si vous essayez d'accéder à une variable qui n'est pas conforme à au moins un de ces critères :

- Prédéfinie par Perl, comme `@ARGV`, `%ENV` et toutes les variables globales avec des signes de ponctuation comme `$.` ou `$_`.
- Déclarée avec `our` (pour une variable globale) ou `my` (pour une variable lexicale).
- Importée depuis un autre paquetage. (Le pragma `use vars` simule une importation, mais utilisez `our` à la place.)
- Pleinement qualifiée en utilisant le nom de son paquetage et le double deux-points séparateur de paquetage.

L'emploi seul d'un opérateur `local` n'est pas suffisant pour rendre heureux `use strict 'vars'` car, malgré son nom, cet opérateur ne change pas le fait qu'une variable nommée soit globale ou non. Il donne seulement à la variable une nouvelle valeur temporaire durant un bloc lors de l'exécution. Vous devrez toujours employer `our` pour déclarer une variable globale ou `my` pour déclarer une variable lexicale. Vous pouvez toutefois localiser un `our` :

```
local our $loi = "martiale";
```

Les variables globales prédéfinies par Perl sont dispensées de ces exigences. Ceci s'applique aux variables globales de tout le programme (celles forcées dans le paquetage `main` comme `@ARGV` ou `$_`) et aux variables par paquetage comme `$a` ou `$b`, qui sont normalement utilisée par la fonction `sort`. Les variables par paquetage utilisées par les modules comme `Exporter` ont toujours besoin d'être déclarées en utilisant `our` :

```
our @EXPORT_OK = qw(nom grade matricule);
```

strict 'subs'

Cette restriction oblige Perl à traiter les mots simples comme des erreurs de syntaxe. Un *mot simple* (*bareword* ou *bearword* dans certains dialectes) est un nom simple ou un identificateur qui n'a aucune autre interprétation forcée par le contexte. (Le contexte est souvent forcé par la proximité d'un mot clef ou d'un token ou par la prédéclaration du mot en question.) Historiquement, les mots simples étaient interprétés comme des chaînes sans guillemets ou apostrophes. Cette restriction rend cette interprétation illégale. Si vous avez l'intention de l'utiliser en tant que chaîne, mettez-la entre guillemets ou entre apostrophes. Si vous avez l'intention de l'utiliser en tant que fonction, prédéclarez-la ou employez des parenthèses.

Comme cas particulier de contexte forcé, souvenez-vous qu'un mot qui apparaît entre des accolades ou du côté gauche de l'opérateur => compte comme s'il était entre apostrophes et n'est donc pas sujet à cette restriction.

```
use strict 'subs';

$x = n_importe_quoi;                # FAUX : erreur de mot simple !
$x = n_importe_quoi();              # Par contre, cela marche toujours.

sub n_importe_quoi;                 # Prédéclaration de fonction.
$x = n_importe_quoi;                # Maintenant ok.

# Ces utilisations sont permises car => met entre apostrophes :
%hachage = (rouge => 1, vert => 2, bleu => 3);

$num_rouge = $hachage{rouge};       # Ok, les accolades mettent entre
                                    # apostrophes ici.

# Mais pas celles-ci :
@nums = @hachage{bleu, vert};       # Faux : erreur de mot simple.
@nums * @hachage{"bleu", "vert"};   # Ok, les mots sont maintenant entre
                                    # guillemets.
@nums = @hachage{qw/bleu vert/};    # Idem.
```

use subs

```
use subs qw/am stram gram/;
@x = am 3..10;
@x = gram stram @x;
```

Ce pragma prédéclare tous les noms de la liste d'arguments comme étant des sous-programmes standards. L'avantage ici est que vous pouvez maintenant utiliser ces fonctions sans parenthèses en tant qu'opérateurs de listes, exactement comme si vous les aviez déclarées vous-mêmes. Ce n'est pas forcément aussi utile que des déclarations complètes car cela ne permet pas d'utiliser des prototypes ou des attributs comme dans :

```
sub am(@);
sub stram(\@) : locked;
sub gram($) : lvalue;
```

Puisqu'il est basé sur le mécanisme d'importation standard, le pragma `use subs` n'a pas de portée lexicale mais une portée de paquetage. C'est-à-dire que les déclarations sont effectives pour la totalité du fichier dans lequel elles apparaissent, mais seulement pour le paquetage courant. Vous ne pouvez pas annuler de telles déclarations avec `no subs`.

use vars

```
use vars qw($bricole @refait %vu);
```

Ce pragma, autrefois utilisé pour déclarer une variable globale, est maintenant quelque peu déprécié en faveur du modificateur `our`. La déclaration précédente s'effectue mieux

en utilisant :

```
our($bricole, @refait, %vu};
```

ou même :

```
our $bricole = "F";
our @refait = "A" .. $bricole;
our %vu = ();
```

Peu importe que vous utilisiez l'une ou l'autre de ces possibilités, rappelez-vous qu'elles portent toutes sur des variables globales de paquetage, et non sur des variables lexicales ayant la portée du fichier.

use warnings

```
use warnings;        # Revient à importer "all".
no warnings;         # Revient à annuler l'importation de "all".

use warnings::register;
if (warnings::enabled()) {
    warnings::warn("un avertissement");
}

if (warnings::enabled("void")) {
    warnings::warn("void", "un avertissement");
}
```

Ce pragma de portée lexicale permet un contrôle souple sur les avertissements internes de Perl, à la fois ceux émis par le compilateur et ceux provenant du système d'exécution.

Autrefois, le seul contrôle que vous aviez sur le traitement des avertissements de votre programme se faisait par le biais de l'option de la ligne de commande **-w** ou de la variable `$^W`. Bien qu'utile, ces solutions tendaient à faire du tout-ou-rien. L'option **-w** active des avertissements dans des morceaux de code que vous pouvez ne pas avoir écrits, ce qui est parfois problématique pour vous et embarassant pour l'auteur original. L'emploi de `$^W`, soit pour activer, soit pour désactiver des blocs de code, peut s'avérer moins qu'optionnel car il ne fonctionne que durant l'exécution et non pendant la compilation.[4] Un autre problème est que cette variable globale à tout le programme possède une portée dynamique et non lexicale. Cela signifie que si vous l'activez dans un bloc et qu'ensuite vous appelez à partir de là un autre code, vous risquez ici encore d'activer des avertissements dans du code qui n'est pas développé avec exactement de tels standards à l'esprit.

Le pragma `warnings` contourne ces limitations en possédant une portée lexicale, des mécanismes durant la compilation qui permettent un contrôle plus fin sur l'endroit où les avertissements peuvent ou ne peuvent pas être déclenchés. Une hiérarchie de catégories d'avertissements (voir la *figure 31-1*) a été définie pour permettre aux groupes d'avertissements d'être activés ou désactivés séparément les uns des autres. (La catégorisation exacte est expérimentale et sujette à modifications.) Ces catégories peuvent être combinées en passant des arguments multiples à `use` ou `no` :

4. En l'absence de blocs `BEGIN`, bien entendu.

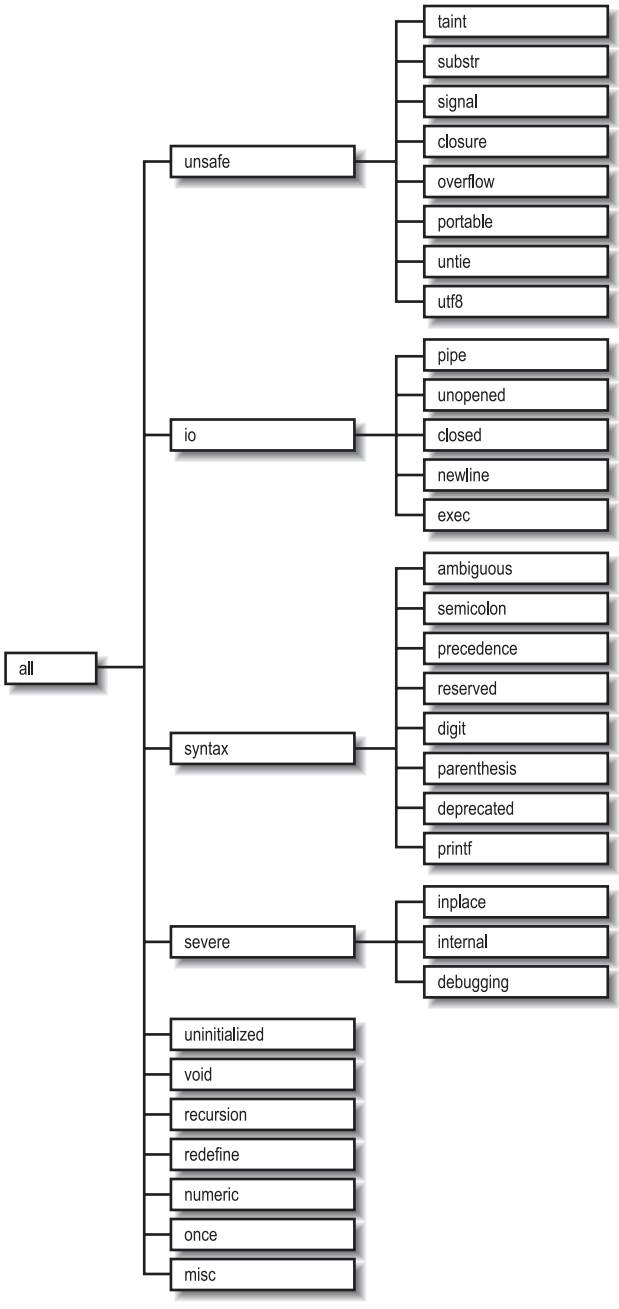


Figure 31-1. Catégories des avertissements Perl

```
use warnings qw(void redefine);
no warnings qw(io syntax untie);
```

Si des instances multiples du pragma `warnings` sont actives pour un groupe donné, leurs effets se cumulent :

```
use warnings "void";       # Seuls les avertissements "void" sont actifs.
...
use warnings "io";         # Les avertissements "void" et "io" sont
                           # maintenant tous deux actifs.
...
no warnings "void";        # Seuls les avertissements "io" sont maintenant
                           # actifs.
```

Pour donner des erreurs fatales pour tous les avertissements activés par un pragma `warnings` particulier, utilisez le mot `FATAL` devant la liste d'importation. Ceci est très utile lorsque vous aimez mieux qu'une certaine condition, qui d'ordinaire n'aurait causé qu'un avertissement, arrête votre programme. Supposez par exemple que vous considériez l'emploi d'une chaîne invalide en tant que nombre (qui normalement aurait donné une valeur de 0) si incorrect que vous vouliez que cet acte effronté tue votre programme. Pendant que vous y êtes, vous décidez que l'utilisation de valeurs non-initialisées en lieu et place de chaînes réelles ou de valeurs numériques doit également être la cause d'un suicide immédiat :

```
{
    use warnings FATAL => qw(numeric unitialized);
    $x = $y + $z;
}
```

Maintenant, si `$y` ou `$z` ne sont pas initialisées (c'est-à-dire, contiennent la valeur scalaire spéciale `undef`) ou si elles contiennent des chaînes qui ne se convertissent pas proprement en valeurs numériques, au lieu de continuer gentiment son chemin ou tout au plus d'émettre une petite complainte si vous aviez activé **`-w`**, votre programme lèvera maintenant une exception. (Pensez à cela comme du Perl tournant en mode Python.) Si vous n'interceptez pas les exceptions, elles causeront une erreur fatale. Le texte de l'exception est le même que celui qui serait normalement apparu dans le message d'avertissement.

Le pragma `warnings` ignore l'option de la ligne de commande **`-w`** et la valeur de la variable `$^W` ; les configurations du pragma sont prioritaires. Toutefois, le drapeau **`-W`** de la ligne de commande passe outre le pragma, en activant totalement les avertissements dans tous le code à l'intérieur de votre programme, y compris le code chargé avec `do`, `require` ou `use`. En d'autres termes, avec **`-W`**, Perl prétend que chaque bloc de votre programme possède un pragma `use warnings 'all'`. Pensez-y comme un *lint*(1) pour les programmes Perl. (Mais voyez également la documentation en ligne du module `B::Lint`.) Le drapeau **`-X`** de la ligne de commande fonctionne à l'inverse. Il prétend que chaque bloc a un `no warnings 'all'` activé.

Plusieurs fonctions sont fournies pour assister les auteurs de modules qui veulent que les fonctions de leur module se comportent comme les fonctions internes en respectant la portée lexicale de l'appelant (c'est-à-dire, de manière à ce que les utilisateurs du module puissent lexicalement activer ou désactiver les avertissements que le module pourrait émettre) :

`warnings::register`

Enregistre le nom du module courant comme une nouvelle catégorie d'avertissements, afin que les utilisateurs du module puissent en stopper les avertissements.

`warnings::enabled(`*`CATEGORIE`*`)`

Renvoie vrai si la catégorie *`CATEGORIE`* est activée dans la portée lexicale du module appelant. Sinon, renvoie faux. Si *`CATEGORIE`* n'est pas fournie, le nom du paquetage courant est utilisé.

`warnings::warn(`*`CATEGORIE, MESSAGE`*`)`

Si le module appelant n'a *pas* positionné *`CATEGORIE`* à « `FATAL` », affiche *`MESSAGE`* sur `STDERR`. Si le module appelant a positionné *`CATEGORIE`* à « `FATAL` », affiche *`MESSAGE`* sur `STDERR`, puis meurt. Si *`CATEGORIE`* n'est pas fournie, le nom du paquetage courant est utilisé.

32

Modules standards

L'édition précédente de ce livre couvrait de manière très complète et définitive tous les modules faisant partie de la distribution standard de Perl. Si nous avions refait la même chose ici, vous auriez payez de livre le double, sans parler de ce que vous auriez dû payer au chiropracteur après avoir essayé de porter le livre jusqu'à chez vous. Ces dernières années, de plus en plus de modules ont été inclus dans la distribution standard ; nous en sommes à ce jour à environ deux cents. Certains d'entre eux, comme `CGI`, proposent eux-mêmes une documentation remarquablement approfondie. Et si vous utilisez la distribution de Perl ActiveState, votre bibliothèque standard est encore plus raffinée.

Ainsi, nous vous offrons à la place une énumération complète des modules standards, classés par type, accompagnés d'une brève description. Puis, cerise sur le gâteau, nous avons sélectionné quelques uns de nos modules favoris, vous en donnant des exemples d'utilisation classique, suivis d'une courte description de leur fonctionnement, juste pour que vous y goûtiez un peu. Les descriptions sont suggestives plutôt que complètes et quasiment tous les modules ont d'autres fonctionnalités que celles que nous montrons. Toutefois, une documentation complète de tous les modules standards est incluse dans chaque distribution de Perl, vous n'avez donc qu'à rechercher les détails sur votre propre système en utilisant le programme *perldoc*, la commande *man*(1) de votre système ou votre navigateur préféré. Voir la section *Documentation en ligne* de la préface. Demandez à votre expert maison si vous ne trouvez pas les documents, car ils sont quasi certainement *quelque part* sur votre système. Et même si ils n'y sont pas, vous pouvez toujours les lire au format pod directement dans les modules eux-mêmes car toute la documentation des modules est insérée dans le fichier de module (*.pm*) lui correspondant et le format pod a été conçu pour être assez lisible. (Pas comme, disons, HTML.)

Listes par type

Le nom des modules ordinaires commence par une majuscule. Des noms en minuscules indiquent des pragmas dont vous trouverez la documentation au chapitre 31, *Modules de pragmas*, plutôt que dans le présent chapitre.

Types de données

Ces modules étendent le système de typage de Perl (ou plutôt ses lacunes) de diverses manières.

Module	Description
`Class::Struct`	Crée des classes d'objets Perl ressemblant à `struct` .
`constant`	Déclare des scalaires et des listes constantes.
`integer`	Force Perl à faire de l'arithmétique avec des entiers au lieu de nombres à virgule flottante.
`Math::BigFloat`	Calcule avec des nombres à virgule flottante d'une longueur arbitraire.
`Math::BigInt`	Calcule avec des entiers de longueur arbitraire.
`Mah::Complex`	Calcule avec des nombres complexes et les fonctions mathématiques associées.
`Math::Trig`	Charge beaucoup de fonctions trigonométriques.
`overload`	Surcharge les opérateurs Perl sur des objets.
`Symbol`	Manipule les tables de symboles de Perl et génère des typeglobs anonymes.
`Time:Local`	Calcule efficacement la date et l'heure origine selon `locatime` ou `gmtime`.

Traitement des chaînes, traitement de textes linguistiques, analyse et recherche

Ces modules agissent avec (ou sur) des textes.

Module	Description
`Search::Dict`	Effectue une recherche binaire d'une chaîne dans un fichier texte trié.
`Text::Abbrev`	Crée une table d'abbréviation à partir d'une liste.
`Text::ParseWords`	Analyse du texte dans une liste de tokens ou dans un tableau de tableaux.
`Text::Soundex`	Utilise l'algorithme Soundex de Knuth.
`Text::Tabs`	Développe ou réduit les tabulations comme *expand*(1) ou *unexpand*(1).
`Text::Wrap`	Équilibre des lignes pour former des paragraphes simples.

Traitement des options, arguments, paramètres et fichiers de configurations

Ces modules traitent votre ligne de commande.

Module	Description
`Getopt::Long`	Traite des options étendues de la ligne de commande dans une forme longue (`-xyz`).
`Getopt::Std`	Traite des options d'un seul caractère avec regroupement d'options.

Noms de fichier, systèmes de fichiers et verrouillage de fichiers

Ces modules fournissent des méthodes d'accès aux fichiers sur plusieurs plates-formes.

Module	Description
`Cwd`	Obtient le chemin du répertoire de travail courant.
`File::Basename`	Décompose de manière portable le nom d'un chemin en ses composants : répertoire, nom de base et extension.
`File::CheckTree`	Lance plusieurs tests de fichiers sur un ensemble d'arborescences de répertoires.
`File::Compare`	Compare de manière portable les contenus de fichiers ou de handles de fichiers.
`File::Copy`	Copie de manière portable des fichiers ou des handles de fichiers ou déplace des fichiers.
`File::DosGlob`	Réalise des globs dans le style Microsoft.
`File::Find`	Parcourt une arborescence de fichiers comme *find*(1).
`File::Glob`	Utilise des globs dans le style Unix.
`File::Path`	Crée ou supprime de manière portable une série de répertoires.
`File::Spec`	Utilise des opérations portables sur les fichiers (interface orientée objet).
`File::Spec::Functions`	Utilise des opérations portables sur les fichiers (interface fonctionnelle).
`File::Spec::Mac`	Utilise des opérations sur les fichiers pour Mac OS.
`File::Spec::OS2`	Utilise des opérations sur les fichiers pour OS/2.
`File::Spec::Unix`	Utilise des opérations sur les fichiers pour Unix.
`File::Spec::VMS`	Utilise des opérations sur les fichiers pour VMS.
`File::Spec::Win32`	Utilise des opérations sur les fichiers pour Microsoft.
`File::stat`	Supplante les fonctions `stat` et `lstat` avec une interface utilisant des méthodes.

Utilitaires pour les handles de fichiers, les handles de répertoires et les flux d'entrées/sorties

Ces modules fournissent un accès orienté objet aux fichiers, aux répertoires et aux IPC.

Module	Description
`DirHandle`	Utilise des méthodes objets pour les handles de répertoires.
`FileCache`	Garde plus de fichiers ouverts simultanément en écriture que votre système ne le permet.
`FileHandle`	Utilise des méthodes objets pour les handles de fichiers.
`IO`	Fournit un frontal pour charger tous les modules `IO::Dir`, `IO::File`, `IO::Handle`, `IO::Pipe`, `IO::Seekable` et `IO::Socket`.
`IO::Dir`	Utilise des méthodes objets pour les handles de répertoires.
`IO::File`	Utilise des méthodes objets en rapport avec les fichiers pour les handles de fichiers.
`IO::Handle`	Utilise des méthodes objets génériques pour les handles de fichiers/
`IO::Pipe`	Utilise des méthodes objets pour les pipes.
`IO::Poll`	Fournit une interface orientée objet à l'appel système *poll*(2).
`IO::Seekable`	Utilise des méthodes objets pour les objets d'entrées/sorties dans lesquels on peut faire des recherches (avec `seek`).
`IO::Select`	Utilise une interface orientée objet facilitant l'appel système *select*(2).
`SelectSaver`	Sauvegarde et restaure les handles de fichiers sélectionnés.

Internationalisation et locales

Ces modules vous aident dans vos activités extra-américaines.

Module	Description
`bytes`	Force l'ancienne sémantique orientée octets.
`charnames`	Définit les noms de caractères utilisés dans les séquences d'échappements `\N{`*nom*`}` pour les chaînes littérales.
`I18N::Collate`	Compare des données scalaires de 8 bits selon la locale courante (déprécié).
`locale`	Utilise ou évite les locales POSIX pour les opérations internes.
`utf8`	Bascule explicitement sur le support UTF-8 et Unicode.

Interfaces au système d'exploitation

Ces modules ajustent votre interface au système d'exploitation.

Module	Description
`Fcntl`	Charge les définitions de *fcntl.h* de la bibliothèque C en tant que constantes Perl.
`filetest`	Contrôle les opérateurs de tests de fichiers (`-r`, `-w`, etc.) pour les systèmes non-traditionnels.
`open`	Positionne les disciplines par défaut pour les appels à la fonction `open`.
`POSIX`	Utilise l'interface Perl à POSIX 1003.1.
`Shell`	Lance des commandes shells de manière transparente à l'intérieur de Perl.
`sigtrap`	Active une gestion simple des signaux.
`Sys::Hostname`	Essaye de manière portable tous les moyens imaginables pour déterminer le nom de l'hôte courant.
`Sys::Syslog`	Utilise les fonctions *syslog*(3) de la bibliothèque C.
`Time::gmtime`	Supplante la fonction interne `gmtime` par une interface méthodes.
`Time::localtime`	Supplante la fonction interne `localtime` par une interface méthodes.
`Time::tm`	Fournit l'objet interne utilisé par `Time::gmtime` et `Time::localtime`.
`User::grent`	Supplante les fonctions internes `getgr*` par une interface méthodes.
`User::pwent`	Supplante les fonctions internes `getpw*` par une interface méthodes.

Communication réseau et interprocessus

Ces modules fournissent des interfaces abstraites à côté des interfaces basiques que nous avons décrites au chapitre 16, *Communication interprocessus*.

Module	Description
`IO::Socket`	Utilise une interface orientée objet générique pour les communications sur des sockets.
`IO::Socket::INET`	Utilise une interface orientée objet pour les sockets du domaine Internet.
`IO::Socket::UNIX`	Utilise une interface orientée objet pour les sockets du domaine Unix (locales).
`IPC::Msg`	Utilise des méthodes objets pour travailler avec les messages System V (classe objet SysV Msg IPC).

Module	Description
IPC::Open2	Ouvre un processus en lecture et écriture simultanées.
IPC::Open3	Ouvre un processus pour lire, écrire et gérer les erreurs.
IPC::Semaphore	Utilise des méthodes objets pour les sémaphores System V.
IPC::SysV	Définit des constantes pour tous les mécanismes des IPC System V.
Net::hostent	Supplante les fonctions internes gethost* par une interface méthodes.
Net::netent	Supplante les fonctions internes getnet* par une interface méthodes.
Net::Ping	Vérifie si un hôte distant est joignable.
Net::protoent	Supplante les fonctions internes getproto* par une interface méthodes.
Net::servent	Supplante les fonctions internes getserv* par une interface méthodes.
Socket	Charge les définitions et les fonctions manipulant des structures de *socket.h* de la bibliothèque C.

World Wide Web

Ces modules interfacent le WWW. Vous devez en avoir entendu parler.

Module	Description
CGI	Accède aux formulaires CGI et à la génération performante et automatique de HTML.
CGI::Apache	Fait fonctionner vos scripts CGI sous l'API Perl-Apache.
CGI::Carp	Écrit dans le journal d'erreurs CGI de *httpd*(8) (ou d'autres).
CGI::Cookie	Lit et écrit des cookies HTTP.
CGI::Fast	Utilise le protocol Fast CGI.
CGI::Pretty	Produit du code HTML agréablement formaté.
CGI::Push	Réalise des scripts CGI *server-push* (*N.d.T.* : sollicités par le serveur).

Interfaces DBM

Ces modules chargent diverses bibliothèques de gestion de bases de donnée.

Module	Description
AnyDBM_File	Fournit une infrastructure pour de multiples bibliothèques DBM.
DB_File	Fournit un accès lié avec tie à la bibliothèque *db*(3) (Berkeley DB, version 1.x).

Module	Description
GDBM_File	Fournit un accès lié avec `tie` à la bibliothèque *gdbm*(3).
NDBM_File	Fournit un accès lié avec `tie` à la bibliothèque *ndbm*(3).
SDBM_File	Fournit un accès lié avec `tie` aux fichiers *sdbm*(3) (simple DBM).

Interfaces utilisateurs

Ces modules fournissent une API décente pour les entrées/sorties de l'interface en ligne de commande.

Module	Description
Term::Cap	Utilise la bibliothèque *termcap*(3).
Term::Complete	Réalise une complétion programmable par des commandes sur des listes de mots.
Term::Readline	Utilise l'un des nombreux paquetages readline.

Authentification, sécurité et encryptage

Ces modules fonctionnent avec les espaces mémoires protégés dans lesquels s'exécutent les programmes.

Module	Description
Opcode	Active ou désactive les lors codes d'opérations (*opcodes*) de la compilation de code Perl en vue d'une utilisation avec le module `Safe`.
ops	Restreint les opérations non-sûres lors de la compilation.
Safe	Compile et exécute du code dans des compartiments restreints.

Langage Perl, extensions et mécanismes internes

(À ne pas confondre avec les intentions et les mécanismes externes.)

Module	Description
attributes	Consulte ou positionne les attributs d'un sous-programme ou d'une variable.
attrs	Consulte ou positionne les attributs d'un sous-programme ou d'une variable (obsolète).
base	Établit l'héritage des classes de base à la compilation.
Data::Dumper	Sérialise des structures de données Perl.
DB	Accède à l'API expérimentale du débogueur Perl.
Devel::DProf	Profile l'exécution d'un programme Perl.

Module	Description
Devel::Peek	Charge des outils de débogage de données pour les programmeurs XS.
diagnostics	Force des avertissements verbeux et élimine les duplications.
Dumpvalue	Affiche à l'écran des valeurs de données.
English	Utilise des noms de variables plus longs pour les variables internes avec une ponctuation.
Env	Accède aux variables d'environnement dans %ENV en tant que scalaires ordinaires.
Errno	Charge les définitions de *errno.h* de la bibliothèque C et lie la variable %! avec tie.
Fatal	Remplace les fonctions internes avec des versions levant des exceptions lors d'un échec.
fields	Lors de la compilation déclare les données attributs d'une classe et vérifie les accès à ces données.
less	Demande moins de quelque chose de la part de Perl (non-implémenté).
re	Altère le comportement par défaut des expressions régulières.
strict	Restreint les constructions bâclées.
subs	Prédéclare les noms de sous-programmes dans le paquetage courant.
vars	Prédéclare des variables globales (obsolète voir our au chapitre 29.

Classes commodes

Ces modules fournissent des classes de base et d'autres facilités.

Module	Description
Tie::Array	Fournit une classe de base pour les tableaux liés avec tie.
Tie::Handle	Fournit une classe de base pour les handles liés avec tie.
Tie::Hash	Fournit une classe de base pour les hachages liés avec tie.
Tie::RefHash	Fournit des références comme clefs de hachage.
Tie::Scalar	Fournit une classe de base pour les scalaires liés avec tie.
Tie::SubstrHash	Fournit une interface similaire aux hachages pour une table de taille fixe avec des clefs de longueur fixe.
UNIVERSAL	Fournit une classe de base pour *toutes* les classes (références consacrées).

Avertissements et exceptions

Que faire lorsque les choses se passent mal.

Module	Description
`Carp`	Fournit des routines qui font des `warn` et des `die` avec la perspective de l'appelant.
`warnings`	Contrôle les avertissements à l'intérieur de la portée lexicale.

Support de documentation

Il existe une quantité de documentation à supporter.

Module	Description
`Pod::Checker`	Vérifie les erreurs de syntaxes des documents pod (utilisé par *podchecker*(1)) .
`Pod::Functions`	Liste les fonctions internes par type.
`Pod::Html`	Convertit les fichiers pod en HTML (utilisé par *pod2html*(1)).
`Pod::InputObjects`	Gère le support de documentation.
`Pod::Man`	Convertit du pod en format *troff*(1) pour le système *man*(1) (utilisé par *pod2man*(1)).
`Pod::Parser`	Fournit une classe de base pour créer des filtres et des traducteurs de pod.
`Pod::Select`	Extrait des sections de pod depuis l'entrée (utilisé par *podselect*(1)).
`Pod::Text`	Convertit des données pod en texte formaté en ASCII (utilisé par *pod2text*(1)).
`Pod::Text::Color`	Convertit des données pod en texte formaté en ASCII avec les séquences d'échapement ANSI pour les couleurs.
`Pod::Text::Termcap`	Convertit des données pod en texte ASCII avec les séquences d'échappement spécifiques du terminal.
`Pod::Usage`	Affiche un message d'utilisation dérivé de la documentation pod incluse dans le source.

Support d'installation de module

Ces modules aident d'autres modules à sauter à travers différents cerceaux.

Module	Description
`AutoLoader`	Charge des sous-programmes à la demande.
`AutoSplit`	Éclate un paquetage pour l'autochargement.
`autouse`	Diffère le chargement d'un module jusqu'à ce qu'une fonction de ce module soit utilisée.

Module	Description
`blib`	Utilise la bibliothèque depuis la version non-installée d'une extension par `Makemaker`.
`Config`	Accède aux informations de configuration de Perl.
`CPAN`	Interroge, rapatrie et construit des modules Perl depuis des sites CPAN.
`Devel::SelfStubber`	Génère des souches (*N.d.T.* : *stubs*) pour un module en utilisant le module `SelfLoader`.
`DynaLoader`	Charge dynamiquement des bibliothèques C ou C++ en tant qu'extensions Perl.
`Exporter`	Implémente une méthode d'importation par défaut pour les modules traditionnels.
`ExtUtils::Command`	Fournit des utilitaires pour remplacer les commandes externes populaires dans un *Makefile*.
`ExtUtils::Embed`	Fournit des utilitaires pour inclure du Perl dans des programmes C ou C++.
`ExtUtils::Install`	Installe des fichiers dans la bibliothèque Perl du système.
`ExtUtils::Installed`	Gère l'inventaire des modules installée.
`ExtUtils::Liblist`	Détermine quelles bibliothèques utiliser et comment les utiliser.
`ExtUtils::MakeMaker`	Crée un *Makefile* pour une extension Perl.
`ExtUtils::Manifest`	Charge des outils pour écrire et vérifier un fichier *MANIFEST*.
`ExtUtils::Miniperl`	Écrit du code C pour *perlmain.c*.
`ExtUtils::Mkbootstrap`	Créer un fichier d'amorçage (*bootstrap*) pour être utilisé par `DynaLoader`.
`ExtUtils::Mksymlists`	Écrit des fichiers d'options de l'éditeur de liens pour les extensions dynamiques.
`ExtUtils::MM_Cygwin`	Supplante lesméthodes se comportant à la manière d'Unix dans `ExtUtils::MakeMaker`.
`ExtUtils::MM_OS2`	Supplanteles méthodes se comportant à la manière d'Unix dans `ExtUtils::MakeMaker`.
`ExtUtils::MM_Unix`	Fournit des méthodes utilisées par `ExtUtils::MakeMaker`.
`ExtUtils::MM_VMS`	Supplante les méthodes se comportant à la manière d'Unix dans `ExtUtils::MakeMaker`.
`ExtUtils::MM_Win32`	Supplante les méthodes se comportant à la manière d'Unix dans `ExtUtils::MakeMaker`.
`ExtUtils::Packlist`	Gère les fichiers *.packlist*.
`ExtUtils::testlib`	Ajoute des répertoires *blib/** à `@INC`.
`FindBin`	Localise le répertoire d'installation pour le programme Perl en train de tourner.
`lib`	Manipule `@INC` lors de la compilation.
`SelfLoader`	Charge des fonctions uniquement à la demande.

Module	Description
`XSLoader`	Charge dynamiquement des bibliothèques C ou C++ en tant qu'extensions Perl.

Support de développement

Ces modules s'occupent des mesures de temps et des tests, pour voir dans quelles proportions votre code ne tourne ni plus rapidement, ni plus proprement.

Module	Description
`Benchmark`	Compare les temps d'exécution de différentes versions de code.
`Test`	Utilise une infrastructure simple pour écrire des scripts de test.
`Test::Harness`	Lance des scripts de test standards avec des statistiques.

Compilateur Perl et générateur de code

Ces modules supportent des générateurs de code vers des sorties (*N.d.T.* : *backends*) diverses et variées pour Perl.

Module	Description
`B`	Charge des générateurs de code Perl (alias le « compilateur Perl »).
`B::Asmdata`	Fournit une donnée autogénérée sur les opérations atomiques Perl pour produire du bytecode.
`B::Assembler`	Assemble du bytecode Perl.
`B::Bblock`	Parcours les blocs basiques dans l'arbre syntaxique.
`B::Bytecode`	Utilise la sortie en bytecode du compilateur Perl.
`B::C`	Utilise la sortie de traduction en C du compilateur Perl.
`B::CC`	Utilise la sortie traduction optimisée en C du compilateur Perl.
`B::Debug`	Parcours l'arbre syntaxique Perl en affichant les infos de débogage à propos des opérations atomiques.
`B::Deparse`	Utilise la sortie du compilateur Perl pour reproduire du code Perl.
`B::Disassembler`	Désassemble du bytecode Perl.
`B::Lint`	Décèle les constructions douteuses.
`B::Showlex`	Montre les variables lexicales utilisées dans des fonctions ou des fichiers.
`B::Stash`	Montre quelles tables de symboles sont chargées.
`B::Terse`	Parcours l'arbre syntaxique Perl en affichant des infos laconiques à propos des opérations atomiques.
`B::Xref`	Génère des rapports avec des références croisées pour les programmes Perl.

Module	Description
`ByteLoader`	Charge du code Perl compilé en bytecode.
`O`	Fournit une interface générique aux sorties (*backends*) du compilateur Perl.

Modules relatifs à Microsoft

Si vous possédez la distribution Perl pour les systèmes Microsoft provenant d'ActiveState, les modules suivants, qui sont uniquement pour Microsoft, sont d'ores et déjà inclus pour vous. Si vous avez juste été chercher les sources de la distribution standard (en voulant peut-être la compiler avec l'émulateur d'environnement Unix de Cygwin) mais que vous tournez sur Wintel, vous pouvez obtenir tous ces modules depuis CPAN.

Module	Description
`Win32::ChangeNotify`	Surveille les événements relatifs aux fichiers et aux répertoires.
`Win32::Console`	Utilise la console Win32 et les fonctions du mode caractère.
`Win32::Event`	Utilise les objets événemetns Win32 depuis Perl.
`Win32::EventLog`	Traite les journaux d'événements Win32 depuis Perl.
`Win32::File`	Gère les attributs de fichier en Perl.
`Win32::FileSecurity`	Gère les *FileSecurity Discretion Access Control Lists* (Listes de contrôle d'accès discret aux fichiers sécurisés) en Perl.
`Win32::IPC`	Charge la classe de base pour les objets de synchronisation Win32.
`Win32::Internet`	Accède aux fonctions de *WININET.DLL*.
`Win32::Mutex`	Utilise les objets mutex Win32 en Perl.
`Win32::NetAdmin`	Gère les groupes et utilisateurs réseaux en Perl.
`Win32::NetResource`	Gère les ressources réseaux en Perl.
`Win32::ODBC`	Utilise les extensions ODBC pour Win32.
`Win32::OLE`	Utilise les extensions d'automatisation OLE.
`Win32::OLE::Const`	Extrait les définitions de constantes depuis *TypeLib*.
`Win32::OLE::Enum`	Utilise *OLE Automation Collection Objects* (objets collections d'automatisation OLE).
`Win32::OLE::NLS`	Utilise *OLE National Langage Support* (support de langages nationaux OLE).
`Win32::OLE::Variant`	Crée et modifie les variables OLE VARIANT.
`Win32::PerfLib`	Accède au compteur de performance de Windows NT.
`Win32::Process`	Crée et manipule des processus.
`Win32::Semaphore`	Utilise les objets sémaphores Win32.
`Win32::Service`	Gère les services du système.
`Win32::Sound`	Joue avec les sons Windows.
`Win32::TieRegistry`	Trifouille la base de registres.

Module	Description
`Win32API::File`	Accède aux appels bas niveau de l'API système Win32 pour les fichiers et les répertoires.
`Win32API::Net`	Gère les comptes LanManager de Windows NT.
`Win32API::Registry`	Accède aux appels bas niveau de l'API système Win32 depuis *WINREG.H*.

Benchmark

```
use Benchmark qw(timethese cmpthese timeit countit timestr);

# Vous pouvez toujours passer du code sous forme de chaînes :
timethese $compteur, {
    'Nom1' => '...code1...',
    'Nom2' => '...code2...',
};

# Ou sous forme de références à des sous-programmes :
timethese $compteur, {
    'Nom1' => sub { ...code1... },
    'Nom2' => sub { ...code2... },
};

cmpthese $compteur, {
    'Nom1' => '...code1...',
    'Nom2' => '...code2...',
};

$t = timeit $compteur, ' code';
print "$compteur boucles de code ont pris : ", timestr($t), "\n";

$t = countit $temps, ' code';
$compteur = $t->iters;
print "$compteur boucles de code ont pris : ", timestr($t), "\n";
```

Le module `Benchmark` peut vous aider à déterminer lequel parmi plusieurs choix possibles s'exécute le plus rapidement. La fonction `timethese` lance les segments de codes spécifiés le nombre de fois demandé et rapporte combien de temps a pris chaque segment. Vous pouvez obtenir un joli diagramme trié si vous appelez `cmpthese` de la même façon.

Les segments de codes peuvent être passés sous forme de références de fonctions au lieu de chaînes (en fait, il faut que ce soit des références de fonctions si vous utiliser des variables lexicales provenant de la portée de l'appelant), mais la surcharge engendrée par l'appel peut influencer les mesures de temps. Si vous ne demandez pas assez d'itérations pour avoir une bonne mesure, la fonctions émet un avertissement.

Des interfaces de bas niveau sont disponibles pour lancer un seul morceau de code soit un certain nombre de fois (`timeit`), soit un certain nombre de secondes (`countit`). Ces

fonctions renvoient des objets `Benchmark` (voir une description dans la documentation en ligne). Avec `countit`, vous savez que cela a tourné pendant assez longtemps pour éviter les avertissements, car vous aviez spécifié un temps d'exécution minimum.

Pour tirer le meilleur profit du module `Benchmark`, vous aurez besoin d'un peu d'entraînement. En général, il n'est pas suffisant de lancer deux ou trois algorithmes sur le même jeu de données, car les mesures de temps ne reflètent que la manière dont ces algorithmes se sont bien comporté sur ce jeu de données précis. Pour obtenir une meilleure impression sur le cas général, vous aurez besoin de lancer plusieurs ensembles de benchmarks en faisant varier les jeux de données utilisés.

Par exemple, imaginez que vous vouliez savoir le meilleur moyen d'obtenir la copie d'une chaîne sans ses deux derniers caractères. Vous pensez à quatre façons de faire cela (il y en a d'autres bien sûr) : `chop` deux fois, copier et substituer, ou utiliser `substr` soit sur la partie gauche, soit sur la partie droite d'un assignement. Vous testez ces algorithmes sur des chaînes de longueur `2`, `200` et `20_000` :

```
use Benchmark qw/countit cmpthese/;
sub lancer($) { countit(5, @_) }
for $taille (2, 200, 20_000) {
    $c = "." x $taille;
    print "\nTAILLE_DONNEES = $taille\n";
    cmpthese {
        chop2    => lancer q{
            $t = $c; chop $t; chop $t;
        },
        subs     => lancer q{
            ($t = $c) =~ s/..\Z//s;
        },
        substr_g => lancer q{
            $t = $c; substr($t, -2) = '';
        },
        substr_d => lancer q{
            $t = substr($c, 0, length($c) -2);
        },
    };
}
```

ce qui affiche :

```
TAILLE_DONNEES = 2
             Rate substr_g     subs    chop2 substr_d
substr_g 579109/s       --      -2%     -36%     -37%
subs     593574/s       2%       --     -34%     -36%
chop2    903371/s      56%      52%       --      -2%
substr_d 920343/s      59%      55%       2%       --

TAILLE_DONNEES = 200
             Rate     subs substr_g substr_d    chop2
subs     561402/s       --      -2%      -8%     -24%
substr_g 574852/s       2%       --      -6%     -22%
substr_d 610907/s       9%       6%       --     -17%
chop2    737478/s      31%      28%      20%       --
```

```
TAILLE_DONNEES = 20000
            Rate substr_d     subs substr_g    chop2
substr_d  35885/s       --     -45%     -45%     -46%
subs      65113/s      81%       --      -1%      -3%
substr_g  65472/s      82%       1%       --      -2%
chop2     66837/s      86%       3%       2%       --
```

Avec de petits jeux de données, l'algorithme « `substr_d` » tourne 2% plus rapidement que l'algorithme « `chop2` », mais sur de grands jeux de données, il tourne 46% plus lentement.

Sur des jeux de données vides (aucun de montré ici), le mécanisme de substitution est le plus rapide. Il n'y a donc pas de solution meilleure que les autres dans tous les cas possibles et même ces mesures ne racontent pas toute l'histoire, puisque vous êtes toujours à la merci de votre système d'exploitation et de la bibliothèque C avec laquelle Perl a été compilé. Ce qui est bon pour vous peut devenir mauvais pour quelqu'un d'autre. Cela prend un certain temps pour être adroit avec les benchmarks. Entretemps, cela facilite les choses d'être un bon menteur.

Carp

```
use Carp;
croak "Nous en sommes là !";

use Carp qw(:DEFAULT cluck);
cluck "Voilà comment nous sommes arrivés là !";
```

Le module `Carp` vous permet d'écrire des modules dont les fonctions rapportent les erreurs de la même façon que les opérateurs internes du point de vue de l'utilisateur du module. Le module `Carp` fournit des routines que vous utilisez pratiquement comme les fonctions internes standard `warn` et `die`, mais qui changent le nom de fichier et le numéro de ligne de façon à ce que l'erreur ait l'air de provenir du code utilisateur plutôt que de votre code. En bref, `Carp` est un moyen formidable de renvoyer la faute sur quelqu'un d'autre que soi.

Il existe en fait quatre fonctions. La fonction `carp` marche comme l'opérateur `warn` mais avec les informations sur le nom de fichier et le numéro de ligne relatives à l'appelant. La fonction `croak` marche comme `die` en levant une exception mais en donnant là encore des informations relatives à l'appelant. Si vous préférez une longue complainte, utilisez `cluck` et `confess` au lieu de `carp` et `croak` respectivement, et vous obtiendrez une trace complète de la pile rapportant qui a appelé qui et armé de quels arguments (sans aucun doûte le colonel Moutarde, dans la bibliothèque avec une clef anglaise). Vous devez importer `cluck` explicitement car il n'est pas exporté normalement. Les gens ne veulent pas en général des traces complètes de la pile pour de simples avertissements, pour une raison ou pour une autre.

CGI

```
use CGI qw(:standard);
$qui   = param("Nom");
```

```
$tel   = param("Telephone");
@choix = param("Choix");
```

Le module CGI vous aide à gérer des formulaires HTML, spécialement les formulaires en plusieurs étapes où il est critique de passer d'une étape à une autre. L'exemple extrêmement simple ci-dessus s'attend à traiter un formulaire avec deux paramètres prenant des valeurs scalaires, comme des champs de texte ou des boutons radios et un autre prenant une valeur multiple, comme des listes déroulantes spécifiées avec « MULTIPLE ». Le module est encore plus sympathique que cela à plusieurs niveaux, en supportant des fonctionnalités comme le traitement des cookies, les valeurs persistantes pour les caddies multiécrans et la génération automatique de listes HTML et de tables que vous pouvez extraire d'une base de données pour n'en citer que quelques unes. Le support de l'exécution à grande vitesse de scripts Perl précompilés *via* les services de mod_perl d'Apache est également fourni. Le livre de chez O'Reilly & Associates *Writing Apache Modules with Perl and C*, de Lincoln Stein et Doug MacEachern, vous dira tout sur ce sujet.

CGI::Carp

```
use CGI::Carp;
warn "Celle-ci est une complainte";   # Estampille avec le nom du
                                      # programme et la date
die "Mais celle-là est sérieuse";     # Mais ne cause pas d'erreur
                                      # serveur 500

use CGI::Carp qw(carpout);            # Importe cette fonction
open(LOG, "> >/var/tmp/mon_cgi-log"
    or die "Impossible d'ajouter à mon_cgi-log : $!\n";
carpout(*LOG);                        # Utilise maintenant le journal
                                      # d'erreurs spécifique au programme

use CGI::Carp qw(fatalsToBrowser);
die "Les messages d'erreurs fatales sont maintenant envoyés aussi au
navigateur";
```

Le module CGI::Carp fournit des versions des fonctions internes Perl warn et die, plus les fonctions carp, cluck, confess et croak du module Carp qui sont plus parlantes et plus sûres également. Elles sont plus parlantes car chaque message inclut la date et l'heure avec le nom du programme qui a émis les messages, ce qui est une aide précieuse lorsque vous utilisez un fichier journal partagé par une centaine de programmes différents qui le polluent tous en même temps avec un millier de messages différents.

Le module est également agréable pour les surfeurs du Web, puisque la mort prématurée d'un script CGI a tendance à causer des erreurs absconses « Server 500 » lorsque l'en-tête HTTP correcte ne part pas du serveur avant que votre programme ne trépasse, et ce module s'assure que cela ne se produit pas. La fonction carpout redirige tous les avertissements et toutes les erreurs vers le handle de fichier spécifié. La directive fatalsToBrowser envoie également une copie de tels messages au navigateur de l'utilisateur. Ces services facilitent le débogage des problèmes dans les scripts CGI.

Class::Struct

```
use Class::Struct;

struct Chef => {            # Crée un constructeur Chef->new().
    nom => '$',             # La méthode nom() accède maintenant à une
                            # valeur scalaire.
    salaire => '$',         # Ainsi que salaire().
    debut => '$',           # Ainsi que debut().
};

struct Boutique => {        # Crée un constructeur Boutique->new().
    proprio => '$',         # La méthode proprio() accède maintenant à
                            # un scalaire.
    adrs => '@',            # Et la méthode adresse() accède à un tableau.
    stock => '%',           # Et la méthode stock() accède à un hachage.
    patron => 'Chef'        # Initialise avec Chef->new().
};

$magasin = Boutique->new();
$magasin->proprio('Abdul Alhazred');
$magasin->adrs(0, 'Miskatonic University');
$magasin->adrs(1, 'Innsmouth, Mass.');
$magasin->stock("livres", 208);
$magasin->stock("charmes", 3);
$magasin->stock("potions", "aucune");
$magasin->patron->nom('Dr. L. P. Hache');
$magasin->patron->salaire('folie');
$magasin->patron->debut(scalar localtime);
```

Le module `Class:Struct` fournit un moyen de « déclarer » une classe ayant des objets dont les champs sont d'un certain type. La fonction qui fait cela est appelée `struct`. Comme les structures d'enregistrements ne sont pas des types de base en Perl, chaque fois que vous désirez créer une classe pour fournir un objet ressemblant à une donnée d'enregistrement, vous devez définir une méthode constructeur ainsi que des méthodes accesseurs pour chaque donnée de champ, quelquefois appelées méthodes « wrapper ». La fonction `struct` du module `Class::Struct` vous soulage de cette tâche en créant une classe à la volée. Vous n'avez qu'à lui communiquer qu'elles doivent être les données membres, ainsi que leur type. La fonction crée une méthode constructeur appelée `new` dans le paquetage spécifié par le premier argument, plus une méthode accesseur d'attribut pour chaque membre, comme spécifié par le second argument, qui doit être une référence de hachage.

Les types de champ sont spécifiés, soit en tant que types internes en utilisant les symboles habituels « `$` », « `@` », « `%` » et « `&` », soit en tant qu'une autre classe en employant le nom de la classe. Le type de chaque champ sera forcé lorsque vous essayerez de modifier la valeur.

Beaucoup de modules standards utilisent `Class:Struct` pour créer leur objets et accesseurs, y compris `Net::hostent` et `User::pwent`, dont vous pouvez regardez les sources et vous en servir comme modèles. Voir également les modules CPAN `Tie::SecureHash` et

`Class::Multimethods` pour des approches plus élaborées de l'autogénération de classes et de méthodes accesseurs. Voir la section *Gestion des données d'instance* au chapitre 12, *Objets*.

Config

```
use Config;
if ($Config{cc} =~ /gcc/) {
    print "Ce perl a été compilé avec GNU C.\n";
}
use Config qw(myconfig config_sh config_vars);
print myconfig();           # comme perl -V sans aucun motif
print config_sh();          # donne absolument tout

config_vars qw/osname osvers archname/;
```

Le mécanisme de configuration utilisé pour compiler et installer Perl rassemble une profusion d'informations sur votre système. Le module `Config` exporte par défaut une variable de hachage liée avec `tie`, nommée `%Config`, qui donne accès à environ 900 valeurs de configuration différentes. (Ces valeurs sont également accessibles à travers l'option **`-V`**:*`MOTIF`* de la ligne de commande de Perl.) `Config` fournit également trois fonctions qui donnent un accès à ces valeurs dans le style du shell, comme montré ci-dessus. Par exemple, le dernier appel pourrait afficher :

```
osname='linux';
osvers='2.4.2';
archname='i386-linux';
```

La documentation en ligne du module décrit les variables de configuration et leurs valeurs possibles. Prenez garde si vous déplacez votre exécutable *perl* vers un système autre que celui sur lequel il a été compilé, ces valeurs peuvent ne pas refléter la réalité actuelle ; par exemple, si vous faites tourner un binaire Linux ou Solaris sur un système BSD.

CPAN

```
# Obtient un shell CPAN interactif.
%perl -MCPAN -e shell

# S'enquiert juste des recommandations de mise à niveau.
% perl -MCPAN -e 'CPAN::Shell->r'

# Installe le module indiqué en mode batch.
% perl -MCPAN -e "install Class::Multimethods"
```

Le module CPAN est une interface automatisée et conviviale au *Comprehensive Perl Archive Network*, décrit au chapitre 22, *CPAN*. Au contraire de la plupart des modules que vous pouvez rencontrer, il est destiné à être invoqué depuis la ligne de commande, exactement comme un petit programme. La première fois que vous l'appelez, le module vous demande le site miroir CPAN par défaut et d'autres renseignements dont il a besoin. Après cela, vous pouvez lancer son shell interactif pour faire des requêtes et sélec-

tionner des modules à installer, demander au module des recommandations sur les modules devant être remis à niveau ou simplement lui faire installer un module particulier.

Cwd

```
use Cwd;
$rep = getcwd();           # Où suis-je ?

use Cwd 'chdir';
chdir "/tmp";              # Met à jour $ENV{PWD}.

use Cwd 'realpath';
print realpath("/usr////spool//mqueue/../");  # affiche /var/spool
```

Le module `Cwd` fournit des fonctions indépendantes de la plate-forme pour déterminer le répertoire de travail courant de votre processus. C'est mieux que de lancer un shell exécutant *pwd*(1) car les systèmes non-conformes à POSIX ne garantissent pas qu'ils connaissent une telle commande et Perl ne se limite pas uniquement à tourner sur les plates-formes POSIX. La fonction `getcwd`, laquelle est exportée par défaut, renvoie le répertoire de travail courant en utilisant le mécanisme réputé pour être le plus sûr sur la plate-forme courante. Si vous importez la fonction `chdir`, elle supplante l'opérateur interne avec celui du module, lequel met à jour la variable d'environnement `$ENV{PWD}` ; les commandes que vous pourriez lancer ensuite et qui tiendraient compte de cette variable auraient alors une vision cohérente de leur univers. La fonction `realpath` résout dans son argument représentant un nom de chemin tous les liens symboliques et tous les composants de chemin relatif afin de renvoyer un chemin complet de répertoire sous une forme canonique, exactement comme *realpath*(3).

Data::Dumper

```
use Data::Dumper;
print Dumper($magasin);
```

Lorsqu'on l'utilise sur l'exemple tiré de la définition de `Class::Struct`, ceci affiche :

```
$VAR1 = bless( {
             'Boutique::patron' => bless( {
                                          'Chef::salaire' =>
                                              'folie',
                                          'Chef::nom' =>
                                              'Dr. L. P. Hache',
                                          'Chef::debut' =>
                                              'Fri Apr 13 19:33:04 2001'
                                        }, 'Chef' ),
             'Boutique::stock' => {
                                    'livres' => 208,
                                    'charmes' => 3,
                                    'potions' => 'aucune'
                                  },
```

```
    'Boutique::proprio' => 'Abdul Alhazred',
    'Boutique::adrs' => [
                            'Miskatonic University',
                            'Innsmouth, Mass.'
                        ]
}, 'Boutique' );
```

La fonction `Dumper` du module `Data::Dumper` prend une liste de scalaires (incluant des références, qui peuvent même se référer à des objets) et renvoie une chaîne affichable ou `eval`uable qui reproduit précisément une copie fidèle de l'original. Vous pouvez utiliser ceci pour écrire une version de sauvegarde d'une structure de donnée dans un fichier ordinaire sur disque ou dans un fichier DBM, ou encore la passer à un autre processus *via* un pipe ou une connexion socket. Ce module peut être utilisé avec les modules `MLDBM` de CPAN et `DB_File` pour implémenter un fichier DBM à même de stocker de manière transparente des valeurs de données complexes, et non seulement des chaînes à plat. D'autres module de transformation en chaînes (*stringification*) (ou *sérialisation*, ou formatage de présentation, *marshalling*) incluent `Storable` et `FreezeThaw`, tous deux disponibles depuis CPAN.

DB_File

```
use DB_File;
```

Lie avec `tie` un hachage à un fichier dans le style de DBM :

```
tie(%hachage, "DB_File", $fichier)    # Ouvre une base de données.
    or die "Impossible d'ouvrir $fichier : $!";

$v = $hachage{"clef"};                # Extrait depuis la base de données.
$hachage{"clef"} = "valeur";          # Positionne une valeur dans la base.
untie %hachage;
```

Lie avec `tie` un hachage à un fichier B-tree (arbre balancé ou équilibré), mais continue d'y accéder en tant que hachage DBM ordinaire :

```
tie(%hachage, "DB_File", "mon_arbre", O_RDWR|O_CREAT, 0666, $DB_BTREE)
    or die "Impossible d'ouvrir le fichier 'mon_arbre' : $!";

while (($c, $v) = each %hachage) {    # Parcourt dans l'ordre
    print "$c => $v\n";
}
```

Lie avec `tie` un tableau à un fichier texte brut :

```
tie(@lignes, "DB_File", $fic_texte, O_RDWR|O_CREAT, 0666, $DB_NUM_ENREG)
    or die "Impossible d'ouvrir le fichier texte $fic_texte : $!";

# Écrit quelques lignes dans le fichier, en écrasant les anciennes.
$lignes[0] = "première ligne";
$lignes[1] = "deuxième ligne";
$lignes[2] = "troisième ligne";

push @lignes, "pénultième", "dernière";   # Met deux lignes à la fin
                                          # du fichier
```

```
$wc = scalar @lignes;                    # Compte le nombre de lignes
$derniere = pop @lignes;                 # Supprime et extrait la dernière
                                         # ligne
```

Le module `DB_File` donne un accès lié avec `tie` à une DB Berkeley.[1] La fonction `tie` vous donne par défaut une base de données standard dans le style de DBM, avec certaines fonctionnalités que l'on ne retrouve dans aucune bibliothèque DBM : il n'existe pas de limites de taille ni pour les clefs, ni pour les valeurs et vos données sont stockées dans un format indépendant de l'ordre des octets.

Le deuxième mécanisme de `tie` utilise des B-trees (arbres balancés ou équilibrés) pour vous donner un véritable fichier ISAM (*Indexed Sequential Access Method*, méthode d'accès séquentiel indexé), c'est-à-dire un hachage dont les clefs sont automatiquement triées dans l'ordre alphabétique par défaut, mais configurable par l'utilisateur.

Le troisième mécanisme de `tie` lie un tableau à un fichier d'enregistrements (des lignes de texte par défaut) de façon à ce que les modifications dans le tableau se répercutent automatiquement sur le disque. Ceci simule un accès aléatoire par numéro de ligne dans un fichier texte ordinaire. L'interface standard est conforme à la version 1.x de Berkeley DB ; si vous souhaitez employer les nouvelles fonctionnalités disponibles dans Berkeley 2.x ou 3.x, utilisez plutôt le module `BerkeleyDB` de CPAN.

À partir de la version 2.x, Berkeley DB supporte intrinsèquement le verrouillage, ce qui n'était pas le cas dans les versions précédentes. Voir la section *Verrouillage de fichier* du chapitre 16 pour un description de la manière sûre dont vous pouvez verrouillez n'importe quel type de fichier de base de données en utilisant `flock` sur un fichier sémaphore.

Dumpvalue

```
use Dumpvalue;

Dumpvalue->new->dumpValue($magasin);
```

Lorsqu'on l'utilise sur l'exemple tiré de la définition de `Class::Struct`, ceci affiche :

```
'Boutique::adrs' => ARRAY(0x816d548)
    0 'Miskatonic University'
    1 'Innsmouth, Mass.'
'Boutique::patron' => Chef=HASH(0x80f1538)
    'Chef::debut' => 'Fri Apr 13 19:33:04 2001'
    'Chef::nom' => 'Dr. L. P. Hache'
    'Chef::salaire' => 'folie'
'Boutique::proprio' => 'Abdul Alhazred'
'Boutique::stock' => HASH(0x81976d0)
    'charmes' => 3
    'livres' => 208
    'potions' => 'aucune'
```

Il s'agit d'un autre module facilitant l'affichage de données complexes. Il n'est pas tant

1. Pourvu que vous ayez la bibliothèque installée sur votre système. Sinon, vous pouvez la compiler et l'installer assez facilement.

destiné au formatage de présentation (*marshalling*) qu'à un affichage agréable. Il est utilisé par la commande x du débogueur Perl. En tant que tel, il offre une quantité impressionnante d'options pour contrôler le format de sortie. Il donne également accès aux tables de symboles de paquetages de Perl pour vider le contenu d'un paquetage entier.

English

```
use English;

# Utilise des noms dans le style de awk
$RS = '';                   # au lieu de $/
while (<>) {
    next id $NR < 10;       # au lieu de $.
    ...
}

# Idem, mais encore plus long à la Cobol
$INPUT_RECORD_SEPARATOR = '';
while (<>) {
    next if $INPUT_LINE_NUMBER < 10;
    ...
}
```

Le module English fournit des alias encombrants pour les variables internes à destination des prodigieux dactylos qui ont une haine viscérale des identificateurs non-alphabétiques (et un *amour* viscéral pour la touche Blocage Majuscules, ou *Caps Lock*). Comme avec toutes les importations, ces alias ne sont disponibles que dans le paquetage courant. Les variables sont toujours accessibles sous leur véritable noms. Par exemple, un fois que vous avez fait un use de ce module, vous pouvez utiliser $PID si $$ vous dérange ou $PROGRAM_NAME si $0 vous donne mal au cœur. Certaines variables ont plus d'un alias. Voir le chapitre 28, *Noms spéciaux* pour une description complète de toutes les variables internes et de leur alias en anglais tirés du module English.

Errno

```
use Errno;
unless (open(HF, $chemin)) {
    if ($!{ENOENT}) {        # Nous n'avons pas besoin d'importer ceci !
        warn "$chemin n'existe pas\n";
    else {
        warn "Échec de l'ouverture de $chemin : $!";
    }
}

use Errno qw(EINTR EIO :POSIX)
if ($! == ENOENT) { ... }
```

Le module Errno rend accessibles les noms symboliques pour l'ensemble des valeurs de statut d'erreur lorsqu'un appel système échoue, mais il ne les exporte pas toutes par dé-

faut. Le module n'a qu'un seul tag d'exportation, « `:POSIX` », qui n'exporte que les symboles définis dans le standard POSIX 1003.1. Le module rend également magique la variable globale `%!` en utilisant `tie`. Vous pouvez indicer le hachage `%!` en employant n'importe quelle valeur valide d'*errno* sur votre système, pas seulement les valeurs POSIX, et sa valeur n'est vraie que s'il s'agit de l'erreur actuelle.

Exporter

Dans votre fichier *Mon_Module.pm* :

```
package Mon_Module;

use strict;
use Exporter;

our $VERSION = 1.00;            # Ou supérieure...
our @ISA = qw(Exporter);

our @EXPORT      = qw(f1 %h);    # Symboles importés par défaut.
our @EXPORT_OK   = qw(f2 f3);    # Symboles importés seulement sur demande.
our %EXPORT_TAGS = (             # Mappage pour :raccourcis.
    a => [qw(f1 f2 f3)],
    b => [qw(f2 %h)],
);

# Votre code ici.

1;
```

Depuis un programme ou un autre module qui fait un `use` du vôtre :

```
use Mon_Module;                  # Importe tout ce qu'il a dans @Export.
use Mon_Module ();               # Charge le module sans rien importer.
use Mon_Module "f1", "f2", "%h"; # Deux fonctions et une variable.
use Mon_Module qw(:DEFAULT f3);  # Tout ce qu'il y a dans @Export +
                                 # une fonction.
use Mon_Module "f4";             # Erreur fatale car f4 n'est pas exportée.
```

Dès que quelqu'un invoque une déclaration `use` pour charger votre module, il appelle la méthode `import` depuis votre module pour rapatrier tous les symboles dont il a besoin dans le paquetage de l'appelant. Votre module (celui qui fait les exportations) peut définir la méthode `import` comme il le souhaite, mais la manière standard est d'en hériter à partir du module de classe `Exporter`. C'est ce que fait le code ci-dessus.

Le module `Exporter` sert de classe de base pour les modules qui souhaitent établir leur propres exportations. Curieusement, les modules orientés objet n'utilisent généralement pas `Exporter`, puisqu'ils n'exportent normalement rien (les appels de méthodes n'ont pas besoin d'être exportés). Toutefois, le module `Exporter` lui-même est accessible de manière orientée objet à cause du tableau `@ISA` que vous avez installé, comme dans notre exemple. Lorsqu'un autre programme ou qu'un autre module fait un `use` de votre module, vous utiliserez automatiquement la méthode `Exporter::import` par le biais de l'héritage.

Le tableau @EXPORT du module contient une liste de symboles (des fonctions et même des variables) que le code appelant importe automatiquement avec une instruction use sans rien d'autre. Le tableau @EXPORT_OK est composé des symboles qui peuvent être importés si on les demande nommément. Le numéro $VERSION est consulté si l'instruction use demande une version particulière (ou plus récente) du module. Beaucoup, beaucoup d'autres fonctionnalités sont disponibles. Voir le chapitre 11, *Modules*, ainsi que la page de man en ligne du module Exporter.

Fatal

Le module Fatal fait échouer les fonctions plus spectaculairement. Il remplace les fonctions qui normalement retournent faux en cas d'échec avec des sur-couches qui lèvent une exception si la véritable fonction renvoie faux. Ainsi, vous pouvez utiliser en toute sûreté ces fonctions sans tester explicitement leur valeur de retour à chaque appel.

Les fonctions définies par l'utilisateur et les fonctions internes peuvent toutes être enrobées, sauf les fonctions internes qui ne peuvent pas être exprimées par des prototypes. Une tentative de supplanter une fonction interne non-enrobable lève une exception. Ceci inclut system, print, printf, exec, split, grep et map ou plus généralement, toute *FONC* pour laquelle prototype "CORE::*FONC*" renvoie faux, y compris la fonction prototype elle-même.

Si le symbole :void apparaît dans la liste d'importation, les fonctions citées ensuite dans la liste se limitent à lever une exception lorsque la fonction est appelée dans un contexte void c'est-à-dire, lorsque la valeur renvoyée est ignorée. (Faites attention à la dernière instruction dans un sous-programme.) Par exemple :

```
use Fatal qw(:void open close);

# open proprement vérifié, aucune exception n'est donc levée en cas d'échec.
if (open(HF, "< /rien_de_tel") { warn "aucun /rien de tel : $!"; }

# close improprement vérifié, une exception est donc levée en cas d'échec.
close HF;
```

Fcntl

```
use Fcntl;                # Importe les constantes standard de fcntl.h.
use Fcntl ":flock";       # Importe les constantes LOCK_*.
use Fcntl ":seek";        # Importe SEEK_CUR, SEEK_SET et SEEK_END.
use Fcntl ":mode";        # Importe les constantes de test de stat S_*.
use Fcntl ":Fcompat";     # Importe les constantes F*.
```

Le module Fcntl fournit des constantes à employer avec diverses fonctions internes de Perl. L'ensemble importé par défaut inclut des constantes comme F_GETFL et F_SETFL pour fcntl, SEEK_SET et SEEK_END pour seek et sysseek et O_CREAT et O_EXCL pour sysopen. Les tags d'importation supportés incluent « :flock » pour accéder aux constantes LOCK_EX, LOCK_NB, LOCK_SH et LOCK_UN pour flock ; « :mode » pour aller chercher les constantes de *sys/stat.h* comme S_IRUSR et S_ISFIFO ; « :seek » pour obtenir les trois arguments de seek et sysseek ; et « :Fcompat » pour avoir les symboles désuets commençant par un « F » mais pas par« F_ », comme FAPPEND, FASYNC et FNONBLOCK. Voir la

documentation en ligne du module `Fcntl` et la documentation de votre système d'exploitation pour les appels systèmes concernés, comme *fcntl*(2), *lseek*(2), *open*(2) et *stat*(2).

File::Basename

```
use File::Basename;

$nom_complet = "/usr/local/src/perl-5.6.1.tar.gz";

$fichier = basename($nom_complet);
# fichier= "perl-5.6.1.tar.gz"

$rep = dirname($nom_complet);
rep="/usr/local/src"

($fichier,$rep,$ext) = fileparse($nom_complet, qr/\..*/);
# rep="/usr/local/src/" fichier="perl-5" ext=".6.1.tar.gz"

($fichier,$rep,$ext) = fileparse($nom_complet, qr/\.[^.]*/);
# rep="/usr/local/src/" fichier="perl-5.6.1.tar" ext=".gz"

($fichier,$rep,$ext) = fileparse($nom_complet, qr/\.\D.*/);
# rep="/usr/local/src/" fichier="perl-5.6.1" ext=".tar.gz"

($fichier,$rep,$bak) = fileparse("/tmp/fichier.bak",
                           qr/~+$/, qr/\.(bak|orig|save)/)
# rep="/tmp/" fichier="fichier" bak=".bak";

($fichier,$rep,$bak) = fileparse("/tmp/fichier~",
                           qr/~+$/, qr/\.(bak|orig|save)/)
# rep="/tmp/" fichier="fichier" bak="~";
```

Le module `File::Basename` fournit des fonctions pour décomposer les noms de chemin en leurs composants individuels. La fonction `dirname` sort la portion représentant le répertoire et `basename` la portion ne représentant pas le répertoire. La fonction plus élaborée `fileparse` distingue dans le nom du chemin complet, le nom du répertoire, le nom du fichier et le suffixe ; vous devez fournir une liste d'expressions régulières décrivant les suffixes qui vous intéressent. Les exemples ci-dessus illustrent comment le choix de motifs pour le suffixe affecte le résultat. Par défaut, ces fonctions analysent des noms de chemin selon les conventions natives de votre plate-forme. La fonction `fileparse_set_fstype` sélectionne les règles d'analyse pour une plate-forme différente, comme `fileparse_set_fstype("VMS")` pour analyser les noms utilisant les règles de VMS, même lorsque l'on tourne sur des systèmes non-VMS.

File::Compare

```
use File::Compare;

printf "fichier_A et fichier_B sont %s.\n",
```

```
    compare("fichier_A","fichier_B") ? "différents" : "identiques";

use File::Compare 'cmp';
sub arranger($) {
    my $ligne = $_[0];
    for ($ligne) {
        s/\^s+//;     # Coupe les espaces au début.
        s/\s+$//;     # Coupe les espaces à la fin.
    }
    return uc($ligne);
}

if (not cmp("fichier_A", "fichier_B",
            sub {arranger $_[0] eq arranger $_[1]} ) {
    print "fichier_A et fichier_B sont à peu près pareils.\n";
}
```

Le module `File::Compare` fournit une fonction, `compare`, qui compare le contenu des deux fichiers qu'on lui passe. Elle renvoie `0` si les fichiers contiennent les mêmes données, `1` s'ils contiennent des données différentes et `-1` si l'on a rencontré une erreur en accédant les fichiers cités. Si vous passez comme troisième argument une référence de sous-programme, cette fonction est appelée répétitivement pour déterminer si deux lignes sont équivalentes. Pour assurer la compatibilité avec le programme *cmp*(1), vous pouvez explicitement importer la fonction en tant que `cmp`. (Ceci n'affecte pas l'opérateur binaire `cmp`.)

File::Copy

```
use File::Copy;

copy("/tmp/fichier_A", "/tmp/fichier_A.orig") or die "copie échouée : $!";
copy("/etc/motd", *STDOUT)                    or die "copie échouée : $!";
move("/tmp/fichier_A", "/tmp/fichier_A.orig") or die "déplacement échoué : $!";

use File::Copy qw/cp mv/;                 # Importe les noms Unix habituels.
cp("/tmp/fichier_A", "/tmp/fichier_A.orig")   or die "copie échouée : $!";
mv("/tmp/fichier_A", "/tmp/fichier_A.orig")   or die "déplacement échoué : $!";
```

Le module `File::Copy` exporte deux fonctions, `copy` et `move`, qui respectivement copie ou renomme leur premier argument vers leur second, de la même manière qu'un appel aux commandes Unix *cp*(1) et *mv*(1) (noms que vous pouvez utiliser si vous les importez explicitement). La fonction `copy` accepte également comme arguments des handles de fichiers. Ces fonctions renvoient vrai lorsqu'elle fonctionnent et faux lorsqu'elles échouent, en positionnant `$!` (`$OS_ERROR`) de manière appropriée. (Malheureusement, vous ne pouvez pas dire si quelque chose comme « Permission denied » s'applique au premier fichier ou au second.) Ces fonctions sont en quelque sorte un compromis entre le confort et la précision. Elles ne supportent pas les nombreuses options et optimisations de *cp*(1) et *mv*(1), comme la copie récursive, la copie de sauvegarde automatique, la conservation des dates et heures originales et des informations sur les propriétaires des fichiers, ou la confirmation interactive. Si vous avez besoin de ces fonctionnalités, il

vaut certainement mieux appeler les versions de votre plate-forme pour ces commandes.[2] Gardez bien à l'esprit que tous les systèmes ne supportent pas les mêmes commandes ou n'emploient pas les mêmes options pour ces commandes.

```
system("cp -R -pi /tmp/rep1 /tmp/rep2) == 0
    or die "le statut de la commande externe cp est $?";
```

File::Find

```
use File::Find;

# Affiche tous les répertoires en-dessous du répertoire courant.
find sub { print "$File::Find::name\n" if -d }, ".";

# Calcule l'espace total utilisé par les fichiers des répertoires listés.
@reps = @ARGV ? @ARGV : ('.');
my $tot = 0;
find sub { $tot += -s }, @reps;
print "@reps contiennent $tot octets\n";

# Modifie le comportement par défaut pour traverser les liens
# symboliques et visiter les sous-répertoires en premier
find { wanted => \&ma_fonction, follow => 1, bydepth=>1 }, ".";
```

La fonction `find` du module `File::Find` descend récursivement dans des répertoires. Son premier argument doit être une référence à une fonction et tous les arguments suivants doivent être des répertoires. La fonction est appelée pour chaque fichier des répertoires listés. À l'intérieur de cette fonction, la variable `$_` est positionnée avec le nom de base du fichier actuellement visité et le répertoire de travail courant du processus est par défaut mis à ce répertoire. La variable de paquetage `$File::Find::name` est le nom de chemin complet du fichier visité. Une autre convention d'appel prend comme premier argument une référence à un hachage contenant des spécifications d'options, incluant « `wanted` », « `bydepth` », « `follow` », « `follow_fast` », « `follow_skip` », « `no_chdir` », « `untaint` », « `untaint_pattern` » et « `untaint_skip` », comme le détaille complètement la documentation en ligne. Ce module est également utilisé par le programme de conversion standard *find2perl*(1) qui vient avec la distribution Perl.

File::Glob

```
use File::Glob ':glob';      # Supplante le glob interne.
@liste = <*.[Cchy]>;         # Utilise maintenant le glob POSIX, pas le
                             # glob csh.

use File::Glob qw(:glob csh_glob);
```

2. Ou aller chercher les versions PPT si votre plate-forme est un défi pour ces outils. *N.d.T. : Perl Power Tools : The Unix Reconstruction Project*, un projet dont le but avoué est : « Notre objectif est, de manière assez simple, de réimplémenter le jeu de commandes classiques Unix en Perl pur, et de nous amuser autant que possible en faisant cela. » .

```
@sources = bsd_glob("*.{C,c,h,y,pm,xs}", GLOB_CSH);
@sources = csh_glob("*.{C,c,h,y,pm,xs}");        # (idem)

use File::Glob ':glob';
# appelle glob avec des arguments supplémentaires
$rep_maison = bsd_glob('~hackerjr', GLOB_TILDE | GLOB_ERR);
if (GLOB_ERROR) {
    # Une erreur s'est produite en développant le répertoire maison
}
```

La fonction `bsd_glob` du module `File::Glob` implémente la routine *glob*(3) de la librairie C. Un second argument facultatif contient des options gouvernant des propriétés supplémentaires de correspondances. Le tag d'importation `:glob` importe à la fois la fonction et les options nécessaires.

Le module implémente également une fonction `csh_glob`. C'est elle que l'opérateur interne de Perl `glob` et l'opérateur de globalisation de fichiers *`GLOBPAT`* appellent réellement. Appeler `csh_glob` revient (en grande partie) à appeler `bsd_glob` de cette manière :

```
bsd_glob(@_ ? $_[0] : $_,
    GLOB_BRACE | GLOB_NOMAGIC | GLOB_QUOTE | GLOB_TILDE);
```

Si vous importez le tag `:glob`, tous les appels aux opérateurs internes de globalisation de fichiers dans le paquetage courant appelleront alors réellement la fonction du module `bsd_glob` à la place de la fonction `csh_glob`. Une raison pour laquelle vous voudriez faire cela, bien que `bsd_glob` s'occupe de motifs comprenant des espaces, est que `csh_glob` s'en occupe d'une manière, heu, historique. Les anciens scripts écriraient `<*.c *.h>` pour globaliser les deux motifs. De toute façon, aucune des fonctions n'est gênée par des espaces dans les véritables noms de fichiers.

La fonction `bsd_glob` prend un argument contenant le motif de globalisation de fichiers (et pas un motif d'expression régulière) plus un argument facultatif pour les options. Les noms de fichiers commençant par un point ne correspondent pas à moins de le demander spécifiquement. La valeur de retour est influencée par les options du second argument, qui doivent être des bits joints entre eux par des OU :[3]

`GLOB_BRACE`

Traite en amont la chaîne pour développer les chaînes `{motif,motif,...}` comme *csh*(1) le ferait. Le motif `{}` n'est pas développé pour des raisons historiques, surtout pour faciliter la saisie des motifs de *find*(1).

`GLOB_CSH`

Synonyme de `GLOB_BRACE | GLOB_NOMAGIC | GLOB_QUOTE | GLOB_TILDE`

`GLOB_ERR`

Renvoie une erreur lorsque la fonction `bsd_glob` rencontre un répertoire qu'elle ne peut ouvrir ou lire. D'ordinaire, `bsd_glob` saute les erreurs pour rechercher d'autres correspondances.

3. À cause des restrictions de syntaxe dans l'opérateur interne `glob`, il se peut que vous deviez appeler la fonction en tant que `bsd_glob` si vous voulez lui passer le second argument.

GLOB_MARK

Ajoute un slash à la fin des répertoires.

GLOB_NOCASE

Par défaut, les noms de fichiers sont sensibles à la casse (majuscules/minuscules) ; cette options fait que bsd_glob ignore les différences de casse. (Mais voir ci-dessous les exceptions sur les systèmes à la MS-DOS).

GLOB_NOCHECK

Si le motif ne correspond à aucun nom de chemin, alors bsd_glob renvoie une liste uniquement constituée du motif, comme le fait */bin/sh*. Si l'option GLOB_QUOTE est positionnée, les conséquences qu'elle entraîne sont présentes dans le motif renvoyé.

GLOB_NOMAGIC

Idem que GLOB_NOCHECK mais le motif n'est renvoyé que s'il ne contient aucun des caractères spéciaux *, ? ou [. NOMAGIC est là pour simplifier l'implémentation du comportement historique de globalisation de *csh*(1) et ne devrait probablement pas être utilisé ailleurs.

GLOB_NOSORT

Par défaut, les noms de chemins sont triés dans un ordre croissant (en utilisant les comparaisons de caractères habituelles, sans respecter les valeurs des locales). Cette option évite ce tri avec comme résultat une légère amélioration de la vitesse.

GLOB_QUOTE

Utilise le caractère antislash \ pour les protections : chaque occurrence dans le motif d'un antislash suivi par un caractère est remplacée par ce caractère, évitant les interprétations spéciales du caractère en question. (Mais voir ci-dessous les exceptions sur les systèmes à la MS-DOS).

GLOB_TILDE

Autorise l'emploi de motifs dont le premier composant du chemin est *~UTILISATEUR*. Si *UTILISATEUR* est omis, le tilde lui-même (ou suivi par un slash) représente le répertoire maison de l'utilisateur courant.

La fonction bsd_glob renvoie une liste (qui peut être vide) des chemins correspondants, cette liste pouvant devenir marquée si cela importe à votre programme. En cas d'erreur, GLOB_ERROR sera vrai et $! ($OS_ERROR)sera positionnée avec l'erreur système standard. Il est granti que GLOB_ERROR sera faux s'il ne se passe aucune erreur, et de valoir soit GLOB_ABEND, sinon GLOB_NOSPACE. (GLOB_ABEND signifie que la fonction bsd_glob s'est arrêtée à cause d'une erreur et GLOB_NOSPACE à cause d'un manque de mémoire.) Si bsd_glob a déjà trouvé des chemins correspondant lorsque l'erreur se produit, elle renvoie la liste des noms de fichiers trouvés jusque là *mais positionne aussi* GLOB_ERROR. Remarquez que cette implémentation de bsd_glob diffère de la plupart des autres en ce qu'elle ne considère pas ENOENT et ENOTDIR comme des conditions d'arrêt sur erreur. À la place, elle poursuit son traitement malgré ces erreurs, sauf si l'option GLOB_ERR est positionnée.

Si aucune option n'est donnée, les comportements par défaut de votre système sont respectés, c'est-à-dire que les noms de fichiers qui diffèrent seulement par leur casse seront confondus sur VMS, OS/2, les vieux Mac OS (mais pas Mac OS X) et les systèmes Micro-

soft (mais pas lorsque Perl a été compilé avec Cygwin). Si vous ne donnez aucune option et que vous voulez toujours que ce comportement s'applique, vous devez alors inclure `GLOB_NOCASE` dans les options. Quelque soit le système sur lequel vous êtes, vous pouvez modifier votre comportement par défaut en important à l'avance les options `:case` ou `:nocase`.

Sur les systèmes à la MS-DOS, l'antislash est un caractère séparateur de répertoires valide.[4] Dans ce cas, l'emploi d'un antislash comme caractère de protection (*via* `GLOB_QUOTE`) interfère avec l'emploi de l'antislash en tant que séparateur de répertoires. La meilleure solution (la plus simple et la plus portable) est d'utiliser des slashs comme séparateurs de répertoires, des antislashs pour les protections. De toute façon, cela n'a rien à voir avec les attentes de certains utilisateurs, ainsi les antislashs (sous `GLOB_QUOTE`) ne protègent que les métacaractères de globalisation [,], {, }, -, ~ et \ lui-même.Tous les autres antislashs sont passés inchangés, si vous parvenez à en avoir avec le propre mécanisme de Perl pour la protection par antislashs dans les chaînes. Cela peut prendre jusqu'à quatre antislashs pour finalement correspondre à un seul dans le système de fichier. Ceci est tellement complètement délirant que même les utilisateurs des systèmes à la MS-DOS devraient sérieusement penser à employer des slashs. Si vous voulez vraiment utiliser des antislashs, jetez un œil au module standard `File::DosGlob` qui est plus susceptible de correspondre à vos goûts que la globalisation de fichiers à la mode Unix.

File::Spec

```
use File::Spec;    # style orienté objet

$chemin = File::Spec->catfile("sous_rep", "fichier");
    # 'sous_rep/fichier' sur Unix, OS2 ou Mac OS X
    # 'sous_rep:fichier' sur les (vieux) Macs d'Apple
    # 'sous_rep\fichier' sur Microsoft

$chemin = File::Spec->catfile("", "rep1", "rep2", "fichier");
    # '/rep1/rep2/fichier' sur Unix, OS2 ou Mac OS X
    # ':rep1:rep2:fichier' sur les (vieux) Macs d'Apple
    # '\rep1\rep2\fichier' sur Microsoft

use File::Spec::Unix;
$chemin = File::Spec->catfile("sous_rep", "fichier");
    # 'sous_rep/fichier' (même si exécuté sur des systèmes non-Unix)

use File::Spec::Mac;
$chemin = File::Spec->catfile("sous_rep", "fichier");
    # 'sous_rep:fichier'

use File::Spec::Win32;
```

4. Bien que techniquement, le slash aussi — du moins aussi longtemps que cela concerne les noyaux et les appels systèmes ; les commandes shell sont remarquablement moins bien éclairées.

```
$chemin = File::Spec->catfile("sous_rep", "fichier");
    # 'sous_rep\fichier'

# Utilise plutôt l'interface fonctionnelle.
use File::Spec::Functions;
$chemin = catfile("sous_rep", "fichier");
```

La famille de modules `File::Spec` vous permet de construire des chemins utilisant des répertoires et des noms de fichiers sans coder en dur les séparateurs de répertoires spécifiques aux plates-formes. Les systèmes supportés incluent Unix, VMS, Mac et Win32. Ces modules offrent tous une méthode de classe `catfile` qui concatène chaque composant de chemin en utilisant le séparateur de chemin spécifique à la plate-forme. Le module `File::Spec` renvoie des résultats différents selon votre plate-forme courante. Les autres renvoient des résultat spécifique à la plate-forme en question. Le module `File::Spec::Functions` fournit une interface fonctionnelle.

File::stat

```
use File::stat;
$st = stat($fichier)    or die "stat de $fichier impossible : $!";
if ($st->mode & 0111 and $st->nlink > 1) {
    print "$fichier est exécutable et possède plusieurs liens\n";
}

use File::stat ":FIELDS";
stat($fichier)          or die "stat de $fichier impossible : $!";
if ($st_mode & 0111 and $st_nlink > 1) {
    print "$fichier est exécutable et possède plusieurs liens\n";
}

@infos_stat = CORE::stat($fichier);      # Utilise la fonction interne,
                                         # même si elle est supplantée
```

Le module `File::stat` fournit une interface méthode aux fonctions internes Perl `stat` et `lstat` en les remplaçant par des versions renvoyant une objet `File::stat` (ou `undef` en cas d'échec). Cet objet possède des méthodes nommées de manière à renvoyer le champ de la structure du même nom depuis l'appel système habituel *stat*(2) ; il s'agit de `dev`, `ino`, `mode`, `nlink`, `uid`, `gid`, `rdev`, `size`, `atime`, `mtime`, `ctime`, `blksize` et `blocks`. Vous pouvez également importer les champs de la structure dans votre propre espace de noms en tant que variables ordinairex en utilisant le tag d'importation « `:FIELDS` ». (Les opérateurs internes `stat` et `lstat` continuent à être supplantés.) Ces champs se présentent comme des variables scalaires avec un « `st_` » au début du nom du champ. C'est-à-dire que la variable `$st_dev` correspond à la méthode `$st->dev`.

File::Temp

```
use File::Temp qw(tempfile tempdir);

$rep = tempdir(CLEANUP => 1);
($hf, $fichier) = tempfile(DIR => $rep);
```

```
($hf, $fichier) = tempfile($exemple, DIR => $rep);
($hf, $fichier) = tempfile($exemple, SUFFIX => ".data");
$hf = tempfile();

use File::Temp ':mktemp';

($hf, $fichier) = mkstemp("fic_tmpXXXXX");
($hf, $fichier) = mkstemps("fic_tmpXXXXX", $suffixe);
$rep_tmp = mkdtemp($exemple);
$fichier_non_ouvert = mktemp($exemple);
```

Nouveauté dans la version 5.6.1 de Perl, le module `File::Temp` fournit des fonctions pratiques pour créer et ouvrir de manière sûre des fichiers temporaires. Il vaut mieux utiliser ce module qu'essayer de choisir vous-même un fichier temporaire. Sinon, vous tomberez dans les mêmes pièges que tout le monde avant vous. Ce module vous protège aussi bien des situations de concurrence (*race conditions*) que du danger d'employer des répertoires où d'autres peuvent écrire ; voir le chapitre 23, *Sécurité*. La fonction `tempfile` renvoie à la fois un handle de fichier et un nom de fichier. Le plus sûr est d'utiliser ce qui est déjà ouvert et d'ignorer complètement le nom de fichier (excepté peut-être pour les messages d'erreurs). Une fois que le fichier est fermé, il est automatiquement supprimé. Pour assurer une compatibilité avec la bibliothèque C, le tag d'importation `:mktemp` donne un accès aux fonctions dont les noms sont familiers aux programmeurs C, mais s'il vous plaît, souvenez-vous que les noms de fichiers sont toujours moins sûrs que les handles de fichiers.

FileHandle

```
use FileHandle;

$hf = new FileHandle;
if ($hf->open("< fichier")) {
    print $ligne while defined($ligne = $hf->getline);
    $hf->close;
}

$pos = $hf->getpos; # comme tell()
$hf->setpos($pos); # comme seek()

($hf_lec, $hf_ecr) = FileHandle::pipe();

autoflush STDOUT, 1;
```

Le module `FileHandle` sert surtout de mécanisme pour dissimuler les variables de Perl contenant des ponctuations sous des appellations plus longues, davantage dans le style orienté objet. Il est fourni pour assurer une compatibilité avec des versions antérieures mais maintenant il s'agit véritablement d'un frontal vers plusieurs modules plus spécifiques, comme `IO::Handle` et `IO::File`.[5] Sa meilleure propriété est l'accès de bas niveau

5. Comme il charge beaucoup de code, ce module vous coûte environ un méga-octet en mémoire.

qu'il fournit vers certaine fonctions rares de la bibliothèque C (*clearerr*(3), *fgetpos*(3), *fsetpos*(3) et *setvbuf* (3)).

Variable	Méthode
$|	autoflush
$,	output_field_separator
$\	output_record_separator
$/	input_record_separator
$.	input_line_number
$%	format_page_number
$=	format_lines_per_page
$-	format_lines_left
$~	format_name
$^	format_top_name
$:	format_line_break_characters
$^L	format_formfeed

Au lieu d'écrire :

```
$hfs = select(HANDLE );
$~ = 'UnFormat';
$| = 1;
select($hfs);
```

vous pouvez juste écrire :

```
use FileHandle;
Handle->format_name('UnFormat');
Handle->autoflush(1);
```

Actuellement, trois méthodes (`output_field_separator`, `output_record_separator` et `input_record_separator`) ne font que prétendre être des méthodes par handle : les positionner sur un handle affecte tous les handles de fichiers. Elles sont ainsi seulement supportées en tant que méthodes de classes, pas en tant que méthodes par handle de fichier. Cette restriction sera peut-être abandonnée un jour.

Pour obtenir un handle de fichier de portée lexicale, au lieu d'utiliser l'autovivifiation de handle de fichier :

```
open my $hf, "< un_fichier"
    or die "impossible d'ouvrir un_fichier : $!";
```

on peut écrire :

```
use FileHandle;
my $hf = FileHandle->new("< un_fichier")
    or die "impossible d'ouvrir un_fichier : $!";
```

`FileHandle` hérite de `IO::File`, qui lui-même hérite de `IO::Handle` et `IO::Seekable`. Les fonctionnalités du module sont virtuellement toutes accessibles plus efficacement *via* les appels purs et basiques de Perl, sauf pour les suivants, qui peuvent ne pas tous être implémentés sur les plates-formes non-Unix :

HANDLE->blocking(*EXPR*)

Appelé avec un argument, active les entrées/sorties non-bloquantes si l'argument est faux et désactive les entrées/sorties non-bloquantes (c'est-à-dire active les entrées/sorties bloquantes) s'il est vrai. La méthode renvoie la précédente valeur positionnée (ce qui est toujours le comportement actuel si aucun argument n'est donné). En cas d'erreur, `blocking` positionne `$!` et renvoie `undef`. Ceci pourrait être accompli avec `fcntl` directement mais l'interface `FileHandle` est plus simple à utiliser.

HANDLE->clearerr

Appelle la fonction *clearerr*(3) de la bibliothèque C pour efface les indicateurs internes du handle pour la fin de fichier et le statut d'erreur.

HANDLE->error

Appelle la fonction *ferror*(3) de la bibliothèque C pour tester l'indicateur d'erreur pour le handle donné, en renvoyant si cet indicateur interne est positionné ou non. L'indicateur d'erreur ne peut être réinitialisé fiablement que *via* la méthode `clearerr`. (Certains systèmes le réinitialisent également en appelant l'opérateur `seek`.)

HANDLE->formline(*IMAGE*, *LISTE*)

Cela revient au même que de sauvegarder l'ancienne variable d'accumulateur (`$^A`), d'appeler la fonction `formline` avec l'*IMAGE* et la *LISTE* données, d'afficher le contenu résultant de l'accumulateur vers le handle donné et enfin de restaurer l'accumulateur original. Par exemple, voici comment afficher une variable représentant un texte long en allant à la ligne à la colonne 72 :

```
use FileHandle;
STDOUT->formline("^" . ("<" x 72) . "~~\n", $texte_long);
```

HANDLE->getpos

Appelle la fonction *getpos*(3) de la bibliothèque C, offrant ainsi une interface alternative à `tell`. Sur certains systèmes (non-Unix), la valeur renvoyée peut être un objet complexe et `getpos` et `setpos` peuvent être la seul manière de repositionner de manière portable un flux de texte.

HANDLE->new_tmpfile

Appelle la fonction *tmpfile*(3) de la bibliothèque C pour créer un nouveau fichier temporaire ouvert en mode lecture-écriture et renvoie un handle sur ce flux. Sur les systèmes où c'est possible, le fichier temporaire est anonyme, c'est-à-dire qu'il est supprimé avec `unlink` après sa création tout en restant ouvert. Vous devez utiliser cette fonction ou `POSIX::tmpnam` comme décrit dans le module `POSIX` pour créer de manière sûre un fichier temporaire sans vous exposer aux problèmes de sécurités annexes mais néanmoins sérieux avec les situations de concurrence (*N.d.T.* : *race conditions*). Depuis la version 5.6.1 de Perl, le module `File::Temp` est maintenant l'interface recommandée.

HANDLE->setbuf(*TAMPON*)

Appelle la fonction *setbuf* (3) de la bibliothèque C avec la variable *TAMPON* donnée. La méthode passe `undef` pour indiquer une sortie qui n'utilisent pas le mécanisme de tampon. Une variable utilisée comme tampon par `setbuf` ou `setvbuf` *ne doit pas*

être modifiée d'aucune manière jusqu'à ce que le handle soit fermé ou jusqu'à ce que la fonction `setbuf` ou `setvbuf` soit appelée à nouveau. Sinon il peut en résulter une corruption de mémoire qui vous ferait de la peine.

HANDLE`->setpos(`*EXPR*`)`

Appelle la fonction *fsetpos*(3) de la bibliothèque C, offrant ainsi une autre interface à `seek`. L'argument ne doit être qu'une valeur correcte renvoyée par `getpos`, comme décrit précédemment.

HANDLE`->setvbuf(`*TAMPON*`, `*TYPE*`, `*TAILLE*`)`

Appelle la fonction *setvbuf* (3) de la bibliothèque C avec le *TAMPON* donné. Les constantes standards de la bibliothèque C `_IONBF` (sans utilisation d'un tampon), `_IOLBF` (tampon vidé ligne par ligne) et `_IOFBF` (mécanisme complet de tampon) sont disponibles pour le champ *TYPE* si elles sont explicitement importées. Voir la mise en garde dans `setbuf`.

HANDLE`->sync`

Appelle la fonction *fsync*(3) de la bibliothèque C pour synchroniser un fichier dans un état en-mémoire vers le support physique. Remarquez que `sync` n'agit pas sur le handle mais sur le descripteur de fichier, toute donnée conservée dans les tampons ne sera donc pas synchronisée à moins que le tampon ne soit tout d'abord vidé (*N.d.T.* : *flushed*).

HANDLE`->untaint`

Marque le handle de fichier ou de répertoire en lui fournissant une donnée non-marquée. Lorsque l'on tourne en mode marqué (voir le chapitre 23), les données lues depuis des fichiers externes sont considérés comme étant indignes de confiance. N'invoquez pas cette méthode aveuglément : vous court-circuiteriez les meilleures tentatives de Perl pour vous protéger de vous-même.

Getopt::Long

Si votre programme fait :

```
use Getopt::Long;
GetOptions("verbeux"  => \$verbeux,
           "debogage" => \$debogage,
           "sortie=s" => \$sortie);
```

il peut être appelé ainsi depuis la ligne de commande :

```
% prog --verbeux autres arguments ici
% prog --debogage autres arguments ici
% prog -v -d autres arguments ici
% prog --sortie=un_fichier autres arguments ici
% prog -s un_fichier autres arguments ici
```

Le module `Getopt::Long` fournit une fonction `GetOptions` pour traiter les options de la ligne de commande avec des noms assez longs. Il inclut un support pour des choses comme les options en abrégé, les arguments typés comme booléen, chaîne, entier ou à virgule flottante, les variables de tableaux pour les options répétées, les routines de va-

lidation définies par l'utilisateur, le traitement conforme à POSIX ou conforme au style FSF, les options insensibles à la casse et le regroupement traditionnel des options courtes pour ne citer qu'un échantillon de la corne d'abondance de ses fonctionnalités. Si ce module en fait trop, pensez au module plus ordinaire `Getopt::Std` décrit plus loin. Si ce module n'en fait pas assez, regardez le module de CPAN `Getopt::Declare` qui offre une syntaxe plus déclarative pour spécifier les options.

Getopt::Std

```
use Getopt::Std;
```

Vous pouvez utiliser `getopt` et `getopts` avec des variables globales :

```
out ($opt_o, $opt_i, $opt_f);
getopt('oif');       # -o, -i et -f prennent tous des arguments.
                     # Positionne les variables globales $opt_*.
getopts('oif:');     # Maintenant -o et -i sont booléens ; -f prend un arg.
                     # Positionne toujours les variables globales $opt_*
                     # comme effet de bord.
```

Ou vous pouvez les utiliser avec un hachage privé pour les options :

```
my %opts;                 # Nous placerons les résultats ici.
getopt('oif', \%opts);    # Tous les trois prennent toujours un argument.
getopts('oif:', \%opts);  # Maintenant -o et -i sont des booléens
                          # et seul -f prend un argument.
```

Le module `Getopt::Std` fournit deux fonctions, `getopt` et `getopts`, pour vous aider à analyser les arguments de la ligne de commande dans le cas d'options sur un seul caractère. Des deux fonctions, `getopts` est la plus utile car elle vous laisse spécifier que certaines options prennent un argument ; sinon, on s'attend à trouver une option booléenne. Le regroupement d'options standards est supporté. L'ordre ne compte pas, ainsi les options ne prenant pas d'arguments peuvent être regroupées ensemble. Les options qui prennent un argument doivent être les dernières d'un groupe ou être spécifiée à part et leur argument doit venir soit immédiatement après l'option dans la même chaîne, soit comme l'argument suivant du programme. Étant donné l'exemple d'utilisation de `getopts` ci-dessus, voici des appels équivalents :

```
% prog -o -i -f FIC_TEMP autres arguments ici
% prog -o -if FIC_TEMP autres arguments ici
% prog -io -fFIC_TEMP autres arguments ici
% prog -iofFIC_TEMP autres arguments ici
% prog -oifFIC_TEMP autres arguments ici
```

IO::Socket

```
use IO::Socket;
```

En tant que client :

```
$socket = new IO::Socket::INET (PeerAddr => $hote_distant,
                                PeerPort => $port_distant,
                                Proto => "tcp",
```

```
                                    Type => SOCK_STREAM)
    or die "Connexion impossible vers $hote_distant:port_distant : $!\n";

# Ou utilisez l'interface plus simple à un seul argument.
$socket = IO::Socket::INET->new("$hote_distant:$port_distant,");
                                # "localhost:80", par exemple.

print $socket "donnée\n";
$ligne = <$socket>;
```

En tant que serveur :

```
$serveur = IO::Socket::INET->new(LocalPort => port_serveur,
                                 Type => SOCK_STREAM,
                                 Reuse => 1,
                                 Listen => 10) # ou SOMAXCONN
    or die "Impossible d'être un serveur TCP sur le port
            $port_serveur : $!\n";

while ($client = $serveur->accept()) {
    # $client est la nouvelle connexion
    $requete = <$client>;
    print $client "reponse\n";
    close $client;
}

# Simple fonction de connexion TCP renvoyant un handle de fichier
# pour utiliser dans les programmes clients simples.
sub connex_tcp {
    my ($hote, $service) = @_;
    require IO::Socket;
    return IO::Socket::INET->new(join ":", $hote, $service);
}
my $hf    = connex_tcp("localhost, "smtp");    # avec un scalaire
local *HF = connex_tcp("localhost, "smtp");    # avec un handle
```

Le module `IO::Socket` fournit une approche de plus haut niveau pour la gestion des sockets que le module de base `Socket`. Vous pouvez l'utiliser dans un style orienté objet, bien que ce ne soit pas obligatoire, car les valeurs renvoyées sont des handles de fichiers réguliers et peuvent être utilisées en tant que telles, comme le montre la fonction `connex_tcp` dans l'exemple ci-dessus. Ce module hérite ses méthodes de `IO::Handle` et fait lui-même un `require` de `IO::Socket::INET` et `IO::Socket::UNIX`. Voir la description du module `FileHandle` pour d'autres fonctionnalités intéressantes. Voir le chapitre 16 pour une description de l'utilisation des sockets.

IPC::Open2

```
use IPC::Open2;

local(*SA_SORTIE, *SON_ENTREE);     # Crée les handles locaux si besoin.
```

```
$pid_fils = open2(*SA_SORTIE, *SON_ENTREE, $programme, @args)
    or die "Impossible d'ouvrir un pipe vers $programme : $!";
print SON_ENTREE "Voici votre entrée\n";
$sa_sortie = <SA_SORTIE>;
close(SA_SORTIE);
close(SON_ENTREE);
waitpid($pid_fils, 0);
```

La seule fonction exportée, `open2`, du module `IPC::Open2` démarre un autre programme et offre un accès à la fois en lecture et en écriture à cette commande. Les deux premiers arguments doivent être des handles de fichiers valides (ou alors des variables vides dans lesquelles seront placés les handles de fichiers autogénérés). Les arguments restants sont le programme plus ses arguments, qui, s'ils sont passés séparément, ne seront pas interprétés par le shell. Ce module ne récolte pas le processus fils après sa mort. À part les programmes courts où il reste acceptable de laisser le système d'exploitation gérer cela, vous aurez besoin de le faire vous-mêmes. Cela se résume normalement à appeler `waitpid $pid, 0` lorsque vous en avez finit avec ce processus fils. Ne pas faire ceci entraîne l'accumulation de processus défunts (« zombies »).

En pratique, ce module ne fonctionne pas bien avec beaucoup de programmes à cause de la manière dont marche le vidage de tampon dans la bibliothèque d'entrées/sorties standard en C. Si vous contrôlez le code source des deux programmes, vous pouvez toutefois contourner facilement cette restriction en forçant le vidage (*flushing*) de vos tampons de sortie plus fréquemment que par défaut. Si ce n'est pas le cas, les programmes peuvent être ennuyeusement avares dans leur accumulation de données en sortie. Un autre piège potentiel est le verrou mortel (*deadlock*) : si les deux processus sont en train de lire au même moment et qu'aucun n'écrit, alors votre programme se bloquera. Voir le chapitre 16 pour une discussion approfondie.

IPC::Open3

```
use IPC::Open3;

local(*SON_ENTREE, *SA_SORTIE, *SON_ERREUR);

$pid_fils = open3(*SON_ENTREE, *SA_SORTIE, *SON_ERREUR, $cmd, @args);
print SON_ENTREE "trucs\n";
close(SON_ENTREE);                  # Donne une fin de fichier au fils.
@lignes_sortie = <SA_SORTIE>;       # Lit jusqu'à EOF.
@lignes_erreur = <SON_ERREUR>;      # XXX : blocage potentiel si énorme
print "STDOUT:\n", @lignes_sortie, "\n";
print "STDERR:\n", @lignes_erreur, "\n";
close SA_SORTIE;
close SON_ERREUR;
waitpid($pid_fils, 0);
if ($?) {
    print "Ce fils s'est terminé avec comme statut d'attente $?_n";
}
```

Le module `IPC::Open3` fonctionne comme `IPC::Open2` (ce dernier est implémenté dans

les termes du premier), excepté que `open3` donne accès aux handles d'entrée standard, de sortie standard *et* de sortie d'erreurs standard du programme que vous lancez. Les mêmes précautions qu'avec `open2` (voir l'entrée précédente) s'appliquent, plus quelques autres. Au lieu de passer le handle pour la lecture en premier et celui pour l'écriture en second, cette fois-ci c'est l'inverse. De plus, avec `open3`, le danger de verrous mortels (*deadlocks*) est encore plus grand qu'avant. Si vous essayez de lire après la fin de fichier sur l'un des handles de sortie du fils, mais que pendant ce temps il y a beaucoup de trafic en sortie sur l'autre handle, le processus pair semble se bloquer. Utilisez soit la forme à quatre arguments de `select`, soit le module `IO::Select` pour contourner cela. Voir le chapitre 16 pour plus de détails.

Math::BigInt

```
use Math::BigInt;
$i = Math::BigInt->new($chaine);

use Math::BigInt ':constant';
print 2**200;
```

Ceci affiche :

```
+1606938044258990275541962092341162602522202993782792835301376
```

Le module `Math::BigInt` fournit des objets qui représentent des entiers avec une précision arbitraire et des opérateurs arithmétiques surchargés. Créez ces objets en utilisant le constructeur `new` ou, dans une portée lexicale, en important la valeur spéciale « `:constant` », après laquelle tous les littéraux numériques jusqu'à la fin de cette portée lexicale seront traités comme des objets `Math::BigInt`. Tous les opérateurs standards sur les entiers sont implémentés, y compris (depuis la version 5.6 de Perl) les opérateurs logiques de bits. Dans l'implémentation actuelle, ce module n'est pas ce que vous pouvez appeler d'une rapidité fulgurante mais cela pourrait être le cas dans l'avenir. (Nous aimerions voir à quelle vitesse *vous* calculez de tête `2**200`.)

Math::Complex

```
use Math::Complex;

$z = Math::Complex->make(5, 6);
$z = cplx(5, 6);         # idem mais plus court.
$t = 4 - 3*i + $z;       # effectue des opérations complexes standard
print "$t\n";            # affiche 9+3i
print sqrt(-9), "\n";    # affiche 3i
```

Le module `Math::Complex` fournit des objets représentant des nombres complexes avec des opérateurs surchargés. Ce sont des nombres avec une partie réelle et une partie imaginaire comme ceux qui satisfont les racines paires de nombres négatifs, comme montré ci-dessus. À côté des opérateurs arithmétiques, nombre de fonctions internes mathématiques sont également supplantées par des versions qui connaissent les nombres complexes, incluant `abs`, `log`, `sqrt`, `sin`, `cos` et `atan2`. D'autres fonctions fournies sont `Re` et `Im` pour donner les parties réelles et imaginaires des arguments complexes, plus une bat-

terie complète de fonctions trigonométriques étendues, comme `tan`, `asin`, `acos`, `sinh`, `cosh` et `tanh`. Le module exporte également la constante `i`, qui, comme vous pouvez l'imaginer, contient la valeur de *i* ; c'est-à-dire, la racine carrée de -1.

Math::Trig

```
use Math::Trig;

$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);

$pi_sur_deux = pi/2;

$rad = deg2rad(120);
```

Perl ne définit lui-même que trois fonctions trigonométriques : `sin`, `cos` et `atan2`. Le module `Math::Trig` les supplante avec des versions plus agréables et fournit toutes les autres fonctions trigos, incluant `tan`, `csc`, `cosec`, `sec`, `cot`, `cotan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh` et beaucoup d'autres. De plus, la constante `pi` est définie, tout comme les fonctions de conversion comme `deg2rad` et `grad2rad`. Un support est fourni pour les systèmes de coordonnées cartésiennes, cylindriques ou sphériques. Ce module fait un usage implicite de `Math::Complex` si besoin est (et vice-versa) pour les calculs nécessitant des nombres imaginaires.

Net::hostent

```
use Socket;
use Net::hostent;

print inet_ntoa(gethost("www.perl.com")->addr);   # affiche 208.201.239.50
printf "%vd", gethost("www.perl.com")->addr;      # idem

print gethost("127.0.0.1")->name;                 # affiche localhost

use Net::hostent ':FIELDS';
if (gethost($nom_ou_nombre)) {
    print "le nom est $h_name\n";
    print "les alias sont $h_aliases\n";
    print "les adresses sont ",
        join ", " => map { inet_ntoa($_) } @h_addr_list;
}
```

Les exportations par défaut de ce module supplantent les fonctions intrinsèques `gethostbyname` et `gethostbyaddr`, en les remplaçant avec des versions qui renvoient un objet `Net::hostent` (ou `undef` en cas d'échec). Cet objet possède des méthodes accesseurs d'attributs qui renvoient des champs nommés comme ceux de la structure `struct hostent` de la bibliothèque C définie dans *netdb.h* : `name`, `aliases`, `addrtype`, `length` ou `addr_list`. Les méthodes `aliases` et `addr_list` renvoient des références de tableaux ;

les autres renvoient des scalaires. La fonction gethost est un frontal qui fait suivre un argument numérique à gethostbyaddr par le biais de la fonction Socket::inet_aton et le reste à gethostbyname. Comme avec les autres modules semi-pragmatiques qui supplantent des fonctions internes renvoyant des listes, si vous importez le tag « :FIELDS », vous pouvez accéder à des variables de paquetages scalaires ou tabulaires par le même nom que les appels de méthodes préfixés par « h_ ». Les fonctions intrinsèques sont quand même supplantées.

POSIX

```
use POSIX;

# Arrondit des nombres à virgule flottante à l'entier supérieur
# ou inférieur le plus proche.
$n = ceil($n); # arrondi supérieur
$n = floor($n); # arrondi inférieur

# Produit "20-04-2001" pour aujourd'hui.
$ch_date = strftime("%d-%m-%Y"), localtime);

# Produit "vendredi 04/20/01" pour la même date.
$ch_date = strftime(%A %D"), localtime);

# Essaie de nouveaux noms de fichiers temporaires jusqu'à en trouver un qui
# n'existe pas déjà; voir également File::Temp sur CPAN ou dans la v5.6.1.
do {
    $nom = tmpnam();
} until sysopen(HF, $nom, O_CREAT|O_EXCL|O_RDWR, 0666);

# Vérifie si le système a un opérateur chown peu sûr le trahissant.
if (sysconf(_PC_CHOWN_RESTRICTED)) {
    print "Hourra -- seul le super-utilisateur peut appeler chown\n";
}

# Trouve les informations par uname du système courant.
my ($noyau, $hote, $release, $version, $materiel) = uname();

use POSIX ":queue";
while (($pid_mort = waitpid(-1, &WNOHANG)) > 0) {
    # Faites quelque chose avec $pid_mort si vous le voulez.
}

# Devient le leader d'une nouvelle session/nouveau groupe de processus
# (nécessaire pour créer des démons non-affectés par les signaux
# du clavier ou les shells de connexion se terminant).
setsid(0) or die "échec de setsid : $!";
```

Le module POSIX de Perl vous permets d'accéder à tous (ou presque tous) les identificateurs du standard POSIX 1003.1, plus quelques uns du C ANSI que nous ne savions pas

où mettre. Ce module fournit plus de fonctions qu'aucun autre. Voir sa documentation en ligne pour une séance de dissection ou le *POSIX Programmer's Guide* de Donald Lewine (O'Reilly, 1991).

Les identificateurs qui sont des `#defines` sans paramètres en C, comme `EINTR` ou `O_NDELAY` sont automatiquement exporter dans votre espace de noms en tant que fonctions constantes. Les fonctions qui ne sont normalement pas disponibles en Perl (comme `floor`, `ceil`, `strftime`, `uname`, `setsid`, `setlocale` et `sysconf`) sont exportés par défaut. Les fonctions du même nom que les opérateurs internes de Perl, comme `open`, ne sont pas exportées à moins de le demander explicitement, mais la plupart de gens sont susceptibles de préférer des noms de fonctions pleinement qualifiés pour distinguer `POSIX::open` de `CORE::open`.

Quelques fonctions ne sont pas implémentées car elles sont spécifiques au C. Si vous tenter de les appelez, elles affichent un messages vous disant qu'elles ne sont pas implémentées et vous suggèrent d'utiliser un équivalent Perl s'il en existe. Par exemple, une tentative d'accéder à la fonction `setjmp` affiche le message « `setjmp() is C-specific: use eval {} instead` » et `tmpfile` vous dit « `Use method IO::File::new_tmp_file()` ». (Mais depuis la version 5.6.1, vous devriez plutôt utilisez `File::Temp`.)

Le module POSIX vous emmène aussi près du système d'exploitation (ou tout du moins aux parties adressées par le standard POSIX) que ne le ferait n'importe quel programmeur C. Ceci vous donne accès à un pouvoir phénoménale et à des choses très utiles, comme bloquer les signaux et contrôler les options des entrées/sorties du terminal. Toutefois, cela signifie également que votre code finira par ressembler quasiment à du C. Comme démonstration utile de la manière d'éviter les vidages de tampons en entrée, voici un exemple d'un programme complet pour obtenir un seul caractère en entrée sans tampon, sur un système POSIX :

```
#!/usr/bin/perl -w
use strict;
$| = 1;
for (1..4) {
    my $lu;
    print "donnez-moi : ";
    $lu = lire_un();
    print "--> $lu\n";
}
exit;

BEGIN {
    use POSIX qw(:termios_h);
    my ($term, $oterm, $echo, $noecho, $df_entree);
    $df_entree = fileno(STDIN);
    $term      = POSIX::Termios->new();
    $term->getattr($df_entree);
    $oterm     = $term->getlflag();
    $echo      = ECHO | ECHOK | ICANON;
    $noecho    = $oterm & ~$echo;
    sub cbreak {
```

```
        $term->setlflag($noecho);
        $term->setcc(VTIME, 1);
        $term->setattr($df_entree, TCSANOW);
    }
    sub cuisine {
        $term->setflag($oterm);
        $term->setcc(VTIME, 0);
        $term->setattr($df_entree, TCSANOW);
    }
    sub lire_un {
        my $clef = "";
        cbreak();
        sysread(STDIN, $clef, 1);
        cuisine();
        return $clef;
    }
}
END { cuisine() }
```

La page de man du module POSIX donne un inventaire complet des fonctions et constantes qu'il exporte. Il y en a tant que vous finirez souvent par n'en importer qu'un sous-ensemble, comme « :queue », « :sys_stat_h » ou « :termios_h ». Un exemple de blocage de signaux avec le module POSIX est donné au chapitre 16.

Safe

```
use Safe;

$mem_proteg = Safe->new();                # espace mémoire protégé anonyme
$mem_proteg = Safe->new("Nom_Paquetage"); # dans cette table de symboles

# Active ou désactive les codes d'opération par groupe ou par nom
$mem_proteg->permit(qw(:base_core));
$mem_proteg->permit_only(qw(:base_core :base_loop :base_mem));
$mem_proteg->deny("die");

# comme do() mais dans l'espace mémoire protégé
$ok = $mem_proteg->rdo($fichier);

# comme eval() mais dans l'espace mémoire protégé
$ok = $mem_proteg->reval($code);          # sans 'use strict'
$ok = $mem_proteg->reval($code, 1);       # avec 'use strict'
```

Le module Safe tente de fournir un environnement restreint pour protéger le reste du programme des opérations dangereuses. Il utilise deux stratégies différentes pour accomplir ceci. Un peu comme l'utilisation de *chroot*(2) par les démons FTP anonymes altère la vue que l'on a de la racine du système de fichier, la création d'un objet compartiment avec Safe->new("Nom_Paquetage") altère la vue que ce compartiment a de son propre espace de noms. Le compartiment voit maintenant comme sa table de symboles racine (main::), la table de symboles que le reste du programme voit comme

Nom_Paquetage::. Ce qui ressemble à Bidule:: à l'intérieur du compartiment est en réalité Nom_Paquetage::Bidule:: à l'extérieur. Si vous ne donnez pas d'argument au constructeur, un nouveau nom de paquetage aléatoire est sélectionné pour vous. Le second et le plus important service fournit par un compartiment Safe est une manière de limiter le code considéré comme légal à l'intérieur d'un eval. Vous pouvez ajuster l'ensemble des codes d'opérations (*opcodes*) autorisés en utilisant des appels de méthodes sur votre objet Safe. Deux méthodes sont disponibles pour compiler du code dans un compartiment Safe : rdo (« restriction du do ») pour les fichiers et reval (« restriction de l'eval ») pour les chaînes. Ils fonctionnent comme do sur un nom de fichier et comme eval sur une chaîne mais ils s'exécutent dans un espace de noms restreint avec des codes d'opérations limités. Le premier argument est le nom du fichier ou la chaîne à compiler et le second, qui est optionnel, indique si le code doit être compilé sous use strict ou pas.

Il est prévu de réécrire ce module (nous avons l'intention d'isoler l'espace mémoire protégé dans une tâche (*thread*) d'interpréteur différente pour une sécurité supplémentaire), assurez-vous donc de vérifier la page de man de Safe pour les mises à niveau. Voir également le chapitre 23.

Socket

```
use Socket;

$proto = getprotobyname('udp');
socket(SOCK , PF_INET, SOCK_DGRAM, $proto)
    or die "socket : $!";
$iadr  = gethostbyname('hishost.com');
$port  = getservbyname('time', 'udp');
$sin   = sockaddr_in($port, $iadr);
send(SOCK, 0, 0, $sin)
    or die "send : $!";

$proto = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto)
    or die "socket : $!";
$port  = getservbyname('smtp', 'tcp');
$sin   = sockaddr_in($port, inet_aton("127.1"));
$sin   = sockaddr_in(7, inet_aton("localhost"));
$sin   = sockaddr_in(7, INADDR_LOOPBACK);
connect(SOCK, $sin)
    or die "connect : $!";

($port, $iadr) = sockaddr_in(getpeerbyname(SOCK ));
$hote_pair = gethostbyaddr($iadr, AF_INET);
$adr_pair  = inet_ntoa($iadr);

$proto = getprotobyname('tcp');
socket(SOCK, PF_UNIX, SOCK_STREAM, $proto)
    or die "socket : $!";
```

```
unlink('/tmp/usock');    # XXX : l'échec est ignoré intentionnellement
$sun = sockaddr_un('tmp/usock');
connect(SOCK, $sun)
    or die "connect : $!";

use Socket qw(:DEFAULT :crlf);
# Maintenant vous pouvez utiliser CR(), LF() et CRLF() ou
# $CR, $LF et $CRLF pour les fins de lignes.
```

Le module `Socket` donne accès aux constantes du fichier `#include` *sys/socket.h* de la bibliothèque C pour qu'on les utilise dans les fonctions de bas niveau de Perl concernant les sockets. Il fournit également deux fonctions, `inet_aton` et `inet_ntoa`, pour convertir les adresses IP entres leurs représentations ASCII (comme « 127.0.0.1 ») et leur empaquetage réseau, ainsi que deux fonctions spéciales d'empaquetage/dépaquetage, `sockaddr_in` et `sockaddr_un`, qui manipulent les adresses de sockets binaires nécessaires aux appels bas niveau. Le tag d'importation `:crlf` donne des noms symboliques aux conventions diverses de fin de ligne afin que vous n'ayez pas à vous fier aux interprétations natives de `\r` et `\n`, qui peuvent varier. La plupart des protocoles de l'Internet préfèrent CRLF mais tolèrent LF. Le module standard `IO::Socket` fournit une interface de plus haut niveau à TCP. Voir le chapitre 16.

Symbol

```
use Symbol "delete_package";
delete_package("Truc::Bidule");
print "supprimé\n" unless exists $Truc::{"Bidule::"};

use Symbol "gensym";
$sym1 = gensym();                    # Renvoie un nouveau typeglob anonyme.
$sym2 = gensym();                    # Et encore un autre.

package Demo;
use Symbol "qualify";
$sym = qualify("x");                 # "Demo::x"
$sym = qualify("x", "Truc");         # "Truc::x"
$sym = qualify("Bidule::x");         # "Bidule::x"
$sym = qualify("Bidule::x", "Truc"); # "Bidule::x"

use Symbol "qualify_to_ref";
sub pass_handle(*) {
    my $hf = qualify_to_ref(shift, caller);
    ...
}
# Maintenant vous pouvez appeler pass_handle avec HF, "HF", *HF ou \*HF.
```

Le module `Symbol` fournit des fonctions pour faciliter la manipulation des noms globaux : les typeglobs, les noms de formats, les handles de fichiers, les tables de symboles de paquetages et tout autre chose que vous pourriez vouloir nommer *via* une table de symboles. La fonction `delete_package` nettoie complètement l'espace de nom d'un paquetage (en rendant effectivement anonymes toutes références supplémentaires aux ré-

férants de la tables de symboles, y compris les références de code précompilé). La fonction `gensym` renvoie un typeglob anonyme chaque fois qu'elle est appelée. (Cette fonction n'est plus tellement utilisée de nos jours, maintenant que les scalaires indéfinis s'autovivifient avec les handles de fichiers appropriés lorsqu'ils sont utilisés comme arguments de `open`, `pipe`, `socket` et leurs semblables.)

La fonction `qualify` prend un nom qui peut être complètement qualifié avec un paquetage, ou pas, et renvoie un nom qui l'est. Si elle a besoin de préfixer le nom du paquetage, elle utilisera le nom spécifié *via* le second argument (ou s'il est omis, votre nom de paquetage courant). La fonction `qualify_to_ref` fonctionne de la même manière mais produit une référence au typeglob que le symbole représenterait. Ceci est important dans les fonctions qui requièrent qu'ils soient passés en tant que références. En convertissant cet argument avec `qualify_to_ref`, vous pouvez maintenant utiliser le handle fourni même avec `strict refs` activé. Vous pouvez également le consacrer avec `bless` dans une forme objet, puisqu'il s'agit d'une référence valide.

Sys::Hostname

```
use Sys::Hostname;
$hote = hostname();
```

Le module `Sys::Hostname` ne fournit qu'une seule fonction, `hostname`, qui compense ce fait en se remuant le derrière pour essayer de deviner comment votre hôte courant s'appelle. Sur les systèmes qui supportent l'appel système standard *gethostname*(2), celui-ci est utilisé, comme il s'agit de la méthode la plus efficace.[6] Sur les autres systèmes, l'affichage de l'appel standard *hostname*(1) est employé. Sur d'autres encore, on appelle la fonction *uname*(3) de votre bibliothèque C, laquelle est également accessible par la fonction `POSIX::uname` de Perl. Si ces stratégies échouent toutes, d'autres efforts héroïques sont tentés. Quelles que soient les opinions de votre système sur ce qui est sensé ou insensé, Perl essaie de son mieux pour faire avec. Sur certains systèmes, ce nom d'hôte peut ne pas être pleinement qualifié avec le nom de domaine ; voir le module `Net::Domain` de CPAN si vous avez besoin de cela.

Il faut également prendre en considération que `hostname` ne renvoie qu'une seule valeur mais que votre système peut avoir plusieurs interfaces réseaux configurées, vous pouvez donc ne pas vous voir renvoyer le nom associé à l'interface à laquelle vous vous intéressez si vous prévoyez d'utiliser ce module pour certains types de programmation de sockets. Il s'agit des cas où vous devrez probablement glaner dans l'affichage de la commande *ifconfig*(8) ou dans l'équivalent moral sur votre système.

Sys::Syslog

```
use Sys::Syslog;                          # Perd setlogsock.
use Sys::Syslog qw(:DEFAULT setlogsock);  # Obtient également setlogsock.
```

6. Laquelle est accessible directement par la fonction non-exportée `Sys::Hostname::ghname`, mais ne dites à personne qu'on vous l'a dit.

```
openlog($programme, 'cons,pid', 'user');
syslog('info', 'ceci est un autre test');
syslog('mail|warning', 'ceci est un meilleur test : %d', time());
closelog();

syslog('debug', 'ceci est le dernier test');

setlogsock('unix');
openlog("$programme $$", 'ndelay', 'user');
syslog('info', 'le problème était %m'); # %m = $! pour le journal système
syslog('notice', 'prog_truc : ceci est vraiment terminé');

setlogsock('unix');                        # "inet" ou "unix"
openlog("mon_programme", $logopt, $services);
syslog($priorite, $format, @args);
$ancien_masque = setlogmask($priorite_masque);
closelog();
```

Le module `Sys::Syslog` se comporte comme la fonction *syslog*(3) de votre bibliothèque C, en envoyant des messages au démon de votre journal système, *syslogd*(8). Il est particulièrement utile dans les démons ou d'autres programmes qui ne disposent pas de terminal pour recevoir les affichages de diagnostics ou pour des programmes préoccupés par la sécurité qui veulent produire un enregistrement plus durable de leurs actions (ou des actions de quelqu'un d'autre). Les fonctions supportées sont :

`openlog` *IDENT*`,` *LISTE_OPT*`,` *SERVICES*

Établit une connexion avec votre amical démon syslog. *IDENT* est la chaîne sous laquelle enregistrer le message (comme `$0`, le nom de votre programme). *LISTE_OPT* est une chaîne avec des options séparées par des virgules comme « `cons` », « `pid` » et « `ndelay` ». *SERVICES* est quelque chose comme « `auth` », « `daemon` », « `kern` », « `lpr` », « `mail` », « `news` » ou « `user` » pour les programmes système et parmi « `local0` » .. « `local7` » pour les programmes locaux. Davantage de messages sont enregistrés en utilisant un service donnée et la chaîne identificatrice.

`syslog` *PRIORITE*`,` *FORMAT*`,` *ARGS*

Envoie un message au démon en employant la *PRIORITE* donnée. Le *FORMAT* est exactement comme `printf` en comblant les séquences d'échappement des pourcentages avec les *ARGS* qui suivent excepté le fait qu'en respectant les conventions de la fonction *syslog*(3) de la bibliothèque standard, la séquence spéciale « `%m` » est interpolée avec `errno` (la variable `$!` de Perl) à cet endroit.

`setlogsock` *TYPE*

TYPE doit valoir soit « `inet` », soit « `unix` ». Certains démons de systèmes ne font pas attention par défaut aux messages de syslog provenant du domaine Internet, vous pouvez alors lui affecter « `unix` » à la place, puisqu'il ne s'agit pas de la valeur par défaut.

`closelog`

Rompt la connexion avec le démon.

Pour que ce module fonctionne avec les versions de Perl antérieures à la 5.6.0, votre administrateur système doit lancer *h2ph*(1) sur votre fichier d'en-tête *sys/syslog.h* pour créer un fichier de bibliothèque *sys/syslog.ph*. Toutefois, ceci n'était pas accompli par défaut à l'installation de Perl. Les versions ultérieures utilisent maintenant une interface XS, la préparation de *sys/syslog.ph* n'est donc plus nécessaire.

Term::Cap

```
use Term::Cap;

$ospeed = eval {
    require POSIX;
    my $termios = POSIX::Termios->new();
    $termios->getattr;
    $termios->getospeed;
} || 9600;

$terminal = Term::Cap->Tgetent({ TERM => undef, OSPEED => $ospeed });
$terminal->Tputs('cl', 1, STDOUT);             # Efface l'écran.
$terminal->Tgoto('cm', $col, $lig, STDOUT);    # Déplace le curseur.
```

Le module `Term::Cap` donne accès aux routines *termcap*(3) de votre système. Voir la documentation de votre système pour plus de détails. Le systèmes qui ne disposent que de *terminfo*(5) mais pas de *termcap*(5) échoueront. (Beaucoup de système terminfo peuvent émuler termcap.) Toutefois, vous trouverez sur CPAN un module `Term::Info` ainsi que `Term::ReadKey`, `Term::ANSIColor` et divers modules `Curses` pour vous aider avec les entrées caractère par caractère, les affichages en couleur ou la gestions des écrans de terminaux à un niveau plus élevé que `Term::Cap` ou `Term::Info`.

Text::Wrap

```
use Text::Wrap;      # Importe wrap().

@lignes = (< <"FIN_G&S" =~ /\\S.*\S/g);

    Ceci est particulièrement rapide,
    un boniment inintelligible
    n'est généralement
    pas entendu,
    et s'il
    l'est, on
    s'en fiche.

FIN_G&S

$Text::Wrap::columns = 50;
print wrap(" " x 8, " " x 3, @lignes), "\n";
```

Ceci affiche :

```
    Ceci est particulièrement rapide, un boniment inintelligible n'est
généralement pas entendu, et s'il l'est, on s'en fiche.
```

Le module `Text::Wrap` implémente un formateur de paragraphes simples. Sa fonction `wrap` formate un simple paragraphe à la fois en coupant les lignes sans le faire au milieu d'un mot. Le premier argument est le préfixe ajouté au début de la première ligne renvoyée. Le second est la chaîne utilisée comme préfixe pour toutes les lignes sauf la première. Tous les arguments suivants sont joints en utilisant un saut de ligne comme séparateur et ils sont renvoyés comme une seule chaîne de paragraphe reformatée. Vous devez apprécier vous-même la largeur de votre terminal ou au moins spécifier ce que vous désirez dans `$Text::Wrap::columns`. Bien que l'on puisse utiliser l'appel `ioctl TIOCWINSZ` pour apprécier le nombre de colonnes, il serait plus facile pour ceux qui ne sont pas habitués à la programmation en C, d'installer le module `Term::Readkey` de CPAN et d'utiliser la routine `GetTerminalSize` de ce module.

Time::Local

```
use Time::Local;
$date = timelocal($sec,$min,$heures,$jour_m,$mois,$annee);
$date = timegm($sec,$min,$heures,$jour_m,$mois,$annee);

$date = timelocal(50, 45, 3, 18, 0, 73);
print "scalar localtime donne : ", scalar(localtime($date)), "\n";
$date += 28 * 365.2425 * 24 * 60 * 60;
print "Vingt-huit années plus tard, il est maintenant\n\t",
    scalar(localtime($date)), "\n";
```

Ceci affiche :

```
scalar localtime donne : Thu Jan 18 03:45:50 1973
Vingt-huit années plus tard, il est maintenant
        Wed Jan 17 22:43:26 2001
```

Le module `Time::Local` fournit deux fonctions, `timelocal` et `timegm`, qui fonctionne exactement à l'inverse des fonctions `localtime` et `gmtime`, respectivement. C'est-à-dire qu'elles prennent une liste de valeurs numériques pour les divers composants de ce que `localtime` renvoie dans un contexte de liste et déterminent quel paramètre passé en entrée à `localtime` aurait produit ces valeurs. Vous pouvez utiliser ceci si vous voulez comparer ou lancer des calculs sur deux dates différentes. Bien que ces fonctions ne soient pas génériques pour analyser les dates et les heures, si vous pouvez vous arranger pour avoir vos paramètres dans le bon format, elles sont souvent suffisantes. Comme vous pouvez le voir dans l'exemple ci-dessus, la date et l'heure possèdent tout de même quelque excentricités et même de simples calculs échouent souvent pour faire ce qu'on leur demande à cause des années bissextiles, des secondes intercallaires et de la phase de la lune. Deux modules de CPAN, énormes mais complets, s'occupent de ces problèmes et bien plus encore : `Date::Calc` et `Date::Manip`.

Time::localtime

```
use Time::localtime;
print "L'année est %d\n", localtime->year() +1900;

$maintenant = ctime();

use Time::localtime;
use File::stat;
$date_chaine = ctime(stat($fichier)->mtime);
```

Ce module supplante la fonction intrinsèque `localtime` en la remplaçant avec une version qui renvoie un objet `Time::tm` (ou `undef` en cas d'échec). Le module `Time::gmtime` fait de même, mais il remplace la fonction intrinsèque `gmtime` à la place. L'objet renvoyé possède des méthodes qui accèdent aux champs nommés comme ceux de la structure `struct tm` dans *time.h* de la bibliothèque C : `sec`, `min`, `hour`, `mday`, `mon`, `year`, `wday`, `yday` et `isdst`. La fonction `ctime` donne un moyen d'obtenir la fonction originale (dans son sens scalaire) `CORE::localtime`. Remarquez que les valeurs renvoyées proviennent directement d'une structure `struct tm`, elles se situent donc dans les mêmes intervalles, voir l'exemple ci-dessus pour la manière correcte de produire une année sur quatre chiffres. La fonction `POSIX::strftime` est encore plus utile pour formater les dates et les heures avec une grande variété de styles plus alléchants les uns que les autres.

User::grent

```
use User::grent;
$gr = getgrgid(0)         or die "Pas de groupe zéro";
if ($gr->name eq "audio" && @{$gr->members} > 1) {
    print "le gid zéro s'appelle audio et compte d'autres membres";
}

$gr = getgr($n_importe_qui); # Accepte à la fois une chaîne ou un nombre.

use User::grent ':FIELDS';
$gr = getgrgid(0)         or die "Pas de groupe zéro";
if ($gr_name eq "audio" && @gr_members > 1) {
    print "le gid zéro s'appelle audio et compte d'autres membres";
}
```

Les exportations par défaut de ce module supplantent les fonctions intrinsèques `getgrent`, `getgruid` et `getgrnam`, en les remplaçant par des versions qui renvoient un objet `User::grent` (ou `undef` en cas d'échec). Cet objet possède des méthodes qui accèdent aux champs nommés comme ceux de la structure `struct group` dans *grp.h* de la bibliothèque C : `name`, `passwd`, `gid` et `members` (et pas `mem` comme en C !). Les trois premières renvoient des scalaires et la dernière une référence de tableau. Vous pouvez également importer les champs de la structure dans votre propre espace de noms en tant que variables ordinaires en utilisant le tag d'importation « `:FIELDS` », bien que ceci supplante toujours les fonctions intrinsèques. Les variables (les trois scalaires et le tableau) sont préfixées par « `gr_` ». La fonction `getgr` est une simple bascule frontale qui fait suivre tout argument numérique à `getgrid` et tout argument chaîne à `getgrnam`.

User::pwent

```
# Par défaut, supplante seulement les fonctions intrinsèques.
use User::pwent;
$pw = getpwnam("daemon")  or die "Pas d'utilisateur daemon";
if ($pw->uid == 1 && $pw->dir =~ m#^/(bin|tmp)?$# ) {
    print "uid 1 sur le répertoire racine";
}

$pw = getpw($n_importe_qui);   # Accepte à la fois une chaîne ou un nombre.
$shell_reel = $pw->shell || '/bin/sh';
for (($nom_complet, $bureau, $tel_bur, $tel_maison) =
      split /\s*,\s*/, $pw->gecos)
{
    s/&/ucfirst(lc($pw->name))/ge;
}

# positionne des variables globales dans le paquetage courant.
use User::pwent qw(:FIELDS);
$pw = getpwnam("daemon")  or die "Pas d'utilisateur daemon";
if ($pw_uid == 1 && $pw_dir =~ m#^/(bin|tmp)?$# ) {
    print "uid 1 sur le répertoire racine";
}

use User::pwent qw/pw_has/;
if (pw_has(qw[gecos expire quota])) { ... }
if (pw_has("name uid gid passwd")) { ... }
printf "Votre struct pwd supporte [%s]\n", scalar pw_has();
```

Les exportations par défaut de ce module supplantent les fonctions intrinsèques `getpwent`, `getpwuid` et `getpwnam`, en les remplaçant par des versions qui renvoient un objet `User::pwent` (ou `undef` en cas d'échec). Il est souvent préférable d'utiliser ce module plutôt que les fonctions intrinsèques qu'il remplace, car les fonctions intrinsèques surchargent, voire omettent divers éléments dans la liste renvoyer au nom de la compatibilité ascendante.

L'objet renvoyé possède des méthodes qui accèdent aux champs nommés comme ceux de la structure `passwd` dans *pwd.h* de la bibliothèque C, avec le préfixe « `pw_` » en moins : `name`, `passwd`, `uid`, `gid`, `change`, `age`, `quota`, `comment`, `class`, `gecos`, `dir`, `shell` et `expire`. Les champs `passwd`, `gecos` et `shell` sont marqués. Vous pouvez également importer les champs de la structure dans votre propre espace de noms en tant que variables ordinaires en utilisant le tag d'importation « `:FIELDS` », bien que ceci supplante toujours les fonctions intrinsèques. Vous accédez à ces champs en tant que variables scalaires nommé avec un préfixe « `pw_` » devant le nom de la méthode. La fonction `getpw` est une simple bascule frontale qui fait suivre tout argument numérique à `getpwuid` et tout argument chaîne à `getpwnam`.

Perl croit qu'aucune machine n'a plus d'un champ parmi `change`, `age` ou `quota` d'implémenté, ni plus d'un parmi `comment` ou `class`. Certaines machines ne supportent pas `expire`, `gecos`, on dit que certaines machines ne supportent même pas `passwd`. Vous pouvez appeler ces méthodes quelle que soit la machine sur laquelle vous êtes mais elles

retourneront `undef` si elles ne sont pas implémentées. Voir *passwd*(5) et *getpwent*(3) pour plus de détails.

Vous pouvez déterminer si ces champs sont implémentés en le demandant à la fonction non-portable `pw_has`. Elle renvoie vrai si tous les paramètres sont des champs supportés sur la plate-forme ou faux si au moins un ne l'est pas. Elle lève une exception si vous demandez un champ dont le nom n'est pas reconnu. Si vous ne passez aucun argument, elle renvoie la liste des champs que votre bibliothèque C pense qu'ils sont supportés.

L'interprétation du champ `gecos` varie entre les systèmes mais il comprend souvent quatre champs séparés par des virgules contenant le nom complet de l'utilisateur, l'emplacement de son bureau, son téléphone professionnel et son téléphone personnel. Un `&` dans le champ `gecos` doit être remplacé par le nom de login, `name`, de l'utilisateur avec les majuscules et minuscules correctes. Si le champ `shell` est vide, on suppose qu'il vaut */bin/sh*, bien que Perl ne le fasse pas pour vous. Le `passwd` est donnée par une fonction de hachage à sens unique produisant du charabia au lieu d'un texte lisible, et on ne peut inverser la fonction de hachage en essayant brutalement toutes les combinaisons possibles. Les systèmes sécurisés utilisent souvent une fonction de hachage plus sûre que DES. Sur les systèmes supportant les mots de passes cachés, Perl renvoie automatiquement l'entrée du mot de passe sous sa forme cachée lorsqu'un utilisateur suffisamment puissant l'appelle, même si la bibliothèque C sous-jacente fournie par votre constructeur avait une vision trop courte pour réaliser qu'il aurait dû faire cela.

33

Messages de diagnostic

Ces messages sont classés dans un ordre croissant de désespoir :

Classe	Signification
(W)	Un avertissement (optionnel).
(D)	Une obsolescence (optionnelle).
(S)	Un avertissement sérieux (obligatoire).
(F)	Une erreur fatale (interceptable).
(P)	Une erreur interne (panique) que vous ne devriez jamais rencontrer (interceptable).
(X)	Une erreur extrêmement fatale (non-interceptable).
(A)	Une erreur venue de l'au-delà (non générée par Perl).

La plupart des messages des trois premières classifications ci-dessus (W, D et S) peuvent être contrôlés en utilisant le pragma `warnings` ou les options **`-w`** et **`-W`**. Si un message peut être contrôlé avec le pragma `warnings`, la catégorie correspondante de cet avertissement est donnée après la lettre de classification ; par exemple, (W misc) indique un avertissement de type divers. Le pragma `warnings` est décrit au chapitre 31, *Modules de pragmas*.

Les avertissements peuvent être capturés plutôt qu'affichés en faisant pointer `$SIG{__WARN__}` sur une référence de routine qui sera appelée pour chaque message. Vous pouvez également en prendre le contrôle avant qu'une erreur interceptable « meure » en assignant une référence de sous-programme à `$SIG{__DIE__}`, mais si vous n'appelez pas `die` à l'intérieur de ce gestionnaire, l'exception sera toujours présente en revenant. En d'autres termes, il n'est pas possible de la « défataliser » par ce biais. Vous devez utiliser `eval` à cet effet.

Les avertissements par défaut sont toujours activés sauf s'ils sont explicitement désactivés avec le pragma `warnings` ou l'option **`-X`**.

Dans les messages suivants, *%s*représente une chaîne interpolée qui n'est déterminée qu'au moment de la génération du message. (De même, *%d* représente un nombre interpolé — pensez aux formats de `printf`, mais *%d* signifie ici un nombre dans n'importe quelle base.) Remarquez que certains messages *commencent* par *%s* — d'où une certaine difficulté à les classer dans l'ordre alphabétique. Vous pouvez chercher dans ces messages si celui que vous recherchez n'apparaît pas à l'endroit attendu. Les symboles `"%-?@` sont triés avant les caractères alphabétiques, alors que `[` et `\` le sont après.

Si vous êtes persuadé qu'un bogue provient de Perl et non de vous, essayez de le réduire à un cas de test minimal et faites un rapport avec le programme *perlbug* qui est fourni avec Perl.

N.d.T. Les messages sont indiqués ici en anglais tels qu'ils apparaissent à l'exécution de vos scripts Perl. Toutefois, nous en donnons ici une traduction pour en faciliter la compréhension.

`"%s" variable %s masks earlier declaration in same %s`

(La variable *"%s"* masque une déclaration précédente dans la même *%s*)

(W misc) Une variable `my` ou `our` a été redéclarée dans la portée ou dans l'instruction courante, éliminant effectivement tout accès à l'instance précédente. Il s'agit presque toujours d'une erreur de typographie. Remarquez que la variable déclarée plus tôt existera toujours jusqu'à la fin de la portée ou jusqu'à ce que toutes les fermetures qui s'y réfèrent soient détruites.

`"my sub" not yet implemented`

("`my sub`" n'est pas encore implémenté)

(F) Les sous-programmes de portée lexicale ne sont pas encore implémentés. N'essayez pas encore cela.

`"my" variable %s can't be in a package`

(La variable "`my`" *%s* ne peut pas se trouver dans un paquetage)

(F) Les variables de portée lexicale ne se trouvent pas dans un paquetage, et il ne sert donc à rien d'essayer d'en déclarer une, préfixée d'un qualificateur de paquetage. Utilisez `local` si vous désirez donner une valeur locale à une variable de paquetage.

`"no" not allowed in expression`

("`no`" n'est pas autorisé dans une expression)

(F) Le mot-clef `no` est reconnu et exécuté à la compilation et il ne renvoie aucune valeur utile.

`"our" variable %s redeclared`

(La variable "`our`" *%s* est redéclarée)

(W misc) Il semble que vous avez déjà déclaré la même variable globale auparavant dans la même portée lexicale.

`"use" not allowed in expression`

("`use`" n'est pas autorisé dans une expression)

(F) Le mot-clef `use` est reconnu et exécuté à la compilation et il ne renvoie aucune valeur utile.

`'!' allowed_only_after types %s`

('!' n'est autorisé qu'après les types *%s*)

(F) Le '!' n'est autorisé dans pack et unpack qu'après certains types.

`'|' and '<' may not both be specified on command line`

('|' et '<' ne peuvent pas être spécifiés en même temps sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirections dans la ligne de commande. Il se trouve que STDIN était un pipe et que vous avez essayé de rediriger STDIN en utilisant <. Un seul flux STDIN par personne, s'il vous plaît.

`'|' and '>' may not both be specified on command line`

('|' et '>' ne peuvent pas être spécifiés en même temps sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirections dans la ligne de commande et pense que vous avez essayé de rediriger STDOUT à la fois dans un fichier et dans un pipe vers une autre commande. Vous devez choisir l'un ou l'autre, bien que rien ne vous empêche de rediriger dans un pipe vers un programme ou un script Perl qui sépare la sortie standard en deux flux, comme dans :

```
open(SORTIE,">$ARGV[0]") or die "Impossible d'écrire dans $ARGV[0] : $!";
while (<STDIN>) {
    print;
    print SORTIE;
}
close SORTIE;
```

`/ cannot take a count`

(/ ne peut pas être lié à un compteur)

(F) Vous aviez un canevas pour unpack indiquant une chaîne d'une longueur sujette à un compteur mais vous avez également spécifié une taille explicite pour cette chaîne.

`/ must be followed by a, A ou Z`

(/ doit être suivi par A, a ou Z)

(F) Vous aviez un canevas pour unpack indiquant une chaîne d'une longueur sujette à un compteur ; cette chaîne doit être suivie d'une des lettres a, A ou Z pour indiquer quelle sorte de chaîne doit être dépaquetée.

`/ must be followed by a*, A* ou Z*`

(/ doit être suivi par A*, a* ou Z*)

(F) Vous aviez un canevas pour unpack indiquant une chaîne d'une longueur sujette à un compteur. Actuellement, les seules choses dont on peut compter la longueur sont a*, A* ou Z*.

`/ must follow a numeric type`

(/ doit suivre un type numérique)

(F) Vous aviez un canevas pour unpack contenant un #, mais celui-ci ne suivait pas l'une des spécifications numériques d'unpack.

`% may only be used in unpack`

(% ne peut être utilisé que dans `unpack`)

(F) Vous ne pouvez pas empaqueter une chaîne avec `pack` en fournissant une somme de contrôle car le traitement de cette dernière fait perdre de l'information et vous ne pouvez pas revenir en arrière.

`Repeat count in pack overflows`

(Dépassement du compteur de répétitions dans `pack`)

(F) Vous ne pouvez pas spécifier un compteur de répétitions tellement grand qu'il dépasse la limite pour vos entiers signés.

`Repeat count in unpack overflows`

(Dépassement du compteur de répétitions dans `unpack`)

(F) Vous ne pouvez pas spécifier un compteur de répétitions tellement grand qu'il dépasse la limite pour vos entiers signés.

`/%s/ Unrecognized escape \\%c passed through`

(`/%s/` : Rencontre d'une séquence d'échappement `\\%c` inconnue)

(W regexp) Vous avez employé une combinaison avec un caractère antislash qui n'est pas reconnue par Perl. Cette combinaison apparaît dans une variable interpolée ou une expression régulière délimitée par `'`. Le caractère a été interprété littéralement.

`/%s/ Unrecognized escape \\%c in character class passed through`

(`/%s/` : Rencontre d'une séquence d'échappement `\\%c` inconnue dans une classe de caractères)

(W regexp) Vous avez employé une combinaison avec un caractère antislash qui n'est pas reconnue par Perl à l'intérieur d'une classe de caractères. Le caractère a été interprété littéralement.

`/%s/ should probably be written as "%s"`

(`/%s/` devrait probablement être écrit `"%s"`)

(W syntax) Vous avez employé un motif là où Perl s'attendait à trouver une chaîne, comme dans le premier argument de `join`. Perl utilisera comme chaîne le résultat vrai ou faux de la correspondance entre le motif et `$_`, ce qui n'est probablement pas ce que vous aviez en tête.

`%s(...) interpreted as function`

(`%s(...)` interprété comme une fonction)

(W syntax) Vous avez dérogé à la règle qui dit que tout opérateur de liste suivi par des parenthèses devient une fonction prenant comme arguments tous les opérateurs de listes trouvé entre les parenthèses.

`%s() called too early to check protoype`

(`%s()` appelée trop tôt pour vérifier le prototype)

(W prototype) Vous avez appelé une fonction possédant un prototype avant que l'analyseur syntaxique ne voie sa déclaration ou sa définition et Perl ne peut pas vérifier si l'appel se conforme au prototype. Pour obtenir une vérification correcte

du prototype, vous devez soit ajouter une déclaration du prototype pour le sous-programme en question plus haut dans le code, soit déplacer la définition du sous-programme avant l'appel. Ou alors, si vous êtes certains d'appeler la fonction correctement, vous pouvez mettre une esperluette devant le nom pour éviter l'avertissement.

`%s argument is not a HASH or ARRAY element`

(L'argument `%s` n'est un élément ni de HACHAGE, ni de TABLEAU)

(F) L'argument de `exists` doit être un élément de hachage ou de tableau, comme :

```
$truc{$bidule}
$ref->{"susie"}[12]
```

`%s argument is not a HASH or ARRAY element or slice`

(L'argument `%s` n'est un élément ni de HACHAGE, ni de TABLEAU, ni une tranche)

(F) L'argument de `delete` doit être soit un élément de hachage ou de tableau, comme :

```
$truc{$bidule} $ref->{"susie"}[12]
```

soit une tranche de hachage ou de tableau, comme :

```
@truc[$bidule, $machin, $chouette]
@{$ref->[12]}{"susie", "queue"}
```

`%s argument is not a subroutine name`

(L'argument `%s` n'est pas un nom de sous-programme)

(F) L'argument de `exists` pour `exists &sub` doit être un nom de sous-programme et non un appel de sous-programme. `exists &sub()` générera cette erreur.

`%s did not return a true value`

(`%s` n'a pas renvoyé une valeur vraie)

(F) Un fichier requis par `require` (ou utilisé par `use`) doit retourner une valeur vraie pour indiquer qu'il s'est compilé correctement et que l'initialisation de son code s'est bien passée. Il est habituel de placer un `1;` à la fin d'un tel fichier, bien qu'une quelconque valeur fasse l'affaire.

`%s failed--call queue aborted`

(`%s` a échoué — file d'appels abandonnée)

(F) Une exception non capturée a été levée pendant l'exécution d'un sous-programme `CHECK`, `INIT` ou `END`. Le traitement du reste de la file contenant de telles routines s'est terminé prématurément.

`%s found where operator expected`

(`%s` trouvé là où l'on attendait un opérateur)

(S) L'analyseur lexicographique de Perl sait s'il attend un terme ou un opérateur. S'il voit ce qu'il sait être un terme alors qu'il s'attend à un opérateur, il vous donne ce message d'alerte. Habituellement, cela indique qu'un opérateur ou un séparateur a été omis, comme un point virgule.

`%s had compilation errors`

(*%s* a engendré des erreurs de compilation)

(F) C'est le message final lorsqu'un *perl -c* échoue.

`%s has too many errors`

(*%s* a engendré trop d'erreurs)

(F) L'analyseur abandonne son analyse du programme après 10 erreurs. Les messages suivant ne seraient pas sensés.

`%s matches null string many times`

(*%s* correspond trop de fois à la chaîne vide)

(W regexp) Le motif que vous avez spécifié rentrerait dans une boucle infinie si le moteur d'expressions régulières ne contrôlait pas cela.

`%s never introduced`

(*%s* n'a jamais été introduit)

(S internal) Le symbole en question a été déclaré mais sa portée s'est terminée avant qu'il soit possible de s'en servir.

`%s package attribute may clash with future reserved word: %s`

(L'attribut de paquetage *%s* risque de rentrer en conflit avec le mot *%s* réservé pour une utilisation ultérieure)

(W reserved) Un nom d'attribut en minuscules géré par un paquetage spécifique a été utilisé. Ce mot pourrait avoir un sens pour Perl un jour, même si ce n'est pas encore le cas. Peut-être devriez-vous employer à la place un nom avec des majuscules et des minuscules mélangées.

`%s syntax OK`

(*%s* syntaxe OK)

(F) Message final lorsqu'un *perl -c* a réussi.

`%s: Command not found`

(*%s*: Commande introuvable)

(A) Vous avez lancé accidentellement votre script avec *csh* au lieu de Perl. Vérifiez la ligne avec `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`%s: Expression syntax`

(*%s*: Syntaxe d'expression)

(A) Vous avez lancé accidentellement votre script avec *csh* au lieu de Perl. Vérifiez la ligne avec `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`%s: Undefined variable`

(*%s*: Variable indéfinie)

(A) Vous avez lancé accidentellement votre script avec *csh* au lieu de Perl. Vérifiez la ligne avec `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`%s: not found`

(*%s*: introuvable)

(A) Vous avez lancé accidentellement votre script avec le Bourne Shell au lieu de Perl. Vérifiez la ligne avec `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`(in cleanup) %s`

((au nettoyage) *%s*)

(W misc) Ce préfixe indique habituellement qu'une méthode `DESTROY` a levé l'exception indiquée. Puisque les destructeurs sont appelés habituellement par le système à un moment arbitraire durant l'exécution, et bien souvent un grand nombre de fois, cet avertissement n'est affiché qu'une seule fois pour tous les échecs qui sinon auraient causé une répétition du même message.

Les échecs des fonctions de rappel (*callbacks*) définies par l'utilisateur et propagées par l'emploi du flag `G_KEEPERR` peuvent également entraîner cet avertissement. Voir *perlcall*(1).

`(Missing semicolon on previous line?)`

(Point-virgule manquant sur la ligne précédente ?)

(S) Il s'agit d'une supposition donnée en complément du message « *%s* `found where operator expected` ». Pour autant, n'ajoutez pas systématiquement un point-virgule simplement parce que vous avez lu ce message

`-P not allowed for setuid/setgid script`

(`-P` n'est pas autorisé pour un script setuid/setgid)

(F) Le préprocesseur C a voulu ouvrir le script en l'appelant par son nom, ce qui entraîne une situation de concurrence (*N.d.T.* : *race condition*) qui casse la sécurité.

`-T and -B not implemented on filehandles`

(`-T` et `-B` ne sont pas implémentés sur les handles de fichiers)

(F) Perl ne peut utiliser le tampon d'entrée/sortie du handle de fichier quand il ne connaît pas votre type d'entrées/sorties standards. Vous devez utiliser un nom de fichier à la place.

`-p destination: %s`

(Destination de `-p` : *%s*)

(F) Une erreur est survenue durant l'affichage sur la sortie standard implicite invoquée par l'option de la ligne de commande `-p`. (Cette sortie est redirigée vers `STDOUT` à moins que vous ne l'ayez redirigée avec `select`.)

`500 Server error`

(Erreur de serveur 500)

Voir Server error (Erreur de serveur).

`?+* follows nothing in regexp`

(`?+*` ne suivent rien dans l'expression régulière)

(F) Vous avez commencé une expression rationnelle avec un quantifiant. Mettez un antislash devant si vous vouliez l'employer au sens littéral.

`@ outside of string`

(@ à l'extérieur de la chaîne)

(F) Vous aviez un canevas empaqueté qui spécifiait une position absolue en dehors de la chaîne à dépaqueter.

`<> should be quotes`

(<> devraient être des apostrophes)

(F) Vous avez écrit `require <fichier>` alors que vous auriez dû écrire `require 'fichier'`.

`\1 better written as $1`

(Il est préférable d'écrire `$1` plutôt que `\1`)

(W syntax) À l'extérieur des motifs, les références arrières (*backreferences*) existent en tant que variables. L'utilisation d'antislashs du côté droit d'une substitution était autorisée pour nos grands-pères, mais stylistiquement, il mieux vaut adopter la forme des variables attendue par les autres programmeurs Perl. De plus, cela fonctionne mieux s'il y a plus de neuf références arrières.

`accept() on closed socket %s`

(`accept()` sur une socket fermée *%s*)

(W closed) Vous avez tenté de faire un `accept` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel de `socket` ?

`Allocation too large: %lx`

(Allocation trop grande : `%lx`)

(X) Vous ne pouvez allouer plus de 64K sur une machine MS-DOS.

`Applying %s to %s will act on scalar(%s)`

(Appliquer *%s* à *%s* fonctionne avec un scalaire (*%s*))

(W misc) Les opérateurs de recherche de correspondance (`//`), de substitution (`s///`) et de translation (`tr///`) fonctionnent avec des valeurs scalaires. Si vous appliquez l'un d'eux à un tableau ou à un hachage, il convertit ce dernier en une valeur scalaire — la longueur du tableau ou les informations sur le peuplement du hachage — puis travaille sur la valeur scalaire. Ce n'est probablement pas ce que vous vouliez faire.

`Arg too short for msgsnd`

(Argument trop court pour `msgsnd`)

(F) `msgsnd` nécessite une chaîne au moins aussi longue que `sizeof(long)`.

`Ambiguous use of %s resolved as %s`

(Utilisation ambiguë de *%s* résolue avec *%s*)

(W ambiguous | S) Vous avez exprimé quelque chose qui n'est pas interprété comme vous le pensiez. Normalement il est assez facile de clarifier la situation en rajoutant une apostrophe manquante, un opérateur, une paire de parenthèses ou une déclaration.

`Ambiguous call resolved as CORE::%s(), qualify as such or use &`

(Appel ambigu résolu avec `CORE::%s`, qualifiez le ainsi ou utiliser `&`)

(W ambiguous) Un sous-programme que vous avez déclaré a le même nom qu'un mot clef de Perl et vous avez utilisé ce nom sans le qualifier pour que l'on puisse distinguer l'un de l'autre. Perl décide d'appeler la fonction interne car votre sous-programme n'est pas importé.

Pour forcer l'interprétation sur l'appel de votre sous-programme, mettez une esperluette avant le nom ou qualifiez votre sous-programme avec son nom de paquetage. Ou alors, vous pouvez importer le sous-programme (ou prétendre le faire avec le pragma `use subs`).

Pour l'interpréter en silence comme l'opérateur Perl, utilisez le préfixe `CORE::` sur cet opérateur (ex. `CORE::log($x)`) ou déclarez le sous-programme en tant que méthode objet.

`Args must match #! line`

(Les arguments doivent correspondre à la ligne `#!`)

(F) L'émulateur setuid nécessite que les arguments invoqués par Perl correspondent avec ceux utilisé sur la ligne `#!`. Comme certains systèmes imposent un unique argument sur la ligne `#!`, essayez de combiner les options ; par exemple, changez **`-w -U`** en **`-wU`**.

`Argument "%s" isn't numeric`

(L'argument "*%s*" n'est pas numérique)

(W numeric) La chaîne indiquée est utilisée comme un argument avec un opérateur qui s'attend plutôt à une valeur numérique. Si vous êtes chanceux, le message indique quel opérateur a été si malchanceux.

`Array @%s missing the @ in argument %d of %s()`

(D deprecated) Les versions vraiment antérieures de Perl permettaient d'omettre le `@` des tableaux à certains endroits. Ceci est maintenant largement obsolète.

`assertion botched: %s`

(assertion sabotée : *%s*)

(P) Le paquetage `malloc` fourni avec Perl a subi une erreur interne.

`Assertion failed: file "%s"`

(L'assertion a échoué : fichier "*%s*")

(P) Une assertion générale a échoué. Le fichier en question doit être examiné.

`Assignment to both a list and a scalar`

(Assignement simultané à une liste et à un scalaire)

(F) Si vous affectez à un opérateur conditionnel, les deuxième et troisième arguments doivent soit être tous les deux des scalaires, soit tous les deux des listes. Autrement Perl ne connaît pas le contexte à fournir au côté droit.

`Attempt to free non-arena SV: 0x%lx`

(Tentative de libérer un SV hors d'une zone : `0x%lx`)

(P internal) Tous les objets SV sont supposés être alloués dans une zone qui sera nettoyée à la sortie du script. Un SV a été découvert en dehors d'une de ces zones.

`Attempt to free nonexistent shared string`

(Tentative de libérer une chaîne partagée inexistante)

(P internal) Perl maintient une table interne qui compte les références de chaînes pour optimiser le stockage et l'accès au clefs de hachages et aux autres chaînes. Ce message indique que quelqu'un a essayé de décrémenter le compteur de références d'une chaîne que l'on ne peut plus trouver dans la table.

`Attempt to free temp prematurely`

(Tentative prématurée de libérer une valeur temporaire)

(W debugging) Les valeurs à l'agonie (*mortalized*) sont supposées être libérées par la routine interne `free_tmps`. Ce message indique que quelque chose d'autre est en train de libérer le SV avant que la routine `free_tmps` ait une chance de le faire, ce qui signifie que la routine `free_tmps` libérera un scalaire non référencé au moment où elle tentera de le faire.

`Attempt to free unreferenced glob pointers`

(Tentative de libérer des pointeurs de glob non référencés)

(P internal) Le compteur de références a été endommagé en ce qui concerne les alias de symboles.

`Attempt to free unreferenced scalar`

(Tentative de libérer un scalaire non référencé)

(W internal) Perl allait décrémenter un compteur de références d'un scalaire pour voir s'il arrivait à 0 et a découvert qu'il était déjà arrivé à 0 auparavant et qu'il se pourrait qu'il ait été libéré et, en fait, qu'il a probablement été libéré. Cela peut indiquer que `SvREFCNT_dec` a été appelé trop de fois ou que `SvREFCNT_inc` n'a pas été appelé assez de fois ou encore que le SV a été achevé (*mortalized*) alors qu'il n'aurait pas dû l'être ou enfin que la mémoire a été corrompue.

`Attempt to join self`

(Tentative de se rejoindre soi-même)

(F) Vous avez essayer de rejoindre avec `join` une tâche (*thread*) à partir d'elle-même, ce qui est une opération impossible. Vous pourriez rejoindre la mauvaise tâche ou avoir besoin de déplacer le `join` vers une autre tâche.

`Attempt to pack pointer to temporary value`

(Tentative d'empaqueter un pointeur vers une valeur temporaire)

(W pack) Vous avez essayé de passer une valeur temporaire (comme le résultat d'une fonction ou une expression calculée) au canevas `p` de `pack`. Cela signifie que le résultat contient un pointeur vers un endroit qui pourrait devenir invalide à tout moment, même avant la fin de l'instruction en cours. Utilisez des valeurs littérales ou globales comme arguments du canevas `p` de `pack` pour éviter cet avertissement.

`Attempt to use reference as lvalue in substr`

(Tentative d'utiliser une référence en tant que lvalue dans `substr`)

(W substr) Vous avez soumis une référence utilisée en tant que lvalue comme premier argument de `substr`, ce qui est vraiment étrange. Peut-être avez-vous oublié de la déréférencer en premier lieu.

`Bad arg length for %s, is %d, should be %d`

(Mauvaise longueur d'argument pour *%s*, vaut *%d*, devrait valoir *%d*)

(F) Vous avez passé un tampon de taille incorrecte à `msgctl`, `semctl` ou `shmctl`. En C, les tailles correctes sont respectivement `sizeof(struct msqid_ds *)`, `sizeof(struct semid_ds *)` et `sizeof(struct shmid_ds *)`.

`Bad evalled substitution pattern`

(Mauvaise évaluation du motif de substitution)

(F) Vous avez utilisé l'option `/e` pour évaluer l'expression de remplacement pour une substitution mais Perl a trouvé une erreur de syntaxe dans le code à évaluer, très certainement une parenthèse fermante inattendue.

`Bad filehandle: %s`

(Mauvais handle de fichier : *%s*)

(F) Un symbole a été passé à quelque chose qui s'attendait à un handle de fichier mais le symbole n'a aucun handle de fichier qui lui soit associé. Peut-être avez-vous oublié de faire un `open` ou dans vous l'avez fait dans un autre paquetage.

`Bad free() ignored`

(Mauvais `free()` ignoré)

(S malloc) Une routine interne a appelé `free` sur quelque chose qui n'avait jamais été alloué avec `malloc` dans un premier temps. Ce message est obligatoire mais peut être désactivé en positionnant la variable d'environnement `PERL_BADFREE` à 1.

Ce message peut être assez fréquent avec `DB_file` sur les systèmes avec une édition de liens dynamiques en « dur », comme AIX et OS/2. C'est un bug de Berkeley DB.

`Bad hash`

(Mauvais hachage)

(P) Une des routines internes concernant un hachage a passé un pointeur HV nul.

`Bad index while coercing array into hash`

(Mauvais indice lors de la transformation d'un tableau en hachage)

(F) La recherche dans le hachage, 0ème élément d'un pseudo-hachage, a découvert un indice illégal. La valeurs des indices doit être supérieure ou égale à 1.

`Bad name after %s::`

(Mauvais nom après *%s*`::`)

(F) Vous avez commencé un nom de symbole en utilisant un préfixe de paquetage puis vous n'avez pas fini le symbole. En particulier, vous ne pouvez pas faire d'interpolation en dehors des guillemets, ainsi :

```
$var = 'ma_var';
$sym = mon_paquetage::$var;
```

n'est pas identique à :

```
$var = 'ma_var';
$sym = "mon_paquetage::$var";
```

`Bad realloc() ignored`

(Mauvais `realloc()` ignoré)

(S malloc) Une routine interne a appelé `realloc` sur quelque chose qui n'avait jamais été alloué avec `malloc` dans un premier temps. Ce message est obligatoire mais peut être désactivé en positionnant la variable d'environnement `PERL_BADFREE` à 1.

`Bad symbol for array`

(Mauvais symbole pour un tableau)

(P) Une requête interne a demandé à ajouter une entrée de tableau à quelque chose qui n'est pas une entrée dans la table des symboles.

`Bad symbol for filehandle`

(Mauvais symbole pour un handle de fichier)

(P) Une requête interne a demandé à ajouter une entrée de handle de fichier à quelque chose qui n'est pas une entrée dans la table des symboles.

`Bad symbol for hash`

(Mauvais symbole pour un hachage)

(P) Une requête interne a demandé à ajouter une entrée de hachage à quelque chose qui n'est pas une entrée dans la table des symboles.

`Badly placed ()'s`

(Parenthèses mal disposées)

(A) Vous avez lancé accidentellement votre script avec *csh* au lieu de Perl. Vérifiez la ligne avec `#!` ou lancez vous-même votre script dans Perl avec *perl nom_script*.

`Bareword "%s" not allowed while "strict subs" in use`

(Mot simple "`%s`" interdit lorsque "`strict subs`" est activé)

(F) Avec « `strict subs` » activé, un mot simple n'est autorisé que comme identificateur de sous-programme, entre accolades ou à la gauche du symbole `=>`. Peut-être devriez-vous déclarer au préalable votre sous-programme ?

`Bareword "%s" refers to nonexistent package`

(Le mot simple "`%s`" se réfère à un paquetage inexistant)

(W bareword) Vous avez utilisé un mot simple qualifié de la forme `Truc::` mais le compilateur n'a pas vu d'autre utilisation de cet espace de noms avant cet endroit. Peut-être devriez-vous déclarer au préalable votre paquetage ?

`Bareword found in conditional`

(Mot simple détecté dans une condition)

(W bareword) Le compilateur a trouvé un mot simple là où il attendait une condition, ce qui indique souvent qu'un `||` ou un `&&` a été analysé comme faisant partie du dernier argument de la construction précédente, par exemple :

```
open FOO || die;
```

Cela peut également indiqué une constante mal orthographiée qui a été interprétée en tant que mot simple :

```
use constant TYPO => 1;
if (TYOP) { print "truc" }
```

Le pragma `strict` est très utile pour éviter de telles erreurs.

`BEGIN failed--compilation aborted`

(Échec de `BEGIN` — compilation abandonnée)

(F) Une exception non interceptable a été levée pendant l'exécution d'un sous-programme `BEGIN`. La compilation s'arrête immédiatement et l'on quitte l'interpréteur.

`BEGIN not safe after errors--compilation aborted`

(`BEGIN` non sécurisé après des erreurs — compilation abandonnée)

(F) Perl a trouvé un sous-programme `BEGIN` (ou une directive `use`, ce qui implique un `BEGIN`) après qu'une ou plusieurs erreurs de compilation sont déjà survenues. Comme l'environnement du `BEGIN` ne peut être garanti (à cause des erreurs) et comme le code qui suit est susceptible de dépendre de son bon fonctionnement, Perl rend la main.

`Binary number > 0b11111111111111111111111111111111 non-portable`

(Nombre binaire >`0b11111111111111111111111111111111`, non portable)

(W portable) Le nombre binaire que vous avez spécifié est plus grand que `2**31-1` (4 294 967 295) et n'est donc pas portable entre les systèmes.

`bind() on closed socket %s`

(`bind` sur une socket fermée *`%s`*)

(W closed) Vous avez essayé de faire un `bind` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur renvoyée par l'appel de `socket` ?

`Bit vector size > 32 non-portable`

(Taille de vecteur de bits >32, non portable)

(W portable) L'utilisation de tailles de vecteurs de bits supérieures à 32 n'est pas portable.

`Bizarre copy of %s in %s`

(Copie étrange de *`%s`* dans *`%s`*)

(P) Perl a détecté une tentative de copie d'une valeur interne qui n'est pas copiable.

`Buffer overflow in prime_env_iter: %s`

(Dépassement du tampon dans `prime_env_iter` : *`%s`*)

(W internal) Il s'agit d'un avertissement spécifique à VMS. Pendant que Perl se préparait à parcourir `%ENV`, il a rencontré un nom logique ou une définition de symbole qui était trop long, il l'a donc tronqué vers la chaîne indiquée.

`Callback called exit`

(La fonction de rappel a appelé `exit`)

(F) Un sous-programme invoqué depuis un paquetage externe *via* `call_sv` s'est terminée en appelant `exit`.

`Can't "goto" out of a pseudo block`

("`goto`" impossible vers l'extérieur d'un pseudo-bloc)

(F) Une instruction `goto` a été exécutée pour quitter ce qui semblerait être un bloc, excepté que ce n'est pas un bloc correct. Ceci se produit habituellement si vous

essayez de sauter hors d'un bloc sort ou d'un sous-programme, ce qui n'a pas de sens.

`Can't "goto" into the middle of a foreach loop`

("goto" impossible vers le milieu d'une boucle `foreach`)

(F) Une instruction `goto` a été exécutée pour sauter au milieu d'une boucle `foreach`. Vous ne pouvez y aller depuis cet endroit.

`Can't "last" outside a loop block`

("last" impossible vers l'extérieur d'un bloc de boucle)

(F) Une instruction `last` a été exécutée pour sortir du bloc courant, excepté ce léger problème qu'il n'existe pas de bloc courant. Remarquez que les blocs `if` ou `else` ne comptent pas comme des blocs de boucle, tout comme tout comme les blocs passés à `sort`, `map` ou `grep`. Vous pouvez habituellement doubler les accolades pour obtenir le même effet, car les accolades intérieures seront considérées comme un bloc qui boucle une seule fois.

`Can't "next" outside a loop block`

("next" impossible vers l'extérieur d'un bloc de boucle)

(F) Une instruction `next` a été exécutée pour recommencer le bloc courant mais il n'existe pas de bloc courant. Remarquez que les blocs `if` ou `else` ne comptent pas comme des blocs de boucle, tout comme les blocs passés à `sort`, `map` ou `grep`. Vous pouvez habituellement doubler les accolades pour obtenir le même effet, car les accolades intérieures seront considérées comme un bloc qui boucle une seule fois.

`Can't read CRTL environ`

(Impossible de lire l'environnement de CRTL)

(S) Il s'agit d'un avertissement spécifique à VMS. Perl a essayé de lire un élément de `%ENV` depuis le tableau d'environnement interne de CRTL et a découvert que le tableau n'existait pas. Vous devez déterminer où CRTL a incorrectement placé son environnement ou définir `PERL_ENV_TABLES` (voir *perlvms*(1)) pour que le tableau d'environnement ne soit pas recherché.

`Can't "redo" outside a loop block`

("redo" impossible vers l'extérieur d'un bloc de boucle)

(F) Une instruction `redo` a été exécutée pour recommencer le bloc courant mais il n'existe pas de bloc courant. Remarquez que les blocs `if` ou `else` ne comptent pas comme des blocs de boucle, tout comme les blocs passés à `sort`, `map` ou `grep`. Vous pouvez habituellement doubler les accolades pour obtenir le même effet, car les accolades intérieures seront considérées comme un bloc qui boucle une seule fois.

`Can't bless non-reference value`

(Impossible de consacrer avec `bless` une valeur qui ne soit pas une référence)

(F) Seules les références en dur peuvent être consacrées avec `bless`. C'est de cette manière que Perl « applique » l'encapsulation des objets.

`Can't break at that line`

(Impossible de s'arrêter à cette ligne)

(S internal) Il s'agit d'un message d'erreur qui n'est affiché que lorsque le pro-

gramme s'exécute avec le débogueur, indiquant que le numéro de ligne spécifié n'est pas l'emplacement d'une instruction à laquelle on peut s'arrêter.

`Can't call method "%s" in empty package "%s"`

(Impossible d'appeler la méthode "*%s*" dans le paquetage vide "*%s*")

(F) Vous avez appelé une méthode correctement et vous avez indiqué correctement le paquetage pris en tant que classe, mais ce paquetage n'a *rien* de défini et n'a pas de méthodes.

`Can't call method "%s" on unblessed reference`

(Impossible d'appeler la méthode "*%s*" sur une référence non consacrée avec `bless`)

(F) Un appel de méthode doit connaître depuis quel paquetage elle est supposé être lancée. Il trouve habituellement ceci dans la référence de l'objet que vous soumettez mais vous n'avez pas fourni de référence d'objet dans ce cas. Une référence n'est pas une référence d'objet jusqu'à ce qu'elle ait été consacrée (avec `bless`).

`Can't call method "%s" without a package or object reference`

(Impossible d'appeler la méthode "*%s*" dans un paquetage ou une référence d'objet)

(F) Vous avez utilisé la syntaxe d'un appel de méthode mais la place réservée à la référence d'objet ou au nom du paquetage contient une expression qui retourne une valeur définie qui n'est ni une référence d'objet ni un nom de paquetage. Quelque chose comme ceci reproduira l'erreur :

```
$MAUVAISE_REF = 42;
traite $MAUVAISE_REF 1,2,3;
$MAUVAISE_REF->traite(1,2,3);
```

`Can't call method "%s" on an undefined value`

(Impossible d'appeler la méthode "*%s*" sur une valeur indéfinie)

(F) Vous avez utilisé la syntaxe d'un appel de méthode, mais la place réservée à la référence d'objet ou au nom du paquetage contient une une valeur indéfinie. Quelque chose comme ceci reproduira l'erreur :

```
$MAUVAISE_REF = undef;
traite $MAUVAISE_REF 1,2,3;
$MAUVAISE_REF->traite(1,2,3);
```

`Can't chdir to %s`

(`chdir` impossible vers *%s*)

(F) Vous avez appelé *perl -x/truc/bidule*, mais */truc/bidule* n'est pas un répertoire dans lequel vous pouvez rentrer avec `chdir`, probablement parce qu'il n'existe pas.

`Can't check filesystem of script "%s" for nosuid`

(Impossible de vérifier le `nosuid` pour le système de fichier du script "*%s*")

(P) Pour une raison ou pour une autre, vous ne pouvez pas vérifier le `nosuid` pour le système de fichier du script.

`Can't coerce %s to integer in %s`

(Impossible de convertir *%s* vers un entier dans *%s*)

(F) On ne peut pas forcer certain types de SV, en particulier les entrées de la table

des symboles (typeglobs) à cesser d'être ce qu'ils sont. Donc vous ne pouvez pas faire des choses telles que :

```
*truc += 1;
```

Vous *pouvez* écrire :

```
$truc = *truc; $truc += 1;
```

mais alors `$truc` ne contient plus de glob.

`Can't coerce %s to number in %s`

(Impossible de convertir *%s* vers un nombre dans *%s*)

(F) On ne peut pas forcer certain types de SVs, en particulier les entrées de la table des symboles (typeglobs) à cesser d'être ce qu'ils sont.

`Can't coerce %s to string in %s`

(Impossible de convertir *%s* vers une chaîne dans *%s*)

(F) On ne peut pas forcer certain types de SVs, en particulier les entrées de la table des symboles (typeglobs) à cesser d'être ce qu'ils sont.

`Can't coerce array into hash`

(Impossible de convertir un tableau vers un hachage)

(F) Vous avez utilisé un tableau là où un hachage était attendu, mais le tableau n'a pas d'informations sur la manière de passer des clés aux indices de tableau. Vous ne pouvez faire cela qu'avec les tableaux qui ont une référence de hachage à l'indice 0.

`Can't create pipe mailbox`

(Impossible de créer un pipe de boîte aux lettres)

(P) Il s'agit d'une erreur spécifique à VMS. Le processus souffre de quotas dépassés ou d'autres problèmes de plomberie.

`Can't declare class for non-scalar %s in "%s"`

(Impossible de déclarer une classe pour le non-scalaire *%s* dans "*%s*")

(S) Actuellement, seules les variables scalaires peuvent être déclarée avec un qualificateur de classe spécifique dans une déclaration `my` ou `our`. Les sémantiques peuvent être étendues à d'autres types de variables à l'avenir.

`Can't declare %s in "%s"`

(Impossible de déclarer *%s* dans "*%s*")

(F) Seuls les scalaires, les tableaux et les hachages peuvent être déclarés comme variables `my` ou `our`. Ils doivent avoir un identificateur ordinaire comme nom.

`Can't do inplace edit on %s: %s`

(Impossible d'éditer sur place *%s*)

(S inplace) La création du nouveau fichier a échoué à cause de la raison indiquée.

`Can't do inplace edit without backup`

(Impossible d'éditer sur place sans copie de sauvegarde)

(F) Vous êtes sur un système comme MS-DOS qui s'embrouille si vous essayez de lire un fichier supprimé (mais toujours ouvert). Vous devez écrire *-i.bak*, ou quelque chose comme ça.

`Can't do inplace edit: %s would not be unique`

(Impossible d'éditer sur place : *%s* ne serait pas unique)

(S inplace) Votre système de fichier ne supporte pas les noms de fichiers plus longs que 14 caractères et Perl n'a pas pu créer un nom de fichier unique durant l'édition sur place avec l'option **-i**. Ce fichier a été ignoré.

`Can't do inplace edit: %s is not a regular file`

(Impossible d'éditer sur place : *%s* n'est pas un fichier ordinaire)

(S inplace) Vous avez essayé d'utiliser l'option **-i** sur un fichier spécial, comme un fichier dans */dev* ou une FIFO. Le fichier a été ignoré.

`Can't do setegid!`

(`setegid` impossible !)

(P) L'appel à `setegid` a échoué pour une raison ou pour une autre dans l'émulateur setuid de *suidperl*.

`Can't do seteuid!`

(`seteuid` impossible !)

(P) L'émulateur setuid de *suidperl* a échoué pour une raison ou pour une autre.

`Can't do setuid`

(`setuid` impossible)

(F) Cela veut généralement dire que le *perl* ordinaire a essayé de faire `exec` *suidperl* pour émuler le setuid, mais n'a pas pu faire le `exec`. Il cherche un nom de la forme *sperl5.000* dans le même répertoire que celui où réside l'exécutable perl sous le nom *perl5.000*, généralement */usr/local/bin* sur les machines Unix. Si le fichier s'y trouve, vérifiez les permissions d'exécution. S'il n'est pas là, demandez à votre administrateur système pourquoi.

`Can't do waitpid with flags`

(`waitpid` impossible avec les drapeaux)

(F) Cette machine n'a pas ni `waitpid` ni `wait4`, ainsi seul `waitpid` sans aucun drapeau est émulé.

`Can't do {n,m} with n > m`

(Impossible de faire `{n,m}` avec `n > m`)

(F) Les minima doivent être inférieurs ou égaux aux maxima. Si vous voulez vraiment que votre expression régulière corresponde 0 fois, faites juste `{0}`.

`Can't emulate -%s on #! line`

(Impossible d'émuler *-%s* sur la ligne `#!`)

(F) La ligne `#!` spécifie une option qui n'a pas de sens à cet endroit. Par exemple, il serait un peu idiot de mettre l'option **-x** sur cette ligne `#!`.

`Can't exec "%s": %s`

(`exec` de *"%s"* impossible : *%s*)

(W exec) L'appel à `system`, `exec`, ou `open` sur un pipe n'a pu s'exécuter pour la raison indiquée. Les raisons typiques incluent de mauvaises permissions sur le fichier,

que le fichier n'a pu être trouvé dans `$ENV{PATH}`, que l'exécutable en question a été compilé pour une autre architecture ou que la ligne `#!` pointe vers un interpréteur qui ne peux pas être lancé pour des raisons similaires. (Ou peut-être votre système ne supporte pas du tout `#!`).

`Can't exec %s`

(`exec %s` impossible)

(F) Perl a essayé d'exécuter le programme indiqué pour vous car c'est ce qui est spécifié sur la ligne `#!`. Si ce n'est pas ce que vous vouliez, vous devrez mentionner `perl` quelque part sur la ligne `#!`.

`Can't execute %s`

(Impossible d'exécuter *%s*)

(F) Vous avez utilisé l'option **-S**, mais les copies du script à exécuter trouvées dans le `PATH` n'ont pas les bonnes permissions.

`Can't find %s on PATH, '.' not in PATH`

(Impossible de trouver *%s* dans le `PATH`, `"."` n'est pas dans le `PATH`)

(F) Vous avez utilisé l'option **-S**, mais le script à exécuter ne peut être trouvé dans le `PATH` ou tout au moins avec des permissions incorrectes. Le script existe dans le répertoire courant, mais le `PATH` l'empêche de se lancer.

`Can't find %s on PATH`

(Impossible de trouver *%s* dans le `PATH`)

(F) Vous utilisez l'option **-S**, mais le script à exécuter ne peut être trouvé dans le `PATH`.

`Can't find an opnumber for "%s"`

(Impossible de trouver un numéro d'opération pour "*%s*")

(F) Une chaîne de la forme `CORE::`*mot* a été donnée à `prototype` mais il n'existe pas de fonction interne nommée *mot*.

`Can't find label %s`

(Impossible de trouver l'étiquette *%s*)

(F) Vous avez dit par un `goto` d'aller à une étiquette qui n'est mentionnée à aucun endroit où il nous est possible d'aller.

`Can't find string terminator %s anywhere before EOF`

(Impossible de trouver la fin de la chaîne *%s* quelque part avant `EOF`)

(F) Les chaînes de caractères en Perl peuvent s'étirer sur plusieurs lignes. Ce message signifie que vous avez oublié le délimitateur fermant. Comme les parenthèses protégées par des apostrophes comptent pour un niveau, il manque dans l'exemple suivant la parenthèse finale :

```
print q(Le caractère '(' commence une nouvelle citation.);
```

Si vous obtenez cette erreur depuis un document « ici-même » (*N.d.T.* : *here document*), vous devez avoir mis des espaces blancs non visibles avant ou après la marque de fermeture. Un bon éditeur pour les programmeurs devrait pouvoir vous aider à trouver ces caractères.

`Can't fork`

(`fork` impossible)

(F) Une erreur fatale est survenue en essayant de faire un `fork`.

`Can't get filespec - stale stat buffer?`

(Impossible d'obtenir la spécification du fichier — tampon `stat` périmé ?)

(S) Il s'agit d'un avertissement spécifique à VMS. Ceci arrive à cause des différences entre les contrôles d'accès sous VMS et sous le modèle Unix qui est celui que Perl présuppose. Sous VMS, les contrôles d'accès sont fait par le nom de fichier, plutôt que par les bits du tampon de `stat`, ce qui fait que les ACL et les autres protections peuvent être pris en compte. Malheureusement, Perl présuppose que le tampon de `stat` contient toutes les informations nécessaires et le fournit, à la place de la spécification du fichier, à la fonction de contrôle d'accès. Il va essayer d'obtenir les spécifications du fichier en utilisant le nom du périphérique et le FID présents dans le tampon de `stat`, mais si vous avez fait ensuite un appel à la fonction `stat` de CRTL, cela ne fonctionnera pas car le nom du périphérique est écrasé à chaque appel. Si cet avertissement apparaît, la recherche de nom a échoué et la fonction de contrôle d'accès a rendu la main en retournant une valeur fausse, histoire d'être un peu conservateur. (Remarque : la fonction de contrôle d'accès connaît l'opérateur `stat` de Perl et les tests de fichiers, vous ne devriez donc jamais voir cet avertissement en réponse à une commande Perl ; il ne se produit que si du code interne prend les tampons de `stat` à la légère.)

`Can't get pipe mailbox device name`

(Impossible d'obtenir le nom de périphérique d'un pipe de boîte aux lettres)

(P) Il s'agit d'une erreur spécifique à VMS. Après avoir créé une boîte aux lettres se comportant comme un pipe, Perl ne peut obtenir son nom pour un usage ultérieur.

`Can't get SYSGEN parameter value for MAXBUF`

(Impossible d'obtenir la valeur du paramètre `SYSGEN` pour `MAXBUF`)

(P) Il s'agit d'une erreur spécifique à VMS. Perl a demande à `$GETSYI` quelle taille vous voulez pour vos tampons de boîtes aux lettres et il n'a pas obtenu de réponse.

`Can't goto subroutine outside a subroutine`

(`goto` vers un sous-programme impossible à l'extérieur d'un sous-programme)

(F) L'appel hautement magique `goto` *`SOUS_PROGRAMME`* ne peut que remplacer l'appel d'une fonction par une autre. Il ne peut en fabriquer une brute de fonderie. En général, vous ne devriez de toute façon l'appeler que depuis une fonction d'`AUTOLOAD`.

`Can't goto subroutine from an eval-string`

(`goto` vers un sous-programme impossible depuis une chaîne `eval`)

(F) L'appel `goto` *`SOUS_PROGRAMME`* ne peut être utilisé pour sortir d'une chaîne `eval`. (Vous pouvez l'utilisez pour sortir d'un `eval` *`BLOC`* mais ce n'est probablement pas ce que vous voulez.)

`Can't ignore signal CHLD, forcing to default`

(Impossible d'ignorer le signal CHLD, la valeur par défaut est forcée)

(W signal) Perl a détecté qu'il avait été lancé avec le signal SIGCHLD (quelquefois appelé SIGCLD) désactivé. Comme la désactivation de ce signal va interférer avec une détermination correcte du statut de sortie des processus fils, Perl a remis le signal à sa valeur par défaut. Cette situation indique généralement que le programme père sous lequel Perl pourrait tourner (exemple : *cron*) est très peu précautionneux.

`Can't localize through a reference`

(Impossible de rendre locale *via* une référence)

(F) Vous avez écrit quelque chose comme `local $$ref`, ce que Perl ne sait actuellement pas gérer car quand il va restaurer l'ancienne valeur de ce vers quoi `$ref` pointait après que la portée du `local` est terminée, il ne peut être sûr que `$ref` soit toujours une référence.

`Can't localize lexical variable %s`

(Impossible de rendre locale la variable *%s*)

(F) Vous avez utilisé `local` sur une variable qui a déjà été déclarée auparavant comme une variable lexicale à l'aide de `my`. Ceci n'est pas permis. Si vous voulez rendre locale une variable de paquetage du même nom, qualifiez-la avec le nom du paquetage.

`Can't localize pseudo-hash element`

(Impossible de rendre local un élément de pseudo-hachage)

(F) Vous avez écrit quelque chose comme `local $tab->{'clef'}`, où `$tab` est une référence vers un pseudo-hachage. Cela n'a pas encore été implémenté mais vous pouvez obtenir un effet similaire en rendant local l'élément du tableau correspondant directement - `local $tab->[$tab->[0]{'clef'}]`.

`Can't locate auto/%s.al in @INC`

(Impossible de trouver `auto/%s.al` dans `@INC`)

(F) Une fonction (ou une méthode) a été appelée dans un paquetage qui autorise le chargement automatique mais il n'existe pas de fonction à charger automatiquement. Les causes les plus probables sont une faute de frappe dans le nom de la fonction ou de la méthode ou un échec dans l'`AutoSplit` du fichier, disons, lors du `make install`.

`Can't locate %s`

(Impossible de trouver *%s*)

(F) Vous avez écrit de faire un `do` (ou un `require` ou un `use`) sur un fichier qui n'a pu être trouvé. Perl cherche le fichier dans tous les emplacements mentionnés dans `@INC`, à moins que le nom du fichier ne précise son chemin complet. Peut-être avez-vous besoin de positionner la variable d'environnement `PERL5LIB` ou `PERL5OPT` pour indiquer où se trouve la bibliothèque ou il est possible que le script ait besoin d'ajouter le nom de la bibliothèque à `@INC`. Ou enfin, vous avez peut-être seulement mal épelé le nom du fichier.

`Can't locate object method "%s" via package "%s"`

(Impossible de trouver la méthode objet "*%s*" *via* le paquetage *%s*)

(F) Vous avez appelé une méthode correctement et vous avez correctement indiqué un paquetage fonctionnant comme une classe mais ce paquetage ne définit pas cette méthode-là, ni aucune de ses classes de base.

`Can't locate package %s for @%s::ISA`

(Impossible de trouver le paquetage *%s* pour `@%s::ISA`)

(W syntax) Le tableau `@ISA` contient le nom d'un autre paquetage qui semble ne pas exister.

`Can't make list assignment to \%ENV on this system`

(Impossible d'assigner une liste à `\%ENV` sur ce système)

(F) L'affectation d'une liste à `%ENV` n'est pas supporté sur certains systèmes, notamment VMS.

`Can't modify %s in %s`

(Impossible de modifier *%s* dans *%s*)

(F) Vous n'êtes pas autorisé à faire d'affectation sur l'élément indiqué, ou à essayer de le changer autrement, comme avec une incrémentation automatique.

`Can't modify non-lvalue subroutine call`

(Impossible de modifier un appel de sous-programme qui ne soit pas une lvalue)

(F) Les sous-programmes destinés à être utilisés dans un contexte de lvalue doivent être déclarés en tant que tels.

`Can't modify nonexistent substring`

(Impossible de modifier une sous-chaîne inexistante)

(P) La routine interne qui fait l'affectation `substr` a capturé un NULL.

`Can't msgrcv to read-only var`

(`msgrcv` d'une var en lecture seule impossible)

(F) La cible d'un `msgrcv` doit être modifiable pour être utilisée comme un tampon de réception.

`Can't open %s: %s`

(Impossible d'ouvrir *%s*: *%s*)

(S inplace) L'ouverture implicite d'un fichier *via* le handle de fichier `<>`, soit implicitement *via* les options de la ligne de commande **`-n`** ou **`-p`**, soit explicitement, a échoué à cause de la raison indiquée. Habituellement c'est parce que vous n'avez pas les permissions en lecture sur le fichier que vous avez indiqué sur la ligne de commande.

`Can't open bidirectional pipe`

(Impossible d'ouvrir un pipe bidirectionnel)

(W pipe) Vous avez essayé d'écrire `open(CMD, "|cmd|")`, ce qui n'est pas supporté. Vous pouvez essayer l'un des nombreux modules de la bibliothèque Perl pour faire cela, comme `IPC::Open2`. Ou alors, dirigez la sortie du pipe dans un fichier en utilisant `>` et ensuite lisez-le dans un handle de fichier différent.

`Can't open error file %s as stderr`

(Impossible d'ouvrir le fichier d'erreur *%s* comme `stderr`)

(F) Il s'agit d'une erreur spécifique à VMS. Perl fait ses propres redirections de la ligne de commande et ne peut ouvrir le fichier spécifié après `2>` ou `2>>` sur la ligne de commande pour y écrire.

`Can't open input file %s as stdin`

(Impossible d'ouvrir le fichier d'entrée *%s* comme `stdin`)

(F) Il s'agit d'une erreur spécifique à VMS. Perl fait ses propres redirections de la ligne de commande et ne peut ouvrir le fichier spécifié après `<` sur la ligne de commande pour y lire.

`Can't open output file %s as stdout`

(Impossible d'ouvrir le fichier de sortie *%s* comme `stdout`)

(F) Il s'agit d'une erreur spécifique à VMS. Perl fait ses propres redirections de la ligne de commande et ne peut ouvrir le fichier spécifié après `>` ou `>>` sur la ligne de commande pour y écrire.

`Can't open output pipe (name: %s)`

(Impossible d'ouvrir le pipe de sortie (nom : *%s*))

(P) Il s'agit d'une erreur spécifique à VMS. Perl fait ses propres redirections de la ligne de commande et ne peut ouvrir le pipe dans lequel envoyer les données destinées à `STDOUT`.

`Can't open perl script "%s": %s`

(Impossible d'ouvrir le script perl "*%s*" : *%s*)

(F) Le script que vous avez spécifié ne peut être ouvert pour la raison indiquée.

`Can't redefine active sort subroutine %s`

(Impossible de redéfinir le sous-programme de `sort` actif)

(F) Perl optimise la gestion interne des sous-programmes de `sort` et garde des pointeurs vers ceux-ci. Vous avez essayé de redéfinir un de ces sous-programmes de `sort` alors qu'il était actuellement actif, ce qui n'est pas permis. Si vous voulez vraiment faire cela, vous devez écrire `sort { &fonc } @x` au lieu de `sort fonc @x`.

`Can't remove %s: %s, skipping file`

(Impossible de supprimer *%s*: *%s*, fichier passé)

(S inplace) Vous avez demandé une édition sur place sans créer un fichier de sauvegarde. Perl a été incapable de supprimer le fichier original pour le remplacer avec celui modifié. Le fichier a été laissé inchangé.

`Can't rename %s to %s: %s, skipping file`

(Impossible de renommer *%s* en *%s*: *%s*, fichier passé)

(S inplace) Le renommage effectué par l'option **`-i`** a échoué pour une raison ou pour une autre, probablement parce que vous n'avez pas le droit d'écrire dans le répertoire.

`Can't reopen input pipe (name: %s) in binary mode`

(Impossible de rouvrir le pipe d'entrée (nom : *%s*) en mode binaire)

(P) Il s'agit d'une erreur spécifique à VMS. Perl pensait que `STDIN` était un pipe et a essayé de le rouvrir pour accepter des données binaires. Hélas, cela a échoué.

`Can't resolve method "%s" overloading "%s" in paquetage "%s"`

(Impossible de résoudre la méthode "*%s*" surchargeant "*%s*" dans le paquetage "*%s*")

(F|P) Une erreur s'est produite lors de la résolution de la surcharge spécifiée par un nom de méthode (à l'inverse d'une référence de sous-programme) : aucune méthode de ce nom ne peut être appelée *via* ce paquetage. Si le nom de la méthode est `???`, il s'agit d'une erreur interne.

`Can't reswap uid and euid`

(Impossible d'échanger à nouveau l'uid et l'euid)

(P) L'appel `setreuid` a échoué pour une raison ou pour une autre dans l'émulateur setuid de *suidperl*.

`Can't return outside a subroutine`

(`return` impossible à l'extérieur d'un sous-programme)

(F) L'instruction `return` a été exécutée dans le fil principal du code, c'est-à-dire là où il n'y a aucun appel de sous-programme dont on peut revenir avec `return`.

`Can't return %s from lvalue subroutine`

(Impossible de renvoyer *%s* d'un sous-programme utilisé comme lvalue)

(F) Perl a détecté que la valeur renvoyée par un sous-programme utilisé comme lvalue est illégale (constante ou variable temporaire). C'est interdit.

`Can't stat script "%s"`

(`stat` du script impossible : *%s*)

(P) Pour une raison ou pour une autre, vous ne pouvez faire de `fstat` sur le script même si vous l'avez déjà ouvert. Bizarre.

`Can't swap uid and euid`

(Impossible d'échanger l'uid et l'euid)

(P) L'appel `setreuid` a échoué pour une raison ou pour une autre dans l'émulateur setuid de *suidperl*.

`Can't take log of %g`

(Impossible de calculer le log de *%g*)

(F) Pour les nombres réels ordinaires, vous ne pouvez calculer le logarithme d'un nombre négatif ou nul. Il existe toutefois un paquetage `Math::Complex` livré en standard avec Perl, si c'est vraiment ce que vous voulez faire cela avec les nombres négatifs.

`Can't take sqrt of %g`

(Impossible de calculer la racine carrée de *%g*)

(F) Pour les nombres réels ordinaires, vous ne pouvez calculer la racine carrée d'un

nombre négatif. Il existe toutefois le paquetage `Math::Complex` livré en standard avec Perl, si vous voulez vraiment faire cela.

`Can't undef active subroutine`

(`undef` du sous-programme actif impossible)

(F) Vous ne pouvez rendre indéfinie une fonction qui est actuellement utilisée. Cependant, vous pouvez la redéfinir pendant qu'elle tourne et même faire un `undef` sur le sous-programme redéfini pendant que l'ancien tourne. Maintenant, c'est à vous de voir.

`Can't unshift`

(`unshift` impossible)

(F) Vous avez essayé de faire un `unshift` sur un tableau « irréel » qui ne peut accepter cette opération, comme la pile principale de Perl.

`Can't upgrade that kind of scalar`

(Impossible de mettre à niveau ce type de scalaire)

(P) La routine interne `sv_upgrade` a ajouté des « membres » à un SV, ce qui fait qu'il se trouve dans une sorte de SV plus spécialisée. Les différents types les plus élevés de SV sont cependant si spécialisés, qu'ils ne peuvent être converti entre eux. Ce message indique qu'une telle conversion a été tentée.

`Can't upgrade to undef`

(Impossible de mettre à niveau vers `undef`)

(P) Le SV indéfini représente le niveau zéro de l'évolutivité. L'évolution vers `undef` indique une erreur dans le code appelant `sv_upgrade`.

`Can't use %%! because Errno.pm is not available`

(Impossible d'utiliser `%%!` car `Errno.pm` n'est pas disponible)

(F) La première fois que le hachage `%!` est utilisé, Perl charge automatiquement le module `Errno`. On attend du module `Errno` qu'il se lie avec `tie` au hachage `%!` pour fournir des noms symboliques pour les valeurs des numéros d'erreur de `$!`.

`Can't use "my %s" in sort comparison`

(Impossible d'utiliser `"my %s"` dans les comparaisons de `sort`)

(F) Les variables globales `$a` et `$b` sont réservées pour les comparaisons de `sort`. Vous avez mentionné `$a` ou `$b` dans la même ligne que les opérateurs `<=>` ou `cmp` et la variable avait été déclarée précédemment en tant que variable lexicale. Soit qualifiez la variable de `sort` avec le nom de paquetage, soit renommez la variable lexicale.

`Can't use %s for loop variable`

(Impossible d'utiliser *`%s`* comme variable de boucle)

(F) Seule une variable scalaire simple peut être utilisée comme variable de boucle dans un `foreach`.

`Can't use %s ref as %s ref`

(Impossible d'utiliser la ref *`%s`* en tant que ref de *`%s`*)

(F) Vous avez mélangé vos types de références. Vous devez déréférencer la référence

du type nécessaire. Vous pouvez utilisez la fonction `ref` pour tester le type de la référence, si besoin.

`Can't use \%c to mean $%c in expression`

(Impossible d'utiliser `\%c` pour signifier `$%c` dans une expression)

(W syntax) Dans une expression ordinaire, l'antislash est un opérateur unaire qui crée une référence vers son argument. L'utilisation d'un antislash pour indiquer une référence arrière vers une sous-chaîne réussissant la correspondance n'est valide qu'à l'intérieur du motif de l'expression régulière. Essayer de faire cela dans un code ordinaire Perl produit une valeur qui s'affiche comme `SCALAR(0xdecaf)`. Utilisez la forme `$1` à la place.

`Can't use bareword ("%s") as %s ref while "strict refs" in use`

(Impossible d'utiliser un mot simple ("`%s`") en tant que ref de `%s` pendant que "`strict refs`" est actif)

(F) Les références en dur sont les seules autorisées par `strict refs`. Les références symboliques sont interdites.

`Can't use string ("%s") as %s ref while "strict refs" in use`

(Impossible d'utiliser une chaîne ("`%s`") en tant que ref de `%s` pendant que "`strict refs`" est actif)

(F) Les références en dur sont les seules autorisées par `strict refs`. Les références symboliques sont interdites.

`Can't use an undefined value as %s reference`

(Impossible d'utiliser une valeur indéfinie comme référence de `%s`)

(F) Une valeur utilisée soit comme une référence en dur, soit comme une référence symbolique doit avoir une valeur définie. Cela aide à débloquer certaines erreurs insidieuses.

`Can't use global %s in "my"`

(Impossible d'utiliser la variable globale `%s` dans "`my`")

(F) Vous avez essayé de déclarer une variable magique en tant que variable lexicale. Cela n'est pas permis car la variable magique ne peut être liée qu'à un seul endroit (nommé variable globale) et il serait incroyablement confus d'avoir des variables dans votre programme qui ressemblent à des variables magiques mais qui n'en sont pas.

`Can't use subscript on %s`

(Impossible d'utiliser un indice sur `%s`)

(F) Le compilateur a essayé d'interpréter une expression entre crochets en tant qu'indice. Mais à gauche des crochets, il y avait une expression qui ne ressemblait pas à une référence de tableau, ni à quoi que ce soit que l'on puisse indicer.

`Can't weaken a nonreference`

(Impossible d'affaiblir une non-référence)

(F) Vous avez tenté d'affaiblir quelque chose qui n'était pas une référence. Seules les références peuvent être affaiblies.

`Can't x= to read-only value`

(`x=` impossible sur une valeur en lecture seule)

(F) Vous avez essayé de répéter une valeur constante (souvent la valeur indéfinie) avec un opérateur d'affectation, ce qui implique de modifier la valeur elle même. Peut-être devriez-vous copier la valeur dans une variable temporaire, et répéter cela.

`Character class [:%s:] unknown`

(Classe de caractères `[:%s:]` inconnue)

(F) La classe dans la syntaxe pour une classe de caractères `[: :]` est inconnue.

`Character class syntax [%s] belongs inside character classes`

(La syntaxe `[%s]` pour une classe de caractères appartient aux classes de caractères)

(W unsafe) Les constructions de classes de caractères `[: :]`, `[= =]` et `[. .]` se rangent dans les classes de caractères, par exemple : `/[012[:alpha:]345]/`. Remarquez que les constructions `[= =]` et `[. .]` ne sont pas implémentées actuellement ; ce sont juste des places réservées pour de futures extensions.

`Character class syntax [. .] is reserved for future extensions`

(Syntaxe `[. .]` pour une classe de caractère réservée pour de futures extensions)

(W regexp) À l'intérieur des classes de caractères dans les expressions régulières (`[]`), la syntaxe commençant par `[.` et se terminant par `.]` est réservée pour de futures extensions. Si vous avez besoin de représenter de telles séquences de caractères à l'intérieur d'une classe de caractères dans une expression régulière, protégez simplement les crochets avec un anti-slash : `\[.` et `.\]`.

`Character class syntax [= =] is reserved for future extensions`

(Syntaxe `[= =]` pour une classe de caractère réservée pour de futures extensions)

(W regexp) À l'intérieur des classes de caractères dans les expressions régulières (`[]`), la syntaxe commençant par `[=` et se terminant par `=]` est réservée pour de futures extensions. Si vous avez besoin de représenter de telles séquences de caractères à l'intérieur d'une classe de caractères dans une expression régulière, protégez simplement les crochets avec un anti-slash : `\[.` et `.\]`.

`chmod() mode argument is missing initial 0`

(Il manque le `0` au début de l'argument représentant le mode de `chmod()`)

(W chmod) Un utilisateur novice aura tendance à écrire :

```
chmod 777, $fichier
```

en ne réalisant pas que 777 sera interprété comme un nombre décimal, équivalent à 01411. Les constantes octales sont introduites en les faisant débuter par un 0 en Perl, tout comme en C.

`Close on unopened file <%s>`

(Ferme un fichier non ouvert *<%s>*)

(W) Vous avez essayé de fermer un descripteur de fichier qui n'avait jamais été ouvert.

`Compilation failed in require`

(Échec de la compilation dans `require`)

(F) Perl n'a pas pu compiler le fichier spécifié dans une instruction `require`. Perl utilise ce message générique lorsqu'aucune des erreurs rencontrées n'est assez grave pour arrêter la compilation immédiatement.

`Complex regular subexpression recursion limit (%d) exceeded`

(Limite de récursion pour les expressions régulières complexes (*%d*) atteinte)

(W regexp) Le moteur d'expressions régulières utilise la récursion dans les situations complexes où les retours arrières sont nécessaires. La profondeur de la récursion est limitée à 32 766, ou peut-être moins sur les architectures où la pile ne peut croître arbitrairement. (Les situations « simples » et « moyennes » peuvent être gérées sans récursion et ne sont pas sujettes à une limite.) Essayez de raccourcir la chaîne examinée, faites des boucles dans le code Perl (exemple : avec `while`) plutôt qu'avec le moteur d'expressions régulières ou réécrivez l'expression régulière pour qu'elle soit plus simple ou qu'elle contienne moins de retours arrières.

`connect() on closed socket %s`

(`connect()` sur une socket fermée *%s*)

(W closed) Vous avez essayé de faire un `connect` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur retournée par l'appel de `socket`?

`Constant is not %s reference`

(La constante n'est pas une référence de *%s*)

(F) Une valeur constante (déclarée peut-être en utilisant le pragma `use constant`) est déréférencée mais équivaut à une référence d'un mauvais type. Le message indique le type de référence qui était attendu. Ceci indique habituellement une erreur de syntaxe en déréférençant la valeur constante.

`Constant subroutine %s redefined`

(Sous-programme constant *%s* redéfini)

(S|W redefine) Vous avez redéfini un sous-programme qui a été désigné auparavant comme étant en ligne (*inlining*).

`Constant subroutine %s undefined`

(Sous-programme constant *%s* indéfini)

(W misc) Vous avez rendu indéfini un sous-programme qui a été désigné auparavant comme étant en ligne (*inlining*).

`constant(%s): %s`

(`constant(%s)`: *%s*)

(F) L'analyseur syntaxique a trouvé des incohérences soit en tentant de définir une constante surchargée, soit en essayant de trouver le nom de caractère spécifié dans la séquence d'échappement `\N{...}`. Peut-être avez-vous oublié de charger les pragmas correspondants `overload` ou `charnames`.

`Copy method did not return a reference`

(La méthode de copie n'a pas renvoyé de référence)

(F) La méthode qui surcharge = est boguée.

`CORE::%s is not a keyword`

(`CORE::%s` n'est pas un mot clef)

(F) L'espace de noms `CORE::` est réservé pour les mots clefs de Perl.

`Corrupt malloc ptr 0x%lx at 0x%lx`

(Pointeur de `malloc` `0x%lx` corrompu en `0x%lx`)

(P) Le paquetage `malloc` distribué avec Perl a eu une erreur interne.

`corrupted regexp pointers`

(Pointeurs d'expressions régulières corrompus)

(P) Le moteur d'expressions régulières est devenu confus parce que le compilateur d'expressions régulières lui a donné.

`corrupted regexp program`

(Programme d'expressions régulières corrompu)

(P) On a passé au moteur d'expressions régulières un programme d'expression régulières sans nombre magique valide.

`Deep recursion on subroutine "%s"`

(Récursion profonde pour le sous-programme "`%s`")

(W recursion) Ce sous-programme s'est appelé lui-même (directement ou indirectement) 100 fois de plus qu'il n'est revenu. Ceci indique probablement une récursion infinie, à moins que vous n'écriviez un étrange programme de benchmark, auquel cas cela indique quelque chose d'autre.

`defined(@array) is deprecated`

(`defined(@tableau)` est déprécié)

(D deprecated) `defined` n'est habituellement pas très utile sur les tableaux car il teste la présence d'une valeur *scalaire* indéfinie. Si vous voulez voir si le tableau est vide, vous n'avez qu'à utiliser `if (@tableau) { # pas vide }`.

`defined(%hash) is deprecated`

(`defined(%hachage)` est déprécié)

(D deprecated) `defined` n'est habituellement pas très utile sur les hachages car il teste la présence d'une valeur *scalaire* indéfinie. Si vous voulez voir si le hachage est vide, vous n'avez qu'à utiliser `if (%hachage) { # pas vide }`.

`Delimiter for here document is too long`

(Le délimiteur pour le document « ici-même » est trop long)

(F) Dans une construction de document « ici-même », comme `<<TRUC`, l'étiquette `TRUC` est trop longue pour que Perl puisse la gérer. Il faut que vous soyez sérieusement tordu pour écrire un code qui déclenche cette erreur.

`Did not produce a valid header`

(N'a pas produit un en-tête valide)

Voir `Server error`.

`(Did you mean &%s instead?)`

(Avez-vous plutôt voulu dire `&%s` ?)

(W) Vous avez probablement fait référence à un sous-programme importé `&TRUC` en faisant `$TRUC` ou quelque chose du genre.

`(Did you mean "local" instead of "our"?)`

(Avez-vous voulu dire `"local"` plutôt que `"our"` ?)

(W misc) Souvenez-vous que `our` ne rend pas locale la variable déclarée comme globale. Vous l'avez redéclarée dans la même portée lexicale, ce qui semble superflu.

`(Did you mean $ or @ instead of %?)`

(Avez-vous voulu dire `$` ou `@` plutôt que `%` ?)

(W) Vous avez probablement écrit `%hachage{$clef}` alors que vous pensiez `$hachage{$clef}` ou `@hachage{@clefs}`. D'un autre côté, peut-être avez-vous juste voulu dire `%hachage` et vous vous êtes laissés emporter.

`Died`

(Mort)

(F) Vous avez appelé à `die` une chaîne vide (l'équivalent de `die ""`) ou vous l'avez appelé sans arguments et à la fois `$@` et `$_` étaient vides.

`(Do you need to predeclare %s?)`

(Avez-vous besoin de prédéclarer `%s`)

(S) Il s'agit d'une supposition donnée en complément du message « `%s found where operator expected` ». Cela veut souvent dire qu'un nom de sous-programme ou de module a été référencé alors qu'il n'a pas encore été défini. C'est peut être un problème d'ordre dans votre fichier ou parce qu'il manque une instruction `sub`, `package`, `require` ou `use`. Si vous référencez quelque chose qui n'est pas encore défini, en fait vous n'avez pas à définir le sous-programme ou le paquetage avant cet endroit. Vous pouvez utilisez un `sub truc;` ou un `package TRUC;` vide pour entrer une déclaration « par avance ».

`Document contains no data`

(Le document ne contient pas de données)

Voir `Server error`.

`Don't know how to handle magic of type '%s'`

(Ne sait pas comment gérer la magie du type '`%s`')

(P) On a jeté un sort sur la gestion interne des variables magiques.

`do_study: out of memory`

(`do_study` : débordement de mémoire)

(P) Ce message aurait plutôt dû être capturé par `safemalloc`.

`Duplicate free() ignored`

(Duplication de `free()` ignorée)

(S malloc) Une routine interne a appelé `free` sur quelque chose qui a déjà été libéré.

`elseif should be elsif`

(`elseif` devrait être `elsif`)

(S) Il n'y a pas de mot-clef « `elseif` » en Perl car Larry pense que c'est très laid. Votre code sera interprété comme une tentative d'appel à la méthode nommée `elseif` pour la classe renvoyée par le bloc qui suit. Ce n'est sûrement pas ce que vous voulez.

`entering effective %s failed`

(L'entrée dans *`%s`* effectif a échoué)

(F) Alors que le pragma `use filetest` était actif, l'échange entre les UID ou les GID réels et effectifs a échoué.

`Error converting file specification %s`

(Erreur en convertissant la spécification de fichier *`%s`*)

(F) Il s'agit d'une erreur spécifique à VMS. Comme Perl doit gérer des spécifications de fichiers dans les syntaxes VMS ou Unix, il convertit celles-ci dans un format unique lorsqu'il doit les traiter directement. Soit vous avez passé des spécifications de fichier invalides, soit vous avez trouvé un cas que les routines de conversion ne gèrent pas. Zut !

`%s``: Eval-group in insecure regular expression`

(*`%s`* : Groupe `eval` dans une expression régulière non sécurisée)

(F) Perl a détecté une donnée marquée en essayant de compiler une expression régulière qui contient l'assertion de longueur zéro (`?{ ... }`) , ce qui n'est pas sécurisé.

`%s``: Eval-group not allowed, use re 'eval'`

(*`%s`* : Groupe `eval` interdit, utilisez `use re 'eval'`)

(F) Une expression régulière contenait l'assertion de longueur nulle (`?{ ... }`), mais cette construction est seulement permise quand le pragma `use re 'eval'` est actif.

`%s``: Eval-group not allowed at run time`

(*`%s`* : Groupe `eval` interdit à l'exécution)

(F) Perl a essayé de compiler une expression régulière contenant l'assertion de longueur nulle (`?{ ... }`) à l'exécution, comme il le ferait si le motif contenait des valeurs interpolées. Comme c'est un risque de sécurité, cela n'est pas permis. Si vous insistez, vous pouvez toujours le faire en construisant explicitement votre motif depuis une chaîne interpolée à l'exécution et l'utiliser dans un `eval`.

`Excessively long <> operator`

(Opérateur `<>` excessivement long)

(F) Le contenu d'un opérateur `<>` ne doit pas excéder la taille maximum d'un identificateur Perl. Si vous essayez seulement d'obtenir les extensions d'une longue liste de fichiers, essayez d'utiliser l'opérateur `glob` ou mettez les noms de fichiers dans une variable et faites un `glob` dessus.

`Execution of %s aborted due to compilation errors`

(Exécution de *%s* abandonnée à cause d'erreurs de compilation)

(F) Le message de résumé final lorsqu'une compilation Perl échoue.

`Exiting eval via %s`

(Sortie d'un `eval` *via %s*)

(W exiting) Vous êtes sorti d'un `eval` d'une façon non conventionnelle, comme un `goto` ou une instruction de contrôle de boucle.

`Exiting format via %s`

(Sortie d'un `format` *via %s*)

(W exiting) Vous êtes sorti d'un `format` d'une façon non conventionnelle, comme un `goto` ou une instruction de contrôle de boucle.

`Exiting pseudoblock via %s`

(Sortie d'un pseudo-bloc *via %s*)

(W exiting) Vous êtes sorti d'une construction tout à fait spéciale de bloc (comme un bloc ou un sous-programme de `sort`) d'une façon non conventionnelle, comme un `goto`, ou une instruction de contrôle de boucle.

`Exiting subroutine via %s`

(Sortie d'un sous-programme *via %s*)

(W exiting) Vous êtes sorti d'un sous-programme d'une façon non conventionnelle, comme un `goto` ou une instruction de contrôle de boucle.

`Exiting substitution via %s`

(Sortie d'une substitution *via %s*)

(W exiting) Vous êtes sorti d'une substitution d'une façon non conventionnelle, comme un `goto` ou une instruction de contrôle de boucle.

`Explicit blessing to '' (assuming package main)`

(Consécration par `bless` explicite vers `''` (paquetage principal `main` adopté))

(W misc) Vous avez consacré par `bless` une référence à une chaîne de longueur nulle. Cela a pour effet de consacrer la référence dans le paquetage principal `main`. Ce n'est pas habituellement ce que vous voulez. Pensez à fournir un paquetage cible par défaut, comme dans `bless($ref, $p || 'Mon_Paquetage');`

`false [] range "%s" in regexp`

(Faux intervalle `[]` `"%s"` dans l'expression régulière)

(W regexp) Un intervalle de classe de caractères doit débuter et finir par un caractère littéral et non par une autre classe de caractères comme `\d` ou `[:alpha:]`. Le `-` dans votre faux intervalle est interprété comme un `-` littéral. Pensez à protéger le `-` comme cela : `\-`.

`Fatal VMS error at %s, line %d`

(Erreur VMS fatale dans *%s*, ligne *%d*)

(P) Il s'agit d'une erreur spécifique à VMS. Il s'est passé quelque chose de fâcheux dans un service du système VMS ou dans une routine CRTL ; le statut de sortie de

Perl devrait fournir plus de détails. Le nom de fichier dans `at %s` et le numéro de ligne dans `line %d` vous indiquent quelle section du code source Perl a été affectée.

`fcntl is not implemented`

(`fcntl` non implémenté)

(F) Votre machine n'implémente apparemment pas `fcntl`. Qu'est-ce que c'est que ça, un PDP-11 ou quelque chose de similaire ?

`Filehandle %s never opened`

(Handle de fichier `%s` jamais ouvert)

(W unopened) Une opération d'entrée/sortie a été tentée sur un handle de fichier qui n'a jamais été initialisé. Vous devez faire un appel à `open` ou à `socket` ou appeler un constructeur du module `FileHandle`.

`Filehandle %s opened only for input`

(Handle de fichier `%s` ouvert seulement pour les entrées)

(W io) Vous avez tenté d'écrire dans un handle de fichier ouvert en lecture seule. Si vous vouliez que ce soit un handle de fichier ouvert en lecture-écriture, vous deviez l'ouvrir avec +<, +> ou +>> au lieu de < ou rien du tout. Si vous désiriez uniquement écrire dans le fichier, utilisez > ou >>.

`Filehandle %s opened only for output`

(Handle de fichier `%s` ouvert seulement pour les sorties)

(W io) Vous avez tenté de lire dans un handle de fichier ouvert en écriture seule. Si vous vouliez que ce soit un handle de fichier ouvert en lecture-écriture, vous deviez l'ouvrir avec +<, +> ou +>> au lieu de < ou rien du tout. Si vous désiriez uniquement lire dans le fichier, utilisez <.

`Final $ should be \$ or $name`

($ final devrait être \$ ou $nom)

(F) Vous devez maintenant décider si le $ final dans une chaîne devait être interprété comme le signe dollar littéral ou s'il était là pour introduire un nom de variable s'avérant manquer. Vous devez donc ajouter soit l'antislash, soit le nom.

`Final @ should be \@ or @name`

(@ final devrait être \@ ou @nom)

(F) Vous devez maintenant décider si le @ final dans une chaîne devait être interprété comme le signe arobase littéral ou s'il était là pour introduire un nom de variable s'avérant manquer. Vous devez donc ajouter soit l'antislash, soit le nom.

`flock() on close filehandle %s`

(`flock()` sur un handle de fichier fermé `%s`)

(W closed) Le handle de fichier sur lequel vous avez essayé de faire `flock` a été lui-même fermé quelque temps auparavant. Vérifiez le flux logique de votre programme. `flock` agit sur des handles de fichiers. Avez-vous tenté d'appeler `flock` sur un handle de répertoire du même nom ?

`Format %s redefined`

(Format `%s` redéfini)

(W redefine) Vous avez redéfini un format. Pour supprimer cet avertissement, écrivez

```
{
    no warnings;
    eval "format NAME =...";
}
```

`Format not terminated`

(Format inachevé)

(F) Un format doit être terminé par une ligne contenant uniquement un point. Perl est arrivé en fin de fichier sans trouver une telle ligne.

`Found = in conditional, should be ==`

(= trouvé dans une condition, devrait être ==)

(W syntax) Vous avez écrit :

```
if ($truc = 123)
```

alors que vous vouliez dire :

```
if ($truc == 123)
```

(ou quelque chose de similaire).

`gdbm store returned %d, errno %d, key "%s"`

(Le stockage gdbm a renvoyé *%d*, numéro d'erreur *%d*, clef "*%s*")

(S) Un avertissement de l'extension `GDBM_File` indique qu'un stockage a échoué.

`gethostent not implemented`

(`gethostent` non implémenté)

(F) Votre bibliothèque C n'implémente apparemment pas `gethostent`, probablement parce que si elle le faisait, elle se sentirait moralement obligée de renvoyer chaque nom d'hôte existant sur Internet.

`get%sname() on closed socket %s`

(`get%sname()` sur une socket fermée *%s*)

(W closed) Vous avez essayé d'obtenir une socket ou une socket parente d'une socket fermée. Peut-être avez-vous oublié de vérifier la valeur renvoyée par l'appel `socket` ?

`getpwnam returned invalid UIC %#o for user "%s"`

(`getpwname` a renvoyé l'UIC `%#O` invalide pour l'utilisateur "*%s*")

(S) Il s'agit d'un avertissement spécifique à VMS. L'appel à `sys$getuai`, sous-jacent de l'opérateur `getpwnam`, a retourné une UIC invalide.

`glob failed (%s)`

(Échec du `glob`)

(W glob) Quelque chose s'est mal passé avec le(s) programme(s) externe(s) utilisé(s) par `glob` et `<*.c>`. Habituellement, cela signifie que vous avez fourni un motif de `glob` qui a entraîné l'échec du programme externe qui s'est terminé avec un statut différent de zéro. Si le message indique que la sortie anormale a engendré une copie du cœur du programme (*core dump*), ceci peut également vouloir dire que

votre *csh* (C shell) est corrompu. Si c'est le cas, vous devriez changer toutes les variables se rapportant à *csh* dans *config.sh*. Si vous utilisez *tcsh*, changez les variables qui s'y réfèrent comme si c'était *csh*. (ex. `full_csh='/usr/bin/tcsh'`) ; sinon, videz-les toutes (sauf `d_csh` qui devrait valoir `'undef'`) pour que Perl croit qu'il manque *csh*. Dans tous les cas, une fois *config.sh* mis à jour, lancez *./Configure -S* et reconstruisez Perl.

`Glob not terminated`

(Glob non terminé)

(F) L'analyseur lexical a vu un signe inférieur à un endroit où il s'attendait à trouver un terme, il recherche donc le signe supérieur correspondant et ne le trouve pas. Il y a des chances que vous ayez oublié des parenthèses nécessaires auparavant dans la ligne et que vous avez réellement voulu dire le symbole `<`.

`Global symbol "%s" requires explicit package name`

(Le symbole global "*%s*" exige un nom de paquetage explicite)

(F) Vous avez écrit `use strict vars`, ce qui indique que toutes les variables doivent soit avoir une portée lexicale (en utilisant `my`), soit être déclarés au préalable en utilisant `our`, soit être explicitement qualifiées pour dire à quel paquetage la variable globale appartient (en utilisant `::`).

`Got an error from DosAllocMem`

(Erreur rencontrée depuis `DosAllocMem`)

(P) Il s'agit d'une erreur spécifique à OS/2. Le plus probable est que vous employez une version obsolète de Perl, cette erreur ne devrait donc jamais se produire de toute façon.

`goto must have label`

(`goto` doit avoir une étiquette)

(F) Au contraire de `next` et de `last`, vous n'êtes pas autorisés à faire un `goto` vers une destination non spécifiée.

`Had to create %s unexpectedly`

(A dû créer *%s* de manière inattendue)

(S internal) Une routine a demandé un symbole dans une table des symboles qui aurait dû déjà exister mais pour une raison ou pour une autre ce n'était pas le cas elle a été crée en urgence pour éviter une copie du cœur du programme (*core dump*).

`Hash %%s missing the % in argument %d of %s()`

(Il manque le `%` du hachage *%%s* dans l'argument *%d* de *%s*())

(D deprecated) Seules les versions de Perl vraiment anciennes vous permettaient d'omettre le `%` dans les noms de hachage à certains endroits. C'est maintenant largement obsolète.

`Hexadecimal number > 0xffffffff non-portable`

(Nombre hexadécimal > `0xffffffff`, non portable)

(W portable) Le nombre hexadécimal que vous avez spécifié est plus grand que `2**32-1` (4 294 967 295) et n'est donc pas portable d'un système à l'autre.

`Identifier too long`

(Identificateur trop long)

(F) Perl limite la taille des identificateurs (noms des variables, fonctions, etc.) à 250 caractères environ pour les noms simples et un peu plus pour les noms composés (comme `$A::B`). Vous avez dépassé les limites de Perl. Les futures versions de Perl seront susceptibles d'éliminer ces limitations arbitraires.

`Ill-formed CRTL environ value "%s"`

(Valeur d'environnement de CRTL "*%s*" mal formée)

(W internal) Il s'agit d'un avertissement spécifique à VMS. Perl a essayé de lire le tableau d'environnement interne de CRTL et a rencontré un élément sans le séparateur = utilisé pour séparer les clefs des valeurs. L'élément est ignoré.

`Ill-formed message in prime_env_iter: |%s|`

(Message mal formé dans `prime_env_iter` : |*%s*|)

(W internal) Il s'agit d'un avertissement spécifique à VMS. Perl a essayé de lire un nom logique ou une définition de symbole CLI en se préparant à parcourir `%ENV` et il n'a pas vu le séparateur attendu entre les clefs et les valeurs, la ligne a donc été ignorée.

`Illegal character %s (carriage return)`

(Caractère illégal *%s* (retour chariot))

(F) D'ordinaire, Perl traite les retours chariots dans le texte du programme au même titre que n'importe quel autre caractère d'espacement, ce qui signifie que vous ne devriez jamais voir cette erreur lorsque Perl a été construit en utilisant les options standards. Pour une raison ou pour une autre, votre version de Perl semble avoir été construite sans ce support. Parlez-en à votre administrateur Perl.

`Illegal division by zero`

(Division par zéro illégale)

(F) Vous avez essayé de diviser un nombre par 0. Soit quelque chose clochait dans votre logique, soit vous devez placer une condition pour vous préserver de cette entrée dénuée de sens.

`Illegal modulus zero`

(Modulo zéro illégal)

(F) Vous avez essayé de diviser un nombre par 0 pour obtenir le reste. La plupart des nombres ne peuvent faire cela gentiment.

`Illegal binary digit`

(Chiffre binaire illégal)

(F) Vous avez utilisé un chiffre autre que 0 ou 1 dans un nombre binaire.

`Illegal octal digit`

(Chiffre octal illégal)

(F) Vous avez utilisé un 8 ou un 9 dans un nombre octal.

`Illegal binary digit %s ignored`

(Chiffre binaire illégal *%s*, ignoré)

(W digit) Vous avez essayé d'utiliser un chiffre autre que 0 ou 1 dans un nombre binaire. L'interprétation du nombre binaire s'est arrêtée avant le chiffre en cause.

`Illegal octal digit %s ignored`

(Chiffre octal illégal *%s*, ignoré)

(W digit) Vous avez peut-être essayé d'utiliser un 8 ou un 9 dans un nombre octal. L'interprétation du nombre octal s'est arrêtée avant le 8 ou le 9.

`Illegal hexadecimal digit %s ignored`

(Chiffre hexadécimal illégal *%s*, ignoré)

(W digit) Vous avez essayé d'utiliser un caractère autre qui n'est ni entre `0` et `9`, ni entre `A` et `F`, ni entre `a` et `f` dans un nombre hexadécimal. L'interprétation du nombre hexadécimal s'est arrêtée avant le caractère illégal.

`Illegal number of bits in vec`

(Nombre de bits illégal dans `vec`)

(F) Le nombre de bits dans `vec` (le troisième argument) doit être une puissance de deux de 1 à 32 (ou 64 si votre plate-forme le supporte).

`Illegal switch in PERL5OPT: %s`

(Option illégale dans `PERL5OPT` : *%s*)

(X) La variable d'environnement `PERL5OPT` ne peut être utilisée que pour positionner les options suivantes : **`-[DIMUdmw]`**.

`In string, @%s now must be written as \@%s`

(Dans une chaîne, `@%s` doit maintenant s'écrire `\@%s`)

(F) Perl essayait auparavant de deviner si vous vouliez un tableau interpolé ou un `@` littéral. Il faisait ceci lorsque la chaîne était utilisée la première fois pendant l'exécution. Les chaînes sont maintenant analysées à la compilation et les instances ambiguës de `@` doivent être éclaircies, soit en préfixant par un antislash pour indiquer qu'il s'agit d'un littéral, soit en déclarant (ou en utilisant) le tableau dans le programme avant la chaîne (lexicalement). (Un jour un `@` sans antislash sera simplement interprété comme une interpolation d'un tableau.)

`Insecure dependency in %s`

(Dépendance non sécurisée dans *%s*)

(F) Vous avez essayé de faire quelque chose que le mécanisme de marquage des variables n'a pas apprécié. Ce mécanisme de marquage est activé quand vous exécutez un script setuid ou setgid, ou lorsque vous spécifiez **`-T`** pour l'activer explicitement. Le mécanisme de marquage étiquette toutes les données dérivées directement ou indirectement de l'utilisateur, qui est considéré indigne de votre confiance. Si l'une de ces données est utilisée dans une opération « dangereuse », vous obtenez cette erreur.

`Insecure directory in %s`

(Répertoire non sécurisé dans *%s*)

(F) Vous ne pouvez utilisez `system`, `exec` ou un pipe ouvert vers un script setuid ou setgid si `$ENV{PATH}` contient un répertoire qui a les droits en écriture pour tout le monde.

`Insecure $ENV{%s} while running %s`

(`$ENV{%s}` non sécurisé alors que *%s* est en train de tourner)

(F) Vous ne pouvez utilisez `system`, `exec` ou un pipe ouvert vers un script setuid ou setgid si `$ENV{PATH}`, `$ENV{IFS}`, `$ENV{CDPATH}`, `$ENV{ENV}` ou `$ENV{BASH_ENV}` sont dérivés de données fournies (ou potentiellement fournies) par l'utilisateur. Le script doit positionner le chemin à une valeur connue, en utilisant une donnée digne de confiance.

`Integer overflow in %s number`

(Dépassement d'entier dans le nombre *%s*)

(W overflow) Le nombre hexadécimal, octal ou binaire que vous avez spécifié soit comme littéral, soit comme argument de `hex` ou `oct` est trop grand pour votre architecture. Sur les machines 32-bits ,les plus grands nombres hexadécimaux, octaux ou binaires représentables sans dépassement sont 0xFFFFFFFF, 037777777777 et 0b11111111111111111111111111111111, respectivement. Remarquez que Perl élève de manière transparente tous les nombres vers une représentation interne en virgule flottante — sujette à des erreurs de perte de précision dans les opération ultérieures.

`Internal inconsistency in tracking vforks`

(Incohérence interne dans la recherche des `vfork`s)

(S) Il s'agit d'un avertissement spécifique à VMS. Perl garde une trace du nombre de fois où vous avez appelé `fork` et `exec`, pour déterminer si l'appel courant à `exec` devrait affecter le script en cours ou un sous-processus (Voir « exec LIST » dans *perlvms*(1)). De toute façon, ce compte est devenu perturbé, Perl fait donc une supposition et traite ce `exec` comme une requête pour terminer le script Perl et exécuter la commande spécifiée.

`internal disaster in regexp`

(désastre interne dans une expression régulière)

(P) Quelque chose s'est très mal passé dans l'analyseur d'expressions régulières.

`internal urp in regexp at /%s/`

(gloups interne dans une expression régulière)

(P) Quelque chose a très mal tourné dans l'analyseur d'expressions régulières.

`Invalid %s attribute: %s`

(Attribut *%s* invalide : *%s*)

(F) L'attribut indiqué pour un sous-programme ou une variable n'a pas été reconnu par Perl ou par un gestionnaire fourni par l'utilisateur.

`Invalid %s attributes: %s`

(Attributs *%s* invalides : *%s*)

(F) Les attributs indiqués pour un sous-programme ou une variable n'ont pas été reconnus par Perl ou par un gestionnaire fourni par l'utilisateur.

`invalid [] range "%s" in regexp`

(Intervalle [] "%s" invalide dans l'expression régulière)

(F) L'intervalle spécifié dans une classe de caractères a un caractère de début supérieur au caractère de fin.

`Invalid conversion in %s: "%s"`

(Conversion invalide dans %s : "%s")

(W printf) Perl ne comprend pas le format de conversion donné.

`Invalid separator character %s in attribute list`

(Caractère de séparation %s invalide dans la liste d'attributs)

(F) Quelque chose d'autre qu'un deux-points ou qu'un espacement a été détecté entre les éléments d'une liste d'attributs. Si l'attribut précédent avait une liste de paramètres entre parenthèses, peut-être que cette liste s'est terminée trop tôt.

`Invalid type in pack: '%s'`

(Type invalide dans pack : '%s')

(F) Le caractère donné n'est pas un type d'empaquetage valide.

(W pack) Le caractère donné n'est pas un type d'empaquetage valide mais l'habitude était de l'ignorer silencieusement.

`Invalid type in unpack: '%s'`

(Type invalide dans unpack : '%s')

(F) Le caractère donné n'est pas un type de dépaquetage valide.

(W unpack) Le caractère donné n'est pas un type de dépaquetage valide mais l'habitude était de l'ignorer silencieusement.

`ioctl is not implemented`

(F) Votre machine n'implémente apparemment pas `ioctl`, ce qui est quelque peu surprenant de la part d'une machine qui supporte le C.

`junk on end of regexp`

(Déchets à la fin de l'expression régulière)

(P) L'analyseur d'expressions régulières s'est embrouillé.

`Label not found for "last %s"`

(Étiquette introuvable pour "last %s")

(F) Vous avez nommé une boucle pour en sortir mais vous n'êtes pas actuellement dans une boucle de ce nom, même si vous comptez d'où vous avez été appelé.

`Label not found for "next %s"`

(Étiquette introuvable pour "next %s")

(F) Vous avez nommé une boucle pour effectuer l'itération suivante mais vous n'êtes pas actuellement dans une boucle de ce nom, même si vous comptez d'où vous avez été appelé.

`Label not found for "redo %s"`

(Étiquette introuvable pour "redo %s")

(F) Vous avez nommé une boucle pour recommencer l'itération en cours mais vous n'êtes pas actuellement dans une boucle de ce nom, même si vous comptez d'où vous avez été appelé.

`leaving effective %s failed`

(la sortie de `%s` effectif a échoué)

(F) Alors que le pragma `use filetest` était actif, l'échange entre les UID ou les GID réels et effectifs a échoué.

`listen() on closed socket %s`

(`listen()` sur une socket fermée `%s`)

(W closed) Vous avez essayé de faire un `listen` sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur renvoyée par votre appel à `socket` ?

`Lvalue subs returning %s not implemented yet`

(Sous-programmes en tant que lvalue retournant `%s` pas encore implémentés)

(F) À cause de limitations dans l'implémentation actuelle, les valeurs de tableaux et de hachages ne peuvent pas être renvoyées dans un contexte de lvalue.

`Malformed PERLLIB_PREFIX`

(`PERLLIB_PREFIX` malformée)

(F) Il s'agit d'une erreur spécifique à OS/2. `PERLLIB_PREFIX` doit être de la forme :

```
prefixe1 ;prefixe2
```

ou :

```
prefixe1  prefixe2
```

avec *`prefixe1`* et *`prefixe2`* non vides. Si *`prefixe1`* est en fait un préfixe d'un chemin de recherche de bibliothèque interne, on lui substitue *`prefixe2`*. L'erreur peut survenir si les composants ne sont pas trouvés ou s'ils sont trop longs. Voir `PERLLIB_PREFIX` dans le fichier *README.os2* inclus dans la distribution de Perl.

`Method for operation %s not found in package %s during blessing`

(La méthode pour l'opération `%s` n'a pas été trouvée dans le paquetage `%s` durant la consécration avec `bless`)

(F) Une tentative a été effectuée pour spécifier une entrée dans une table de surcharge qui ne peut être résolue par un sous-programme valide.

`Method %s not permitted`

(Méthode `%s` interdite)

Voir `Server error`.

`Might be a runaway multi-line %s string starting on line %d`

(`%s` pourrait être une chaîne galopant sur plusieurs lignes, débutant à la ligne `%d`)

(S) Une suggestion indiquant que l'erreur précédente peut avoir été causée par un séparateur manquant dans une chaîne ou un motif car la chaîne se finissait éventuellement plus tôt sur la ligne courante.

`Misplaced _ in number`

(_ mal placé dans un nombre)

(W syntax) Un caractère souligné dans une constante décimale n'était pas sur une limite de 3 chiffres.

`Missing $ on loop variable`

($ manquant dans un variable de boucle)

(F) Apparemment, vous avez programmé en *csh* un peu trop longtemps. Les variables en Perl sont toujours précédées du `$`, au contraire des shells, ou cela peut varier d'une ligne à l'autre.

`Missing %sbrace%s on \N{}`

(*%s*accolades*%s* manquantes sur `\N{}`)

(F) Vous avez employé la mauvaise syntaxe de nom de caractère littéral `\N{nom_caractere}` à l'intérieur d'un contexte de guillemets.

`Missing comma after first argument to %s function`

(Virgule manquante après le premier argument de la fonction *%s*)

(F) Alors que certaines fonctions vous permettent de spécifier un handle de fichier ou un « objet indirect » avant la liste d'arguments, celle-ci n'en fait pas partie.

`Missing command in piped open`

(Commande manquante dans un `open` avec un pipe)

(W pipe) Vous avez utilisé la construction `open(HF, "| commande")` ou `open(HF, "commande |")` mais il manquait la commande ou alors elle était vide.

`Missing name in "my sub"`

(Nom manquant dans `"my sub"`)

(F) La syntaxe réservée pour les sous-programmes de portée lexicale exige qu'ils aient un nom par lequel on puisse les trouver.

`(Missing operator before %s?)`

(Opérateur manquant avant *%s* ?)

(S) Il s'agit d'une supposition donnée en complément du message « *%s* `found where operator expected` ». Souvent l'opérateur manquant est une virgule.

`Missing right curly or square bracket`

(Accolade fermante ou crochet fermant manquant)

(F) L'analyseur lexical a compté plus d'accolades ouvrantes ou de crochets ouvrants que d'accolades fermantes ou de crochets fermants. En règle générale, vous trouverez le signe manquant à côté de l'endroit où vous venez d'éditer.

`Modification of a read-only value attempted`

(Tentative de modification d'une valeur en lecture seule)

(F) Vous avez essayé, directement ou indirectement, de changer la valeur d'une constante. Vous ne pouvez pas, bien entendu, essayer de faire `2 = 1` car le compilateur intercepte ceci. Mais une autre façon simple de faire la même chose est :

```
sub mod { $_[0] = 1 } mod(2);
```

Un autre moyen serait d'affecter à `substr` ce qui se trouve au-delà de la fin de la chaîne.

`Modification of non-creatable array value attempted, subscript %d`

(Tentative de modification d'une valeur de impossible à créer, indice *%d*)

(F) Vous avez essayé de faire naître une valeur de tableau et l'indice était probablement négatif, même en comptant à rebours depuis de la fin du tableau.

`Modification of non-creatable hash value attempted, subscript "%s"`

(Tentative de modification d'une valeur de hachage impossible à créer, indice "*%s*")

(P) Vous avez essayé de faire naître une valeur de hachage et elle n'a pu être créée pour une raison particulière.

`Module name must be constant`

(Le nom de module doit être constant)

(F) Seul un nom de module brut est permis comme premier argument d'un `use`.

`msg%s not implemented`

(msg*%s* non implémenté)

(F) Vous n'avez pas d'implémentation de messages des IPC System V sur votre système.

`Multidimensional syntax %s not supported`

(Syntaxe multidimensionnelle *%s* non supportée)

(W syntax) Les tableaux multidimensionnels ne s'écrivent pas `$truc[1,2,3]`. Ils s'écrivent `$truc[1][2][3]`, tout comme en C.

`Name "%s::%s" used only once: possible typo`

(Nom *%s*`::`*%s*`"` utilisé une seule fois : faute de frappe possible)

(W once) Les fautes de frappe apparaissent fréquemment sous la forme de noms de variable uniques. Si vous avez une bonne raison pour cela, vous n'avez qu'à mentionner la variable à nouveau pour supprimer ce message. La déclaration `our` est destinée à cela.

`Negative length`

(Longueur négative)

(F) Vous avez essayé de faire une opération `read`/`write`/`send`/`recv` avec un tampon de longueur plus petite que 0. C'est difficile à imaginer.

`nested *?+ in regexp`

(`*?+` imbriqué dans l'expression régulière)

(F) Vous ne pouvez quantifier un quantifiant sans faire intervenir de parenthèses. Donc les choses comme `**`, `+*` ou `?*` sont illégales.

Remarquez cependant que les quantifiants de recherche de correspondance minimum, `*?`, `+?`, et `??` ressemblent à des des quantifiants imbriqués mais ne le sont pas.

`No #! line`

(Pas de ligne `#!`)

(F) L'émulateur setuid exige que le script possède une ligne valide de la forme `#!` même sur les machines ne supportant pas la construction `#!`.

`No %s allowed while running setuid`

(Pas de `%s` autorisé alors que l'on tourne avec setuid)

(F) Pour des questions de sécurité, certaines opérations sont trop dangereuses pour être exécutées, voire tentées, dans un script setuid ou setgid. Généralement, il y existera bien un autre moyen pour faire ce que vous voulez, c'est-à-dire que si ce moyen n'est pas sécurisé, il est au moins sécurisable.

`No -e allowed in setuid scripts`

(pas de `-e` autorisé dans les scripts setuid)

(F) Un script setuid ne peut être spécifié par l'utilisateur.

`No %s specified for -%c`

(Pas de `%s` spécifié pour `-%c`)

(F) L'option de la ligne de commande indiquée nécessite un argument obligatoire mais vous n'en avez pas spécifié.

`No comma allowed after %s`

(Pas de virgule autorisée après `%s`)

(F) Lorsqu'un opérateur de liste a un handle de fichier ou un « object indirect », il ne doit pas y avoir une virgule ce premier argument et les arguments suivants. Sinon, il serait juste interprété comme tout autre un argument.

Une situation ténébreuse dans laquelle ce message se produit est lorsque vous vous attendez à importer une constante dans votre espace de noms avec `use` ou `import` mais qu'une telle importation ne s'est pas produite. Vous auriez dû utiliser une liste d'importation explicite des constantes que vous vous attendiez à voir. Une liste d'importation explicite aurait probablement intercepté cette erreur plus tôt. Ou peut-être s'agit-il seulement d'une erreur de frappe dans le nom de la constante.

`No command into which to pipe on command line`

(Pas de commande dans laquelle envoyer un pipe sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirections dans la ligne de commande et a trouvé un | à la fin de la ligne de commande, il ne sait donc pas vers où vous voulez rediriger avec un pipe la sortie de cette commande.

`No DB::DB routine defined`

(Pas de routine `DB::DB` de définie)

(F) Le code en train d'être exécuté a été compilé avec l'option **`-d`** mais pour une raison ou pour une autre le fichier *perl5db.pl* (ou alors un fac-similé) n'a pas défini de routine à appeler au début de chaque instruction. Ce qui est étrange car le fichier aurait dû être chargé automatiquement et n'a pas pu être analysé correctement.

`No dbm on this machine`

(Pas de `dbm` sur cette machine)

(P) Ceci est compté comme une erreur interne ; chaque machine doit fournir dbm de nos jours car Perl est livré avec SDBM.

`No DBsub routine`

(Pas de routine `DBsub`)

(F) Le code en train de s'exécuter a été compilé avec l'option **`-d`** mais pour une raison ou pour une autre le fichier *perl5db.pl* (ou alors un fac-similé) n'a pas défini de routine `DB::sub` à appeler au début de chaque appel de sous-programme ordinaire.

`No error file after 2> or 2>> on command line`

(Pas de fichier d'erreur après `2>` ou `2>>` sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirections dans la ligne de commande et y a trouvé un `2>` ou un `2>>` mais sans trouver le nom du fichier dans lequel écrire les données destinées à `STDERR`.

`No input file after < on command line`

(Pas de fichier d'entrée après `<` sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirections dans la ligne de commande et y a trouvé un `<` mais sans trouver le nom du fichier depuis lequel lire les données de `STDIN`.

`No output file after > on command line`

(Pas de fichier de sortie après `2>` ou `2>>` sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirections dans la ligne de commande et a trouvé un `>` isolé à la fin de la ligne de commande mais sans savoir où vous vouliez redirigez `STDOUT`.

`No output file after > or >> on command line`

(Pas de fichier de sortie après `>` ou `>>` sur la ligne de commande)

(F) Il s'agit d'une erreur spécifique à VMS. Perl effectue ses propres redirection dans la ligne de commande et y a trouvé un `>` ou un `>>` mais sans trouver le nom du fichier dans lequel écrire les données destinées à `STDOUT`.

`No package name allowed for variable %s in "our"`

(Pas de nom de paquetage autorisé pour la variable `%s` dans `"our"`)

(F) Les noms de variable pleinement qualifiés ne sont pas autorisés dans les déclarations `our` car cela n'aurait pas beaucoup de sens avec les sémantiques existantes. Une telle syntaxe est réservée pour de futures extensions.

`No Perl script found in input`

(Pas de script Perl trouvé en entrée)

(F) Vous avez appelé *perl -x* mais aucune ligne n'est trouvée dans le fichier commençant par `#!` et contenant le mot « `perl` ».

`No setregid available`

(Pas de `setregid` de disponible)

(F) *Configure* n'a rien trouvé qui ressemble à l'appel `setregid` pour votre système.

`No setreuid available`

(Pas de `setreuid` de disponible)

(F) *Configure* n'a rien trouvé qui ressemble à l'appel `setreuid` pour votre système.

`No space allowed after -%c`

(Pas d'espace autorisé après *-%c*)

(F) L'argument de l'option de la ligne de commande indiquée doit suivre immédiatement l'option, sans qu'aucun espace n'intervienne.

`No such pseudohash field "%s"`

(Aucun champ de pseudo-hachage "*%s*")

(F) Vous avez essayé d'accéder à un tableau en tant que hachage mais le nom de champ utilisé n'est pas défini. Le hachage à l'indice 0 doit faire correspondre tous les noms de champs valides aux indices du tableau pour que cela fonctionne.

`No such pseudohash field "%s" in variable %s of type %s`

(Aucun champ de pseudo-hachage "*%s*" dans la variable *%s* de type *%s*)

(F) Vous avez essayé d'accéder à un champ d'une variable typée mais le type ignore le nom du champ. Les noms de champ sont recherchés dans le hachage `%FIELDS` dans le paquetage du type à la compilation. Le hachage `%FIELDS` est généralement initialisé avec le pragma `fields`.

`No such pipe open`

(Aucun pipe ouvert)

(P) Il s'agit d'une erreur spécifique à VMS. La routine interne `my_pclose` a essayé de fermer un pipe qui n'avait pas été ouvert. Ceci aurait dû être capturé plus tôt comme une tentative de fermeture d'un handle de fichier non ouvert.

`No such signal: SIG%s`

(Aucun signal: `SIG%s`)

(W signal) Le nom de signal que vous avez spécifié comme indice de `%SIG` n'a pas été reconnu. Faites *kill -l* dans votre shell pour voir les noms de signaux valides sur votre système.

`No UTC offset information; assuming local time is UTC`

(Pas d'information sur le décalage UTC ; adopte UTC comme heure locale)

(S) Il s'agit d'un avertissement spécifique à VMS. Perl a été incapable de trouver le décalage horaire pour l'heure locale, il a donc supposé que l'heure locale du système et UTC étaient équivalentes. SI ce n'est pas le cas, définissez le nom logique `SYS$TIMEZONE_DIFFERENTIAL` pour qu'il traduise le nombre de secondes à ajouter à UTC afin d'obtenir l'heure locale.

`Not a CODE reference`

(Non une référence de CODE)

(F) Perl a essayé d'évaluer une référence vers une valeur de code (c'est-à-dire un sous-programme) mais a trouvé une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref` pour trouver de quelle sorte de référence il s'agissait exactement.

`Not a format reference`

(Non une référence de format)

(F) Nous ne sommes pas sûrs de la manière dont vous vous arrangez pour générer

une référence vers un format anonyme, mais ce message indique que vous l'avez fait et que cela n'existe pas.

`Not a GLOB reference`

(Non une référence de GLOB)

(F) Perl a essayé d'évaluer une référence vers un « typeglob » (c'est-à-dire une entrée dans la table des symboles ressemblant à `*truc`) mais a trouvé une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref` pour trouver de quelle sorte de référence il s'agissait exactement.

`Not a HASH reference`

(Non une référence de HACHAGE)

(F) Perl a essayé d'évaluer une référence vers une valeur de hachage mais a trouvé une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref` pour trouver de quelle sorte de référence il s'agissait exactement.

`Not a perl script`

(Non un script perl)

(F) L'émulateur setuid exige que le script possède une ligne valide de la forme `#!` même sur les machines ne supportant pas la construction `#!`. La ligne doit mentionner « `perl` ».

`Not a SCALAR reference`

(Non une référence de SCALAIRE)

(F) Perl a essayé d'évaluer une référence vers une valeur scalaire mais a trouvé une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref` pour trouver de quelle sorte de référence il s'agissait exactement.

`Not a subroutine reference`

(Non une référence de sous-programme)

(F) Perl a essayé d'évaluer une référence vers une valeur de code (c'est-à-dire un sous-programme) mais a trouvé une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref` pour trouver de quelle sorte de référence il s'agissait exactement.

`Not a subroutine reference in overload table`

(Non une référence de sous-programme dans la table de surcharge)

(F) Une tentative a été faite pour spécifier une entrée dans une table de surcharge qui ne pointe pas d'une façon ou d'une autre vers un sous-programme valide.

`Not an ARRAY reference`

(Non une référence de TABLEAU)

(F) Perl a essayé d'évaluer une référence vers une valeur de tableau mais a trouvé une référence vers quelque chose d'autre à la place. Vous pouvez utiliser la fonction `ref` pour trouver de quelle sorte de référence il s'agissait exactement.

`Not enough arguments for %s`

(Pas assez d'arguments pour `%s`)

(F) La fonction exige plus d'arguments que vous n'en avez spécifiés.

`Not enough format arguments`

(Pas assez d'arguments de format)

(W syntax) Un format spécifiait plus de champs d'images que la ligne suivante ne lui en a fournis.

`Null filename used`

(Nom de fichier vide utilisé)

(F) Vous ne pouvez exiger un nom de fichier vide, notamment car sur beaucoup de machines ceci signifie le répertoire courant !

`Null picture in formline`

(Image vide dans `formline`)

(F) Le premier argument de `formline` doit être une spécification de format d'image valide. On a trouvé que cet argument était vide, ce qui signifie probablement que vous lui avez fourni une valeur non initialisée.

`NULL OP IN RUN`

(OP NUL DANS RUN)

(P debugging) Une routine interne a appelé `run` avec un pointer de code d'opération (*opcode*) nul.

`Null realloc`

(réallocation nulle)

(P) Une tentative a eu lieu pour réallouer le pointeur `NULL`.

`NULL regexp argument`

(argument NUL dans l'expression régulière)

(P) Les routines internes de recherche de motifs ont tout explosé.

`NULL regexp parameter`

(paramètre NUL dans l'expression régulière)

(P) Les routines internes de recherche de motifs ont perdu la tête.

`Number too long`

(Nombre trop long)

(F) Perl limite la représentation des nombres décimaux dans les programmes à environ 250 caractères. Vous avez dépassé cette longueur. Les futures versions de Perl sont susceptibles d'éliminer cette limitation arbitraire. En attendant, essayez d'utiliser la notation scientifique (ex. `1e6` au lieu de `1_000_000`).

`Octal number > 037777777777 non portable`

(Nombre octal >037777777777, non portable)

(W portable) Le nombre octal que vous avez spécifié est plus grand que `2**32-1` (4 294 967 295) et n'est donc pas portable d'un système à l'autre.

`Octal number in vector unsupported`

(Nombre octal dans un vecteur non supporté)

(F) Les nombres commençant par un `0` ne sont actuellement pas autorisés dans les

vecteurs. L'interprétation en octal de tels nombres sera peut-être supportée dans une future version.

`Odd number of elements in hash assignment`

(Nombre impair d'élément dans une affectation de hachage)

(W misc) Vous avez spécifié un nombre impair d'éléments pour initialiser un hachage, ce qui est un impair à la règle car un hachage va avec des paires clé/valeur.

`Offset outside string`

(Décalage en dehors de la chaîne)

(F) Vous avez essayé de faire une opération `read`/`write`/`send`/`recv` avec un décalage pointant en dehors du tampon. Ceci est difficile à imaginer. La seule exception à cette règle est que faire un `sysread` après un tampon étendra ce tampon et remplira de zéros la nouvelle zone.

`oops: oopsAV`

(gloups : gloupsAV)

(S internal) Il s'agit d'un avertissement interne comme quoi la grammaire est abîmée.

`oops: oopsHV`

(gloups gloupsHV)

(S internal) Il s'agit d'un avertissement interne indiquant que la grammaire est abîmée.

`Operation "%s": no method found, %s`

(Opération "*%s*" : aucune méthode trouvée, *%s*)

(F) Une tentative a été faite pour effectuer une opération de surcharge pour laquelle aucun gestionnaire n'a été défini. Alors que certains gestionnaires peuvent être autogénérés en termes d'autres gestionnaires, il n'existe pas de gestionnaire par défaut pour quelque opération que ce soit, à moins que la clé de surcharge `fallback` soit spécifiée comme étant vraie.

`Operator or semicolon missing before %s`

(Opérateur ou point-virgule manquant avant *%s*)

(S ambiguous) Vous avez utilisé une variable ou un appel de sous-programme là où l'analyseur syntaxique s'attendait à trouver un opérateur. L'analyseur a considéré que vous pensiez vraiment utiliser un opérateur mais cela est fortement susceptible de s'avérer incorrect. Par exemple, si vous écrivez accidentellement `*truc *truc`, cela sera interprété comme si vous aviez écrit `*truc * 'truc'`.

`Out of memory!`

(Dépassement de mémoire !)

(X) La fonction interne de Perl `malloc` a renvoyé 0, ce qui indique que la mémoire restante (ou la mémoire virtuelle) n'a pas suffit à satisfaire la requête. Perl n'a pas d'autre option que de sortir immédiatement.

`Out of memory for yacc stack`

(Dépassement de mémoire pour la pile de yacc)

(F) L'analyseur syntaxique *yacc* a voulu agrandir sa pile pour qu'il puisse continuer son analyse mais `realloc` n'a pas voulu lui donner plus de mémoire, virtuelle ou autre.

`Out of memory during request for %s`

(Dépassement de mémoire pendant une requête pour `%s`)

(X|F) La fonction `malloc` a renvoyé 0, ce qui indique que la mémoire restante (ou la mémoire virtuelle) n'a pas suffit à satisfaire la requête.

La requête a été jugée faible, la possibilité de la capturer est donc dépendante de la façon dont Perl a été compilé. Par défaut, elle ne peut être capturée. Cependant, s'il a été compilé à cet effet, Perl peut utiliser le contenu de `$^M` comme une zone d'urgence après être sorti par `die` avec ce message. Dans ce cas l'erreur peut-être capturée *une seule fois*.

`Out of memory during "large" request for %s`

(Dépassement de mémoire pendant une requête "large" pour `%s`)

(F) La fonction `malloc` a renvoyé 0, ce qui indique que la mémoire restante (ou la mémoire virtuelle) n'a pas suffit à satisfaire la requête. Cependant, la requête a été jugée assez large (par défaut 64K à la compilation), la possibilité de sortir en capturant cette erreur est donc permise.

`Out of memory during ridiculously large request`

(Dépassement de mémoire pendant un requête ridiculement large)

(F) Vous ne pouvez allouer plus de `2**31+`« epsilon » octets. Cette erreur est généralement causée par une faute de frappe dans le programme Perl (ex. `$tab[time]` au lieu de `$tab[$time]`).

`page overflow`

(dépassement de page mémoire)

(W io) Un seul appel à `write` a produit plus de lignes que ne peut en contenir une page.

`panic: ck_grep`

(panique : `ck_grep`)

(P) Le programme a échoué sur un test de cohérence interne en essayant de compiler un `grep`.

`panic: ck_split`

(panique : `ck_split`)

(P) Le programme a échoué sur un test de cohérence interne en essayant de compiler un `split`.

`panic: corrupt saved stack index`

(panique : index de la pile de sauvegarde corrompu)

(P) On a demandé à la pile de sauvegarde de restaurer plus de valeurs localisées qu'il n'en existe.

`panic: del_backref`

(panique : référence arrière supprimée)

(P) Le programme a échoué sur un test de cohérence interne en essayant de réinitialiser une référence lâche (*weak reference*).

`panic: die %s`

(panique : `die %s`)

(P) Nous avons passé l'élément du dessus de la pile de contexte à un contexte d'`eval` pour découvrir alors que l'on n'était pas dans un contexte d'`eval`.

`panic: do_match`

(panique : `do_match`)

(P) La routine interne `pp_match` a été appelée avec des données opérationnelles invalides.

`panic: do_split`

(panique : `do_split`)

(P) Quelque chose de terrible s'est mal passé lors de la préparation pour un `split`.

`panic: do_subst`

(panique : `do_subst`)

(P) La routine interne `pp_subst` a été appelée avec des données opérationnelles invalides.

`panic: do_trans`

(panique : `do_trans`)

(P) La routine interne `do_trans` a été appelée avec des données opérationnelles invalides.

`panic: frexp`

(panique : `frexp`)

(P) La fonction de la bibliothèque `frexp` a échoué, rendant le `printf("%f")` impossible.

`panic: goto`

(panique : `goto`)

(P) Nous avons passé l'élément du dessus de la pile à un contexte avec l'étiquette spécifiée et découvert ensuite qu'il ne s'agissait pas d'un contexte que nous savions joindre par un `goto`.

`panic: INTERPCASEMOD`

(panique : `INTERPCASEMOD`)

(P) L'analyseur lexical s'est retrouvé dans un mauvais état à un modificateur de casse (majuscule/minuscule).

`panic: INTERPCONCAT`

(panique : `INTERPCONCAT`)

(P) L'analyseur lexical s'est retrouvé dans un mauvais état en analysant une chaîne de caractères avec des parenthèses.

`panic: kid popen errno read`

(panique : lecture du numéro d'erreur dans le `popen` d'un fils)

(F) Un fils issu d'un `fork` a renvoyé un message incompréhensible à propose de son numéro d'erreur.

`panic: last`

(panique : `last`)

(P) Nous avons passé l'élément du dessus de la pile à un contexte de bloc et découvert par la suite que ce n'était pas un contexte de bloc.

`panic: leave_scope clearsv`

(panique : `clearsv` dans la sortie de la portée)

(P) Une variable lexicale modifiable est maintenant en lecture seule dans le bloc pour une raison ou pour une autre.

`panic: leave_scope inconsistency`

(panique : sortie de la portée incohérente)

(P) La pile de sauvegarde n'est sûrement plus synchronisée. Du moins, il y avait un type `enum` invalide à son sommet.

`panic: malloc`

(panique : `malloc`)

(P) Quelque chose a demandé un nombre négatif d'octets à `malloc`.

`panic: magic_killbackrefs`

(panique : suppression magique de toutes les références arrières)

(P) Le programme a échoué sur un test de cohérence interne en essayant de réinitialiser toutes les références lâches vers un objet.

`panic: mapstart`

(panique : `mapstart`)

(P) Le compilateur s'est embrouillé avec la fonction `map`.

`panic: null array`

(panique : tableau nul)

(P) On a passé à l'une des routines interne pour les tableaux un pointeur d'AV nul.

`panic: pad_alloc`

(panique : `pad_alloc`)

(P) Le compilateur ne sait plus quelle mémoire de travail (*scratchpad*) il allouait et libérait les valeurs temporaires et lexicales.

`panic: pad_free curpad`

(panique : `pad_free curpad`)

(P) Le compilateur ne sait plus quelle mémoire de travail (*scratchpad*) il allouait et libérait les valeurs temporaires et lexicales.

`panic: pad_free po`

(panique : `pad_free po`)

(P) Un décalage incorrect dans la mémoire de travail (*scratchpad*) a été détecté en interne.

`panic: pad_reset curpad`

(panique : `pad_reset curpad`)

(P) Le compilateur ne sait plus quelle mémoire de travail (*scratchpad*) il allouait et libérait les valeurs temporaires et lexicales.

`panic: pad_sv po`

(panique : `pad_sv po`)

(P) Un décalage incorrect dans la mémoire de travail (*scratchpad*) a été détecté en interne.

`panic: pad_swipe curpad`

(panique : `pad_swipe curpad`)

(P) Le compilateur ne sait plus quelle mémoire de travail (*scratchpad*) il allouait et libérait les valeurs temporaires et lexicales.

`panic: pad_swipe po`

(panique : `pad_swipe po`)

(P) Un décalage incorrect dans la mémoire de travail (*scratchpad*) a été détecté en interne.

`panic: pp_iter`

(panique : `pp_iter`)

(P) L'itérateur de `foreach` a été appelé dans un contexte qui n'était pas un contexte de boucle.

`panic: realloc`

(panique : `realloc`)

(P) Quelque chose a demandé un nombre négatif d'octets à `realloc`.

`panic: restartop`

(panique `restartop`)

(P) Une routine interne a demandé un `goto` (ou quelque chose de semblable) mais n'a pas fourni de destination.

`panic: return`

(panique : `return`)

(P) Nous avons passé l'élément du dessus de la pile de contexte à un sous-programme ou à un contexte d'`eval` pour découvrir alors que l'on n'était ni dans un sous-programme, ni dans un contexte d'`eval`.

`panic: scan_num`

(panique : `scan_num`)

(P) La fonction interne de Perl `scan_num` a été appelée sur quelque chose qui n'était pas un nombre.

`panic: sv_insert`

(panique : `sv_insert`)

(P) On a demandé à la routine `sv_insert` d'enlever plus de chaînes qu'il en existe.

`panic: top_env`

(panique : `top_env`)

(P) Le compilateur a tenté de faire un `goto` ou quelque chose d'aussi bizarre.

`panic: yylex`

(panique : `yylex`)

(P) L'analyseur lexical s'est retrouvé dans un mauvais état en traitant un modificateur de casse (majuscule/minuscule).

`panic: %s`

(panique : `%s`)

(P) Une erreur interne.

`Parentheses missing around "%s" list`

(Parenthèses manquantes autour de la liste "`%s`")

(W parenthesis) Vous avez écrit quelque chose comme :

```
my $truc, $machin = @_;
```

alors que vous vouliez dire :

```
my ($truc, $machin) = @_;
```

Souvenez-vous que `my`, `our` et `local` regroupent plus fortement que la virgule.

`Perl %3.3f required--this is only version %s, stopped`

(Perl `%3.3f` requis — il ne s'agit que de la version `%s`, arrêt)

(F) Le module en question utilise des fonctionnalités offertes par une version plus récente que celle exécutée actuellement. Depuis combien de temps n'avez-vous pas fait de remise à niveau ?

`PERL_SH_DIR too long`

(`PERL_SH_DIR` trop long)

(F) Il s'agit d'une erreur spécifique à OS/2. `PERL_SH_DIR` est le répertoire où se trouve le shell *sh*. Voir `PERL_SH_DIR` dans le fichier *README.os2* compris dans la distribution de Perl.

`Permission denied`

(Permission refusée)

(F) L'émulateur setuid de *suidperl* a décidé que vous n'étiez pas quelqu'un de bon.

`pid %x not a child`

(le pid `%x` n'est pas un fils)

(W exec) Il s'agit d'une alerte spécifique à VMS ; on a demandé à `waitpid` d'attendre un processus qui n'est pas un sous-processus du processus courant. Même si cela est correct dans la perspective de VMS, cela n'est probablement pas ce que vous vouliez.

`POSIX getpgrp can't take an argument`

(`getpgrp` POSIX ne prend pas d'argument)

(F) Votre système dispose de la fonction POSIX `getpgrp`, qui ne prend pas d'argument, au contraire de la version BSD, qui prend un PID.

`Possible Y2K bug: %s`

(Bogue possible de l'an 2000)

(W y2k) Vous être en train de concaténer le nombre 19 avec un autre nombre, ce qui pourrait être un problème potentiel pour l'an 2000.

`Possible attempt to put comments in qw() list`

(Tentative possible de mettre des commentaires dans une liste `qw()`)

(W qw) Les listes `qw` contiennent des éléments séparés par des espacements ; comme avec les chaînes littérales, les caractères de commentaires ne sont pas ignorés, mais au lieu de ça traités comme des données littérales. (Vous auriez pu utiliser des délimiteurs différents des parenthèses indiquées ici ; les accolades sont également fréquemment utilisées.)

Vous avez probablement écrit quelque chose comme ceci :

```
@liste = qw(
    a    # un commentaire
    b    # un autre commentaire
);
```

alors que vous auriez dû écrire cela :

```
@liste = qw(
    a
    b
);
```

Si vous voulez vraiment des commentaires, construisez votre liste à l'ancienne, avec des apostrophes et des virgules :

```
@liste = (
    'a', # un commentaire
    'b', # un autre commentaire
);
```

`Possible attempt to separate words with commas`

(Tentative possible de séparer les mots avec des virgules)

(W qw) Les listes `qw` contiennent des éléments séparés par des espacements ; les virgules ne sont donc pas nécessaires pour séparer les éléments. (Vous auriez pu utiliser des délimiteurs différents des parenthèses indiquées ici ; les accolades sont également fréquemment utilisées.)

Vous avez probablement écrit quelque chose comme ceci :

```
qw( a, b, c );
```

ce qui place les caractères virgules parmis certains éléments de la liste. Écrivez-le sans virgules si vous ne voulez pas les voir apparaître dans vos données :

```
qw( a b c );
```

`Possible memory corruption: %s overflowed 3rd argument`

(Corruption possible de la mémoire : le 3e argument de `%s` a débordé)

(F) Un `ioctl` ou un `fcntl` a renvoyé plus que ce Perl demandait. Perl anticipe un tampon de taille raisonnable mais place un octet de sentinelle à la fin du tampon au cas où. On a écrasé cet octet de sentinelle et Perl suppose que la mémoire est maintenant corrompue.

`pragma "attrs" is deprecated, use "sub NAME : ATTRS" instead`

(le pragma "`attrs`" est déprécié, utilisez à la place "`sub NOM : ATTRS`")

(W deprecated) Vous avez écrit quelque chose comme ceci :

```
sub fait {
    use attrs qw(locked)
}
```

Vous auriez dû utiliser à la place la nouvelle syntaxe de déclaration :

```
sub fait : locked
{
    ...
```

Le pragma `use attrs` est maintenant obsolète et n'est fourni que pour assurer une compatibilité antérieure.

`Precedence problem: open %s should be open(%s)`

(Problème de précédence : `open %s` devrait être `open(%s)`)

(S precedence) L'ancienne construction irrégulière :

```
open TRUC || die;
```

est maintenant mal interprétée en :

```
open(TRUC || die);
```

à cause la stricte régularisation de la grammaire de Perl 5 en opérateurs unaires et opérateur de listes. (L'ancien `open` était un peu des deux.) Vous devez mettre des parenthèses autour du handle de fichier ou utiliser le nouvel opérateur `or` à la place de `||`.

`Premature end of script headers`

(Fin prématurée des en-têtes du script)

Voir `Server error`.

`print() on closed filehandle %s`

(`print()` sur le handle de fichier fermé `%s`)

(W closed) Le handle de fichier dans lequel vous imprimez s'est retrouvé fermé à un moment auparavant. Vérifier le flux logique de votre programme.

`printf() on closed filehandle %s`

(`printf()` sur le handle de fichier fermé `%s`)

(W closed) Le handle de fichier dans lequel vous écrivez s'est retrouvé fermé à un moment auparavant. Vérifier le flux logique de votre programme.

`Process terminated by SIG%s`

(Processus terminé par `SIG%s`)

(W) Il s'agit d'un message standard formulé par les applications OS/2, alors que les applications Unix meurent en silence. C'est considéré comme une caractéristique du portage sous OS/2. Cet avertissement peut facilement être désactivé en positionnant les gestionnaires de signaux appropriés. Voir également « Process terminated by SIGTERM/SIGINT » (*N.d.T.* : Processus terminé par SIGTERM/SIGINT) dans le fichier *README.os2* livré avec la distribution de Perl.

`Prototype mismatch: %s vs %s`

(Non correspondance de prototypes : *%s* contre *%s*)

(S) La sous-fonction qui est en train d'être déclarée ou définie a été précédemment déclarée ou définie avec un prototype différent.

`Range iterator outside integer range`

(Itérateur d'intervalle en dehors d'un intervalle)

(F) L'un (ou les deux) des arguments numériques de l'opérateur d'intervalle `..` est en dehors de l'intervalle qui peut être représenté en interne par des entiers. Contourner cela est en forçant Perl à utiliser des incréments de chaînes magiques en ajoutant `0` au début de vos nombres.

`readline() on closed filehandle %s`

(`readline()` sur un handle de fichier fermé *%s*)

(W closed) Le handle de fichier dans lequel vous êtes en train de lire s'est retrouvé fermé un moment auparavant. Vérifier le flux logique de votre programme.

`realloc() of freed memory ignored`

(`realloc()` sur de la mémoire libérée, ignoré)

(S malloc) Une routine interne a appelé `realloc` sur quelque chose qui avait déjà été libéré.

`Reallocation too large: %lx`

(Réallocation trop grande : *%lx*)

(F) Vous ne pouvez allouez plus de 64K sur une machine MS-DOS.

`Recompile perl with -DDEBUGGING to use -D switch`

(Recompilez Perl avec `-DDEBUGGING` pour utiliser l'option `-D`)

(F debugging) Vous ne pouvez utiliser l'option **`-D`** que si Perl a été compilé avec le code qui produit la sortie désirée ; comme cela implique une certaine surcharge, cela n'a pas été inclus dans votre instalaltion de Perl.

`Recursive inheritance detected in package '%s'`

(Héritage récursif détecté dans le paquetage '*%s*')

(F) Plus de 100 niveaux d'héritage ont été utilisés. Ceci indique probablement une boucle accidentelle dans votre hiérarchie de classes.

`Recursive inheritance detected while looking for method '%s' in package '%s'`

(Héritage récursif détecté en recherchant la méthode '*%s*' dans le paquetage '*%s*')

(F) Plus de 100 niveaux d'héritages ont été utilisés. Ceci indique probablement une boucle accidentelle dans votre hiérarchie de classes.

`Reference found where even-sized list expected`

(Référence trouvée où une liste de taille paire était attendue)

(W misc) Vous avez donné une seule référence là où Perl s'attendait à avoir une liste avec un nombre pair d'éléments (pour affection à un hachage). Cela veut habituellement dire que vous avez utilisé le constructeur de hachage anonyme alors que vous vouliez utiliser des parenthèses. Dans tous les cas, un hachage exige des paires clef/valeur.

```
%hachage = { un => 1, deux => 2, };        # MAUVAIS
%hachage = [ qw( un tableau anonyme /)];   # MAUVAIS
%hachage = ( un => 1, deux => 2, );        # juste
%hachage = qw( un 1 deux 2 );              # également correct
```

`Reference is already weak`

(La référence est déjà lâche)

(W misc) Vous avez tenté de relâcher une référence qui était déjà lâche. Ceci n'a aucun effet.

`Reference miscount in sv_replace()`

(Mauvais compte de références dans `sv_replace()`)

(W internal) On a passé à la fonction interne `sv_replace` un nouvel SV avec un nombre de références différent de 1.

`regexp *+ operand could be empty`

(L'opérande *+ de l'expression régulière pourrait être vide)

(F) La partie d'une expression régulière sujette au quantifiant * ou + pourrait correspondre à une chaîne vide.

`regexp memory corruption`

(Corruption de la mémoire de l'expression régulière)

(P) Le moteur d'expressions régulières s'est trompé avec ce que l'expression régulière lui a donné.

`regexp out of space`

(Plus d'espace pour l'expression régulière)

(P) Il s'agit d'une erreur « qui ne peut pas arriver » car `safemalloc` aurait dû la capturer plus tôt.

`Reversed %s= operator`

(Opérateur *%s=* inversé)

(W syntax) Vous avez écrit votre opérateur d'affectation à l'envers. Le = doit toujours arriver en dernier, pour éviter l'ambiguïté avec les opérateurs unaires suivants.

`Runaway format`

(Format s'échappant)

(F) Votre format contenait la séquence de répétition jusqu'à un caractère blanc ~~ mais il a produit 200 lignes d'un coup et la 200^{e} ligne ressemblait exactement à la 199ème. Apparemment vous ne vous êtes pas arrangé pour que les arguments soit

consommés, soit en utilisant ^ au lieu de @ (pour les variables scalaires), soit en faisant un shift ou un pop (pour les variables de tableaux).

`Scalar value @%s[%s] better written as $%s[%s]`

(Valeur scalaire @%s[%s] mieux écrite avec $%s[%s])

(W syntax) Vous avez utilisé une tranche de tableau (indiqué par @) pour sélectionner un seul élément d'un tableau. Généralement il vaut mieux demander une valeur scalaire (indiquée par $). La différence est que $truc[&machin] se comporte toujours comme un scalaire, à la fois lors de son affectation et lors de l'évaluation de son argument, alors que @truc[&machin] se comporte comme une liste lorsque vous y affectez quelque chose en fournissant un contexte de liste à son indice, ce qui peut donner des choses étranges si vous vous attendez à un seul indice.

D'un autre côté, si vous espériez vraiment traiter l'élément du tableau comme une liste, vous devez regarder comment fonctionnent les références car Perl ne va pas par magie faire la conversion entre les scalaires et les listes à votre place.

`Scalar value @%s{%s} better written as $%s{%s}`

(Valeur scalaire @%s{%s} mieux écrite avec $%s{%s})

(W syntax) Vous avez utilisé une tranche de hachage (indiquée par @) pour sélectionner un seul élément d'un hachage. Généralement il vaut mieux demander une valeur scalaire (indiquée par $). La différence est que $truc{&machin} se comporte toujours comme un scalaire, à la fois lors de son affectation et lors de l'évaluation de son argument, alors que @truc{&machin} se comporte comme une liste lorsque vous y affectez quelque chose en fournissant un contexte de liste à son indice, ce qui peut donner des choses étranges si vous vous attendez à un seul indice.

D'un autre côté, si vous espériez vraiment traiter l'élément du hachage comme une liste, vous devez regarder comment fonctionnent les références car Perl ne va pas par magie faire la conversion entre les scalaires et les listes à votre place.

`Script is not setuid/setgid in suidperl`

(Le script n'est pas en setuid/setgid dans suidperl)

(F) Curieusement, le programme *suidperl* a été invoqué sur un script qui n'a pas de bit setuid ou setgid bit positionné. Cela n'a vraiment aucun sens.

`Search pattern not terminated`

(Motif de recherche non terminé)

(F) L'analyseur lexical n'a pas pu trouver le délimiteur final d'une construction // ou m{}. Souvenez-vous que les délimiteurs fonctionnant par paires « ouvrant/fermant » (tels que les parenthèses, les crochets, les accolades ou les signes inférieur/supérieur) comptent les niveaux imbriqués. Omettre le $ au début d'une variable $m peut causer cette erreur.

`%sseek() on unopened file`

(%sseek() sur un fichier non ouvert)

(W unopened) Vous avez essayé d'utiliser la fonction seek ou sysseek sur un handle de fichier qui soit n'a jamais été ouvert, soit a été fermé depuis.

`select not implemented`

(`select` non implémenté)

(F) Cette machine n'implémente pas l'appel système `select`.

`sem%s not implemented`

(`sem%s` non implémenté)

(F) Vous n'avez pas d'implémentation des sémaphores des IPC System V sur votre système.

`semi-panic: attempt to dup freed string`

(Demi-panique : tentative de dupliquer une chaîne libérée)

(S internal) La routine interne `newSVsv` a été appelée pour dupliquer un scalaire qui avait été marqué auparavant comme étant libre.

`Semicolon seems to be missing`

(Il semble manquer un point-virgule)

(W semicolon) Une quasi-erreur de syntaxe a probablement été causée par un point-virgule manquant ou un autre opérateur manquant peut-être, comme une virgule.

`send() on closed socket`

(`send()` sur une socket fermée)

(W closed) La socket dans laquelle vous envoyer des données s'est retrouvée fermée à un moment auparavant. Vérifiez le flux logique de votre programme.

`Sequence (? incomplete`

(Séquence `(?` incomplète)

(F) Une expression régulière finissait par une extension incomplète `(?`.

`Sequence (?#... not terminated`

(Séquence `(?...` non terminée)

(F) Un commentaire d'une expression régulière doit se terminer par une parenthèse fermée. Les parenthèses imbriquées ne sont pas autorisées.

`Sequence (?%s...) not implemented`

(Séquence `(?%s...)` non implémentée)

(F) Une proposition d'extension des expressions régulières a réservé le caractère mais n'a pas encore été écrite.

`Sequence (?%s...) not recognized`

(Séquence `(?%s...)` non reconnue)

(F) Vous avez utilisé une expression régulière qui n'a aucun sens.

`Server error`

(Erreur du serveur)

Il s'agit du message d'erreur généralement vu dans une fenêtre de navigateur lorsque vous essayez de lancer un programme CGI (y compris SSI) sur le Web. Le véritable texte d'erreur varie largement d'un serveur à l'autre. Les variantes les plus

fréquemment vues sont « `500 Server error` », « `Method (`*`something`*`) not permitted` », « `Document contains no data` », « `Premature end of script headers` » et « `Did not produce a valid header` » (« `Erreur de serveur 500` », « `Méthode (`*`quelque_chose`*`) non permise` », « `Le document ne contient aucune donnée` », « `Fin prématurée des en-têtes de script` » et « `N'a pas produit d'en-tête valide` »).

Il s'agit d'une erreur CGI, non d'une erreur Perl.

Vous devez vous assurer que votre script soit exécutable, accessible par l'utilisateur sous lequel le script CGI tourne (qui n'est probablement pas le compte utilisateur sous lequel vous avez fait vos tests), qu'il ne repose sur aucune variable d'environnement (telle que `PATH`) que l'utilisateur ne va pas avoir et qu'il ne se trouve pas à un emplacement inatteignables pour le serveur CGI. Pour plus d'informations, merci de voir les liens suivants :

- *http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html*
- *http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html*
- *ftp://rtfm.mit.edu/pub/usenet/news.answers/www/cgi-faq*
- *http://hoohoo.ncsa.uiuc.edu/cgi/interface.html*
- *http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html*

Vous devriez également regarder la FAQ Perl.

`setegid() not implemented`

(`setegid()` non implémenté)

(F) Vous avez essayé d'affecter `$)` mais votre système d'exploitation ne supporte pas l'appel système `setegid` (ou un équivalent) ou du moins c'est ce que pense *Configure*.

`seteuid() not implemented`

(`seteuid()` non implémenté)

(F) Vous avez essayé d'affecter `$>` mais votre système d'exploitation ne supporte pas l'appel système `seteuid` (ou un équivalent) ou du moins c'est ce que pense *Configure*.

`setpgrp can't take arguments`

(`setpgrp` ne peut pas prendre d'arguments)

(F) Votre système dispose de la fonction `setpgrp` de BSD 4.2, qui ne prend pas d'argument, au contraire de `setpgid` POSIX, qui prend un ID de processus et un ID de groupe de processus.

`setrgid() not implemented`

(`setrgid()` non implémenté)

(F) Vous avez essayé d'affecter `$(` mais votre système d'exploitation ne supporte pas l'appel système `setrgid` (ou un équivalent) ou du moins c'est ce que pense *Configure*.

`setruid() not implemented`

(`setruid()` non implémenté)

(F) Vous avez essayé d'affecter `$<` mais votre système d'exploitation ne supporte pas l'appel système `setruid` (ou un équivalent) ou du moins c'est ce que pense *Configure*.

`setsockopt() on closed socket %s`

(`setsockopt()` sur une socket fermée *%s*)

(W closed) Vous avez essayé de positionner des options sur une socket fermée. Peut-être avez-vous oublié de vérifier la valeur renvoyée par votre appel à `socket` ?

`Setuid/gid script is writable by world`

(Script `setuid/setgid` modifiable par le monde entier)

(F) L'émulateur setuid ne lancera pas un script modifiable par le monde entier car le monde entier peut très bien déjà l'avoir modifié.

`shm%s not implemented`

(`shm%s` non implémenté)

(F) Vous n'avez pas d'implémentation de la mémoire partagée des IPC System V sur votre système.

`shutdown() on closed socket %s`

(`shutdown()` sur une socket fermée *%s*)

(W closed) Vous avez essayé de faire un `shutdown` sur socket fermée. Cela semble quelque peu superflu.

`SIG%s handler "%s" not defined`

(Gestionnaire "*%s*" pour `SIG%s` non défini)

(W signal) Le gestionnaire pour le signal nommé dans `%SIG` n'existe en fait pas. Peut-être l'avez-vous mis dans le mauvais paquetage ?

`sort is now a reserved word`

(`sort` est maintenant un mot réservé)

(F) Il s'agit d'un ancien message d'erreur que pratiquement personne ne rencontrera plus. Mais avant que `sort` soit un mot-clef, les gens l'utilisaient parfois comme un handle de fichier.

`Sort subroutine didn't return a numeric value`

(Le sous-programme de `sort` n'a pas renvoyé de valeur numérique)

(F) Une routine de comparaison pour `sort` doit retourner un nombre. Vous l'avez probablement foutu en l'air en n'utilisant pas `<=>` ou `cmp`, ou en ne les utilisant pas correctement.

`Sort subroutine didn't return single value`

(Le sous-programme de `sort` n'a pas renvoyé une valeur unique)

(F) Une routine de comparaison pour `sort` ne peut pas renvoyer une valeur de liste avec plus ou moins d'un élément.

`Split loop`

(Boucle dans un `split`)

(P) Le `split` bouclait sans fin. (Évidemment, un `split` ne devrait boucler plus de fois qu'il n'y a de caractères en entrée, ce qui s'est passé.)

`Stat on unopened file <%s>`

(`stat` sur un fichier non ouvert <*%s*>)

(W unopened) Vous avez essayé d'utiliser la fonction `stat` (ou un test de fichier équivalent) sur un handle de fichier qui n'a jamais été ouvert ou qui a été fermé depuis.

`Statement unlikely to be reached`

(Instruction peu susceptible d'être atteinte)

(W exec) Vous avez fait un `exec` avec des instructions le suivant autres qu'un `die`. Il s'agit presque toujours d'une erreur car `exec` ne revient jamais à moins qu'il n'y ait une erreur. Vous vouliez probablement utiliser `system` à la place, qui lui revient. Pour supprimer cet avertissement, mettez le `exec` dans un bloc tout seul.

`Strange *+?{} on zero-length expression`

(*+?{} étrange sur une expression de longueur nulle)

(W regexp) Vous avez appliqué un quantifiant d'expression régulière à un endroit où cela n'a aucun sens, comme dans une assertion de longueur nulle. Essayez de plutôt mettre le quantifiant à l'intérieur de l'assertion. Par exemple, le moyen de réussir la correspondance avec `abc`, si c'est suivi par trois répétitions de `xyz` est `/abc(?=(?:xyz){3})/`, et non `/abc(?=xyz){3}/`.

`Stub found while resolving method '%s' overloading '%s' in package '%s'`

(Souche trouvée lors de la résolution de la méthode '*%s*' surchargeant '*%s*' dans le paquetage '*%s*')

(P) La résolution de la surcharge dans l'arbre `@ISA` a peut être été rompue par des souches (*stubs*) d'importation. Les souches ne devraient jamais être implicitement créées mais les appels explicites à `can` peuvent rompre cela.

`Subroutine %s redefined`

(Sous-programme *%s* redéfini)

(W redefine) Vous avez redéfini un sous-programme. Pour supprimer cet avertissement, écrivez :

```
{
    no warnings;
    eval "sub name { ... }";
}
```

`Substitution loop`

(Boucle dans une substitution)

(P) La substitution boucle sans fin. (Évidemment, une substitution ne devrait boucler plus de fois qu'il n'y a de caractères en entrée, ce qui s'est passé.)

`Substitution pattern not terminated`

(Motif pour la substitution non terminé)

(F) L'analyseur lexical ne peut trouver le délimiteur intérieur d'une construction `s///` ou `s{}{}`. Souvenez-vous que les délimiteurs fonctionnant par paires « ouvrant/fermant » (tels que les parenthèses, les crochets, les accolades ou les signes inférieur/supérieur) comptent les niveaux imbriqués. Omettre le `$` au début d'une variable `$s` peut causer cette erreur.

`Substitution replacement not terminated`

(Remplacement pour la substitution non terminé)

(F) L'analyseur lexical ne peut trouver le délimiteur final d'une construction `s///` ou `s{}{}`. Souvenez-vous que les délimiteurs fonctionnant par paires « ouvrant/fermant » (tels que les parenthèses, les crochets, les accolades ou les signes inférieur/supérieur) comptent les niveaux imbriqués. Omettre le `$` au début d'une variable `$s` peut causer cette erreur.

`substr outside of string`

(`substr` à l'extérieur d'une chaîne)

(W substr|F) Vous avez essayé de référencer un `substr` qui pointait en dehors d'une chaîne. C'est-à-dire que la valeur absolue du décalage est plus grande que la longueur de la chaîne. Cet avertissement est fatal si `substr` est utilisé dans un contexte de lvalue (par exemple comme opérateur du côté gauche d'une affectation ou comme argument d'un sous-programme).

`suidperl is no longer needed since %s`

(suidperl n'est plus nécessaire depuis *%s*)

(F) Votre Perl a été compilé avec **`-DSETUID_SCRIPTS_ARE_SECURE_NOW`** mais une version de l'émulateur setuid est tout de même arrivé à se lancer d'une manière ou d'une autre.

`switching effective %s is not implemented`

(l'échange de *%s* effectif n'est pas implémenté)

(F) Avec le pragma `use filetest` actif, nous ne pouvons pas échanger les UID ou les GID réels et effectifs.

`syntax error`

(erreur de syntaxe)

(F) Ce message veut probablement dire que vous avez eu une erreur de syntaxe. Les raisons habituelles incluent :

- un mot clef est mal orthographié ;
- il manque un point-virgule ;
- il manque une virgule ;
- il manque une parenthèse ouvrante ou fermante ;
- il manque un crochet ouvrant ou fermant ;
- il manque une apostrophe ou un guillemet fermant.

Souvent un autre message sera associé avec l'erreur de syntaxe donnant plus d'informations. (Parfois, cela aide d'activer **`-w`**.) Le message d'erreur en lui-même dit souvent à quel endroit de la ligne Perl a décider de s'arrêter. Parfois la véritable erreur se trouve quelque tokens avant car Perl est doué pour comprendre ce que l'on entre aléatoirement. Occasionnellement, le numéro de ligne peut être inexact et la seule manière de savoir ce qui ce passe est d'appeler de façon répétitive, une fois tous les 36 du mois, *perl -c* en éliminant à chaque fois la moitié du programme pour voir où l'erreur disparaît. Il s'agit d'une sorte de version cybernétique du jeu des 20 questions.

`syntax error at line %d: '%s' unexpected`

(erreur de syntaxe à la ligne *%d* : '*%s*' inattendu)

(A) Vous avez accidentellement lancé votre script *via* le Bourne shell au lieu de Perl. Vérifier la ligne `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`System V %s is not implemented on this machine`

(*%s* de System V n'est pas implémenté sur cette machine)

(F) Vous avez essayé de faire quelque chose avec une fonction commençant par `sem`, `shm` ou `msg` mais cet IPC System V n'est pas implémenté sur votre machine. (Sur certaines machines, la fonctionnalité peut exister mais peut ne pas être configurée.)

`syswrite() on closed filehandle`

(`syswrite()` sur un handle de fichier fermé)

(W closed) Le handle de fichier dans lequel vous écrivez s'est retrouvé fermé à un moment auparavant. Vérifiez le flux logique de votre programme.

`Target of goto is too deeply nested`

(La cible du `goto` est trop profondément imbriquée)

(F) Vous avez essayé d'utiliser `goto` pour atteindre une étiquette qui était trop profondément imbriquée pour que Perl puisse l'atteindre. Perl vous fait une faveur en vous le refusant.

`tell() on unopened file`

(`tell()` sur un fichier non ouvert)

(W unopened) Vous avez essayé d'utiliser la fonction `tell()` sur un handle de fichier qui n'a jamais été ouvert ou qui a été fermé depuis.

`Test on unopened file %s`

(Test sur un fichier non ouvert *%s*)

(W unopened) Vous avez essayé d'invoquer un opérateur de test de fichier sur un handle de fichier qui n'est pas ouvert. Vérifiez votre logique.

`That use of $[ is unsupported`

(Cette utilisation de $[n'est pas supportée)

(F) L'affectation de `$[` est maintenant strictement circonscrite et interprétée comme une directive de compilation. Vous devez maintenant écrire une seule possibilité parmi celles-ci :

```
$[ = 0;
$[ = 1;
...
local $[ = 0;
local $[ = 1;
...
```

Ceci est imposé pour prévenir le problème d'un module changeant la base des tableaux depuis un autre module par inadvertance.

`The %s function is unimplemented`

(La fonction `%s` n'est pas implémentée)

La fonction indiquée n'est pas implémentée sur cette architecture, selon les tests de *Configure*.

`The crypt() function is unimplemented due to excessive paranoia`

(La fonction `crypt()` n'est pas implémentée à cause d'une paranoïa excessive)

(F) *Configure* n'a pas pu trouver la fonction `crypt` sur votre machine, probablement parce que votre vendeur ne l'a pas fournie, probablement parce qu'il pense que le gouvernement des États-Unis pense que c'est un secret ou du moins continue à prétendre que c'en est un.

`The stat preceding -l _ wasn't an lstat`

(Le `stat` précédant `-l _` n'était pas un `lstat`)

(F) Cela n'a pas de sens de vérifier si le tampon courant de `stat` est un lien symbolique si le dernier stat que vous avez écrit dans le tampon a déjà passé le lien symbolique pour obtenir le fichier réel. Utilisez un véritable nom de fichier à la place.

`This Perl can't reset CRTL environ elements (%s)`

(Cette copie de Perl ne peut pas réinitialiser les éléments de l'environnement de CRTL (`%s`))

`This Perl can't set CRTL environ elements (%s = %s)`

(Cette copie de Perl ne peut pas positionner les éléments de l'environnement de CRTL (`%s = %s`))

(W internal) Il s'agit d'avertissements spécifiques à VMS. Vous avez essayé de modifier ou de supprimer un élément du tableau d'environnement interne de CRTL mais votre copie de Perl n'a pas été compilée avec un CRTL contenant la fonction interne `setenv`. Vous devrez recompiler Perl avec un CRTL qui le contienne ou redéfinir `PERL_ENV_TABLES` (voir *perlvms*(1)) pour que le tableau d'environnement ne soit pas la cible de la modification de `%ENV` qui a produit cet avertissement.

`times not implemented`

(`times` non implémenté)

(F) Votre version de la bibliothèque C ne fait apparemment pas de `times`. Je suspecte que vous n'êtes pas sous Unix.

`Too few args to syscall`

(Trop peu d'arguments pour `syscall`)

(F) Il doit y avoir au moins un argument à `syscall` pour spécifier l'appel système à appeler, pauvre étourdi !

`Too late for "-T" option`

(Trop tard pour l'option `"-T"`)

(X) La ligne `#!` (ou son équivalent local) dans un script Perl contient l'option **`-T`** mais Perl n'a pas été invoqué avec **`-T`** sur la ligne de commande. Il s'agit d'une erreur car le temps que Perl découvre un **`-T`** dans un script, il est trop tard pour marquer proprement l'intégralité de l'environnement. Donc Perl abandonne.

Si le script Perl a été exécuté en tant que commande en utilisant le mécanisme `#!` (ou son équivalent local), cette erreur peut habituellement être corrigée en éditant la ligne `#!` pour que l'option **`-T`** fasse partie du premier argument de Perl : ex. changer `perl -n -T` en `perl -T -n`.

Si le script Perl a été exécuté en tant que *perl nom_script* alors l'option **`-T`** doit apparaître sur la ligne de commande : *perl -T nom_script*.

`Too late for "-%s" option`

(Trop tard pour l'option "*-%s*")

(X) La ligne `#!` (ou son équivalent local) dans un script Perl contient l'option **`-M`** ou `-m`. Il s'agit d' une erreur car les options **`-M`** et **`-m`** ne sont pas prévues pour être utilisées à l'intérieur des scripts. Utilisez `use` à la place.

`Too late to run %s bloc`

(Trop tard pour lancer le bloc *%s*)

(W void) Un bloc `CHECK` ou `INIT` a été défini à l'exécution, lorsque l'opportunité de le lancer est déjà passée. Vous chargez peut-être un fichier avec `require` ou `do` alors que vous devriez utiliser `use` à la place. Ou vous devriez peut-être mettre le `require` ou le `do` à l'intérieur d'un bloc `BEGIN`.

`Too many ('s`

(Trop de « (»)

`Too many )'s`

(trop de «) »)

(A) Vous avez lancé accidentellement votre script *via csh* au lieu de Perl. Vérifiez la ligne `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`Too many args to syscall`

(Trop d'argument pour `syscall`)

(F) Perl supporte un maximum de 14 arguments seulement pour `syscall`.

`Too many arguments for %s`

(Trop d'argument pour *%s*)

(F) La fonction demande moins d'arguments que vous en avez spécifiés.

`trailing \ in regexp`

(\ à la fin de l'expression régulière)

(F) L'expression régulière se termine par un antislash qui n'est pas protégé par un antislash. Faites-le.

`Transliteration pattern not terminated`

(Motif pour la traduction non terminé)

(F) L'analyseur lexical n'a pas pu trouver le délimiteur intérieur d'une construction `tr///`, `tr[][]`, `y///` ou `y[][]`. Omettre le `$` au début des variables `$tr` ou `$y` peut causer cette erreur.

`Transliteration replacement not terminated`

(Remplacement pour la traduction non terminé)

(F) L'analyseur lexical n'a pas pu trouver le délimiteur final d'une construction `tr///` ou `tr[][]`.

`truncate not implemented`

(`truncate` non implémenté)

(F) Votre machine n'implémente pas de mécanisme de troncation de fichier que *Configure* puisse reconnaître.

`Type of arg %d to %s must be %s (not %s)`

(Le type de l'argument *%d* de *%s* doit être *%s* (pas *%s*))

(F) Cette fonction exige que l'argument à cette position soit d'un certain type. Les tableaux doivent être *@NOM* ou *@{EXPR}*. Les hachages doivent être *%NOM* ou *%{EXPR}*. Aucun déréférencement implicite n'est permis — utilisez la forme *{EXPR}* en tant que déréférencement explicite.

`umask: argument is missing initial 0`

(`umask` : il manque `0` au début de l'argument)

(W umask) Un umask de `222` est incorrect. Il devrait être `0222` car les littéraux octaux commencent toujours par `0` en Perl, tout comme en C.

`umask not implemented`

(`umask` non implémenté)

(F) Votre machine n'implémente pas la fonction `umask` et vous avez essayé de l'utiliser pour restreindre les permissions pour vous-même. (*EXPR* `& 0700`).

`Unable to create sub named "%s"`

(Incapable de créer le sous-programme nommé "*%s*")

(F) Vous avez tenté de créer ou d'accéder à un sous-programme avec un nom illégal.

`Unbalanced context: %d more PUSHes than POPs`

(Contexte déséquilibré : *%d* `PUSH` de plus que de `POP`)

(W internal) Le code de sortie a détecté une incohérence interne entre le nombre de contextes d'exécution dans lesquels on est entré et depuis lesquels on est sorti.

`Unbalanced saves: %d more saves than restores`

(Sauvegardes déséquilibrées : *%d* sauvegardes de plus que de restaurations)

(W internal) Le code de sortie a détecté une incohérence interne entre le nombre de variables qui ont été sauvegardées par `local`.

`Unbalanced scopes: %d more ENTERs than LEAVEs`

(Portées déséquilibrées : *%d* `ENTER` de plus que de `LEAVE`)

(W internal) Le code de sortie a détecté une incohérence interne entre le nombre de blocs dans lequel on est rentré et depuis lesquels on est sorti.

`Unbalanced tmps: %d more allocs than frees`

(Valeurs temporaires déséquilibrés : *%d* allouées de plus que de libérées)

(W internal) Le code de sortie a détecté une incohérence interne entre le nombre de scalaires mortels qui ont été alloués et libérés.

`Undefined format "%s" called`

(Format indéfini "*%s*" appelé)

(F) Le format indiqué ne semble pas exister. Peut-être est-il réellement dans un autre paquetage ?

`Undefined sort subroutine "%s" called`

(Sous-programme de `sort` indéfini "*%s*" appelé)

(F) La routine de comparaison de `sort` spécifiée ne semble pas exister. Peut-être est-elle dans un autre paquetage?

`Undefined subroutine &%s called`

(Sous-programme indéfini `&`*%s* appelé)

(F) Le sous-programme indiqué n'a pas été défini ou s'il l'a été, il a été indéfini depuis.

`Undefined subroutine called`

(Sous-programme indéfini appelé)

(F) Le sous-programme anonyme que vous essayez d'appeler n'a pas été défini ou s'il l'a été, il a été indéfini depuis.

`Undefined subroutine in sort`

(Sous-programme indéfini dans `sort`)

(F) La routine de comparaison de `sort` spécifiée est déclarée mais ne semble pas avoir été définie pour le moment.

`Undefined top format "%s" called`

(Format d'en-tête indéfini "*%s*" appelé)

(F) Le format indiqué ne semble pas exister. Peut-être est-il en réalité dans un autre paquetage?

`Undefined value assigned to typeglob`

(Valeur indéfinie affectée à un typeglob)

(W misc) Une valeur indéfinie a été affectée à un typeglob, comme `*truc = undef`. Ceci n'a aucun effet. Il est possible que vous vouliez dire en réalité `undef *truc`.

`unexec of %s into %s failed!`

(Échec de l'`unexec` de *%s* dans *%s* !)

(F) La routine `unexec` a échoué pour une raison ou pour une autre. Voir votre représentant local FSF (*Free Software Fondation*), qui l'a probablement mis là en premier lieu.

`Unknown BYTEORDER`

(`BYTEORDER` inconnu)

(F) Il n'y a aucune fonction d'échange d'octets pour une machine fonctionnant avec cet ordre d'octets.

`Unknown open() mode '%s'`

(Mode d'`open()` '*%s*' inconnu)

(F) Le deuxième argument d'un `open` à trois arguments n'est pas dans la liste des modes valides : `<`, `>`, `> >`, `+<`, `+>`, `+> >`, `-|`, `|-`.

`Unknown process %x sent message to prime_env_iter: %s`

(Le processus inconnu *%x* a envoyé un message au `prime_env_iter` : *%s*)

(P) Il s'agit d'une erreur spécifique à VMS. Perl lisait des valeurs pour `%ENV` avant de le parcourir et quelqu'un d'autre a collé un message dans le flux de données que Perl attendait. Quelqu'un de très perturbé ou peut-être essayant de subvertir le contenu de `%ENV` pour Perl dans un but néfaste.

`unmatched () in regexp`

(`()` sans correspondance dans l'expression régulière)

(F) Les parenthèses non protégées par un anti-slash doivent toujours être équilibrées dans les expressions régulières. Si vous êtes utilisateur de *vi*, la touche `%` est parfaitement adaptée pour trouver la parenthèse correspondante.

`Unmatched right %s bracket`

(Signe *%s* fermant sans correspondance)

(F) L'analyseur lexical a compté plus d'accolades ou de crochets fermants que d'ouvrants, vous avez donc probablement oublié de mettre un signe ouvrant correspondant. En règle générale, vous trouverez le signe manquant (si l'on peut s'exprimer ainsi) près de l'endroit où vous étiez en train d'éditer en dernier.

`unmatched [] in regexp`

([] sans correspondance dans l'expression régulière)

(F) Les crochets autour d'une classe de caractères doivent se correspondre. Si vous voulez inclure un crochet fermant dans une classe de caractères, protégez le avec un antislash ou mettez le en premier.

`Unquoted string "%s" may clash with future reserved word`

(Chaîne sans guillemets ni apostrophes "*%s*" susceptible d'être en conflit avec un mot réservé dans le futur)

(W reserved) Vous utilisez un mot simple qui pourrait être un jour proclamé comme étant un mot réservé. Le mieux est de mettre un tel mot entre guillemets ou entre apostrophes ou en lettres capitales ou d'insérer un souligné dans son nom. Vous pouvez également le déclarer en tant que sous-programme.

`Unrecognized character %s`

(Caractère *%s* non reconnu)

(F) L'analyseur syntaxique de Perl ne sait absolument pas quoi faire avec le caractère spécifié dans votre script Perl (ou dans un `eval`). Peut-être avez-vous essayé de lancer un script compressé, un programme binaire ou un répertoire en tant que programme Perl.

`Unrecognized escape \\%c passed through`

(Rencontre d'une séquence d'échappement *\\%c* non reconnu)

(W misc) Vous avez employé une combinaison avec un caractère antislash qui n'est pas reconnue.

`Unrecognized signal name "%s"`

(Nom de signal "`%s`" non reconnu)

(F) Vous avez spécifié un nom de signal à la fonction `kill` qui n'est pas reconnue. Faites `kill -l` dans votre shell pour voir les noms de signaux valides sur votre système.

`Unrecognized switch: -%s (-h will show valid options)`

(Option on reconnue : `-%s` (`-h` donne les options valides))

(F) Vous avez spécifié une option illégale pour Perl. Ne faites pas cela. (Si vous pensez ne pas l'avoir fait, vérifiez la ligne `#!` pour voir si elle ne spécifie pas la mauvaise option dans votre dos.)

`Unsuccessful %s on filename containing newline`

(Échec de `%s` sur un nom de fichier contenant un saut de ligne)

(W newline) Une opération de fichier a été tentée sur un nom de fichier et cette opération a échoué, probablement parce que le nom de fichier contenait un caractère de saut de ligne, probablement parce que vous avez oublié de le faire sauter par un `chop` ou un `chomp`.

`Unsupported directory function "%s" called`

(Fonction "`%s`" sur les répertoires non supportée appelée)

(F) Votre machine ne supporte pas `opendir` et `readdir`.

`Unsupported function fork`

(Fonction `fork` non supportée)

(F) Votre version d'exécutable ne supporte pas le `fork`.

Remarquez que sur certains systèmes, comme OS/2, il peut y avoir différentes versions d'exécutables Perl, certaines d'entre elles supportent `fork`, d'autres non. Essayez de changer le nom que vous utilisez pour appelez Perl en `perl_`, `perl__` et ainsi de suite.

`Unsupported function %s`

(Fonction `%s` non supportée)

(F) Cette machine n'implémente apparemment pas la fonction indiquée ou du moins c'est ce que pense *Configure*.

`Unsupported socket function "%s" called`

(Fonction "`%s`" sur les sockets non supportée appelée)

(F) Votre machine ne supporte pas le mécanisme des sockets Berkeley ou du moins c'est ce que pense *Configure*.

`Unterminated <> operator`

(Opérateur `<>` non terminé)

(F) L'analyseur lexical a vu un signe inférieur à un endroit où il s'attendait à un terme, il recherche donc le signe supérieur correspondant et ne le trouve pas. Il y a des chances que vous ayez oublié des parenthèses obligatoires plus tôt dans la même ligne et vous que pensiez vraiment à un `<` signifiant « inférieur à ».

`Unterminated attribute parameter in attribute list`

(Paramètre d'attribut non terminé dans une liste d'attributs)

(F) L'analyseur lexical a vu une parenthèse ouvrante lorsqu'il analysait une liste d'attributs mais n'a pas trouvé la parenthèse fermante correspondante. Vous devez ajouter (ou enlever) un caractère antislash pour équilibrer vos parenthèses.

`Unterminated attribute list`

(Liste d'attributs non terminée)

(F) L'analyseur lexical a trouvé quelque chose d'autre qu'un identificateur simple au début d'un attribut et ce n'était pas un point-virgule ni le début d'un bloc. Peut-être avez-vous terminé trop tôt la liste de paramètres de l'attribut précédent.

`Use of $# is deprecated`

(L'utilisation de `$#` est dépréciée)

(D deprecated) Il s'agissait d'une tentative mal avisée d'émuler une fonctionnalité de *awk* mal définie. Utilisez plutôt un `printf` ou `sprintf` explicite.

`Use of $* is deprecated`

(L'utilisation de `$*` est dépréciée)

(D deprecated) Cette variable activait comme par magie la recherche de motifs en multilignes, à la fois pour vous et pour les sous-programmes malchanceux que vous seriez amené à appeler. Maintenant, vous devriez utiliser les modificateurs `//m` et `//s` pour faire cela sans les effets dangereux d'action à distance de `$*`.

`Use of %s in printf format not supported`

(Utilisation de `%s` non supportée dans un format de `printf`)

(F) Vous avez essayé d'utiliser une fonctionnalité de `printf` qui est accessible en C. Cela veut habituellement dire qu'il existe une meilleure manière de faire cela en Perl.

`Use of bare << to mean <<"" is deprecated`

(L'utilisation d'un `<<` brut signifiant `<<""` est dépréciée)

(D deprecated) Vous êtes maintenant encouragé à utiliser la forme explicite avec des guillemets ou des apostrophes si vous désirez utiliser une ligne vide comme délimiteur final d'un document « ici-même » (*here document*).

`Use of implicit split to @_ is deprecated`

(L'utilisation d'un `split` implicite sur `@_` est dépréciée)

(D deprecated) Vous donnez beaucoup de travail au compilateur lorsque vous écrasez la liste d'arguments d'un sous-programme, il vaut donc mieux affecter explicitement les résultats d'un `split` à un tableau (ou à une liste).

`Use of inherited AUTOLOAD for non-method %s() is deprecated`

(L'utilisation de l'héritage par `AUTOLOAD` pour la non-méthode `%s()` est dépréciée)

(D deprecated) Comme fonctionnalité (hum) accidentelle, les sous-programmes AUTOLOAD étaient recherchés de la même façon que les méthodes (en utilisant la hiérarchie de `@ISA`) même lorsque les sous-programmes à autocharger étaient appelés en tant que fonctions brutes (ex. : `Truc::bidule()`) et pas en tant que méthodes (par exemple, `Truc->bidule()` ou `$obj->bidule()`).

Ce bogue a été rectifié dans Perl 5.005, qui utilisait la recherche de méthodes seulement pour l'AUTOLOAD de méthodes. Cependant, il y a une base significative de code existant susceptible d'utiliser l'ancien comportement. Ainsi, en tant qu'étape intermédiaire, Perl 5.004 émettait cet avertissement optionnel quand des non-méthodes utilisaient des AUTOLOAD héritées.

La règle simple est la suivante : l'héritage ne fonctionnera pas quand il y a autochargement de non-méthodes. La simple correction pour de l'ancien code est : dans tout module qui avait l'habitude de dépendre de l'héritage d'AUTOLOAD pour les non-méthodes d'une classe de base nommée `Classe_Base`, exécutez `*AUTOLOAD = &Classe_Base::AUTOLOAD` au démarrage.

Dans le code qui actuellement fait `use AutoLoader; @ISA = qw(AutoLoader);`, vous pouvez supprimer `AutoLoader` de `@ISA` et changer `use AutoLoader;` en `use AutoLoader 'AUTOLOAD';`.

`Use of reserved word "%s" is deprecated`

(L'utilisation du mot réservé "*%s*" est dépréciée)

(D deprecated) Le mot simple indiqué est un mot réservé. Les futures versions de Perl peuvent l'utiliser en tant que mot-clef, vous feriez donc mieux de mettre explicitement le mot entre guillemets ou entre apostrophes d'une manière appropriée à son contexte d'utilisation ou d'utiliser de surcroît un nom différent. L'avertissement peut être supprimé pour les noms de sous-programmes en ajoutant un préfixe `&` ou en qualifiant la variable avec le paquetage, comme `&local()`, ou `Truc::local()`.

`Use of %s is deprecated`

(L'utilisation de *%s* est dépréciée)

(D deprecated) La construction indiquée n'est plus recommandée, généralement parce qu'il y a une meilleure façon de faire et également parce que l'ancienne méthode a de mauvais effets de bord.

`Use of uninitialized value %s`

(Utilisation d'une valeur non initialisée *%s*)

(W uninitialized) Une valeur indéfinie a été utilisée comme si elle avait déjà été définie. Elle a été interprétée comme un `""` ou un `0` mais peut-être est-ce une erreur. Pour supprimer cet avertissement, affectez une valeur définie à vos variables.

`Useless use of "re" pragma`

(Utilisation inutile du pragma "`re`")

(W) Vous avez fait un `use re` sans aucun argument. Ceci n'est vraiment pas très utile.

`Useless use of %s in void context`

(Utilisation inutile de *%s* dans un contexte vide)

(W void) Vous avez fait quelque chose sans effet de bord dans un contexte qui ne fait rien de la valeur de retour, comme une instruction ne renvoyant pas de valeur depuis un bloc ou le côté gauche d'un opérateur scalaire virgule. Par exemple, vous obtiendrez ceci si vous confondez les précédences en C avec les précédences en

Python en écrivant :

```
$un, $deux = 1, 2;
```

alors que vous vouliez dire :

```
($un, $deux) = (1, 2);
```

Une autre erreur courante est l'utilisation de parenthèses ordinaires pour construire une liste de références alors vous devriez utiliser des accolades ou des crochets, par exemple, si vous écrivez :

```
$tableau = (1,2);
```

alors que vous vouliez dire :

```
$tableau = [1,2];
```

Les crochets transforment explicitement une valeur de liste en une valeur scalaire tandis que les parenthèses ne le font pas. Donc, lorsqu'une liste entre parenthèses est évaluée dans un contexte scalaire, la virgule est traitée comme l'opérateur virgule du C, qui rejette l'argument de gauche, ce qui n'est pas ce que vous voulez.

`untie attempted while %d inner references still exist`

(untie tenté alors que *%d* références intérieures existent toujours)

(W untie) Une copie de l'objet renvoyé par `tie` (ou par `tied`) était toujours valide lorsqu'`untie` a été appelé.

`Value of %s can be "0"; test with defined()`

(La valeur de *%s* peut valoir "0" ; testez avec `defined()`)

(W misc) Dans une expression conditionnelle, vous avez utilisé `<HANDLE>`, `<*>` (glob), `each` ou `readdir` en tant que valeur booléenne. Chacune de ces constructions peut renvoyer une valeur de "0" ; cela rendrait fausse l'expression conditionnelle, ce qui n'est probablement pas ce que vous désiriez. Quand vous utilisez ces constructions dans des expressions conditionnelles, testez leur valeur avec l'opérateur `defined`.

`Value of CLI symbol "%s" too long`

(Valeur du symbole de CLI "*%s*" trop longue)

(W misc) Il s'agit d'un avertissement spécifique à VMS. Perl a essayé de lire la valeur d'un élément de `%ENV` depuis la table de symboles de CLI et a trouvé une chaîne résultante plus longue que 1 024 caractères. La valeur de retour a été tronquée à 1 024 caractères.

`Variable "%s" is not imported %s`

(La variable "*%s*" n'est pas importée de *%s*)

(F) Alors que `use strict` est activé, vous vous êtes référés à une variable globale qu'apparemment vous pensiez avoir importée depuis un autre module car quelque chose d'autre ayant le même nom (habituellement un sous-programme) est exporté par ce module. Cela veut habituellement dire que vous vous êtes trompé de drôle de caractère (`$`, `@`, `%` ou `&`) devant votre variable.

`Variable "%s" may be unavailable`

(La variable "*%s*" peut être inaccessible)

(W closure) Un sous-programme *anonyme* intérieur (imbriqué) se trouve dans un sous-programme *nommé* et encore à l'extérieur se trouve un autre sous-programme ;

et le sous-programme anonyme (le plus à l'intérieur) fait référence à une variable lexicale définie dans le sous-programme le plus à l'extérieur. Par exemple :

```
sub le_plus_exterieur { my $a; sub au_milieu { sub { $a } } }
```

Si le sous-programme anonyme est appelé ou référencé (directement ou indirectement) depuis le sous-programme le plus à l'extérieur, il va partager la variable comme vous pouviez vous y attendre. Mais si le sous-programme anonyme est appelé ou référencé lorsque le sous-programme le plus à l'extérieur est inactif, il verra la valeur de la variable partagée telle qu'elle était avant et pendant le *premier* appel au sous-programme le plus à l'extérieur, ce qui n'est probablement pas ce que vous voulez.

Dans ces circonstances, il vaut habituellement mieux rendre anonyme le sous-programme du milieu anonyme, en utilisant la syntaxe `sub {}`. Perl possède un support spécifique pour les variables partagées dans les sous-programmes anonymes imbriqués ; un sous-programme nommé entre les deux interfère avec cette fonctionnalité.

`Variable "%s" will not stay shared`

(La variable "*%s*" ne restera pas partagée)

(W closure) Un sous-programme *nommé* intérieur (imbriqué) fait référence à une variable lexicale définie dans un sous-programme extérieur.

Lorsque le sous-programme intérieur est appelé, il verra probablement la valeur de la variable du sous-programme extérieur telle qu'elle était avant et pendant le *premier* appel au sous-programme extérieur ; dans ce cas, après que le premier appel au sous-programme extérieur est terminé, les sous-programmes intérieur et extérieur ne partageront plus de valeur commune pour la variable. En d'autres termes, la variable ne sera plus partagée.

De plus, si le sous-programme extérieur est anonyme et fait référence à une variable lexicale en dehors de lui-même, alors les sous-programmes intérieur et extérieur ne partageront *jamais* la variable donnée.

Le problème peut habituellement être résolu en rendant anonyme le sous-programme intérieur, en utilisant la syntaxe `sub {}`. Lorsque des sous-programmes anonymes intérieurs, faisant référence à des variables dans des sous-programmes extérieurs, sont appelés ou référencés, ils sont automatiquement reliés à la valeur courante de ces variables.

`Variable syntax`

(Syntaxe de variable)

(A) Vous avez lancé accidentellement votre script *via csh* au lieu de Perl. Vérifiez la ligne `#!` ou lancez vous-même votre script manuellement dans Perl avec *perl nom_script*.

`Version number must be a constant number`

(Le numéro de version doit être un nombre constant)

(P) La tentative de conversion d'une instruction *`use Module n.n LISTE`* vers le bloc `BEGIN` équivalent a trouvé une incohérence interne dans le numéro de version.

`perl: warning: Setting locale failed.`

(perl : avertissement : Échec avec la configuration des locales)

(S) Le message d'avertissement complet ressemblera quelque peu à :

```
perl: warning: Setting locale failed.
    perl: warning: Please check that your locale settings:
            LC_ALL = "Fr_FR",
            LANG = (unset)
        are supported and installed on your system.
    perl: warning: Falling back to the standard locale ("C").[1]
```

(La configuration des locales qui a échoué variera.) Cette erreur indique que Perl a détecté que vous ou votre administrateur système avez configuré les variables du système ainsi nommées mais Perl n'a pas pu utiliser cette configuration. Ce n'est pas mortel, heureusement : il existe une locale par défaut appelée « C » que Perl peut et va utiliser pour le script puisse ainsi fonctionner. Mais avant que vous résolviez vraiment ce problème, vous allez toutefois avoir le même message d'erreur à chaque fois que vous lancerez Perl. La manière de vraiment résoudre le problème peut être trouvée dans *perllocale*(1), à la section « Locale Problems ».

`Warning: something's wrong`

(Avertissement : quelque chose s'est mal passé)

(W) Vous avez passé à `warn` une chaîne vide (l'équivalent d'un `warn ""`) ou vous l'avez appelé sans arguments et `$_` était vide.

`Warning: unable to close filehandle %s properly`

(Avertissement : incapable de fermer proprement le handle de fichier *%s*)

(S) Le `close` implicite effectué par un `open` a reçu une indication d'erreur sur le `close`. Cela indique habituellement que votre système de fichier a épuisé l'espace disque disponible.

`Warning: Use of "%s" without parentheses is ambiguous`

(Avertissement : L'utilisation de "*%s*" sans parenthèses est ambiguë)

(S ambiguous) Vous avez écrit un opérateur unaire suivi par quelque chose ressemblant à un opérateur binaire mais pouvant également être interprété comme un terme ou un opérateur unaire. Par exemple, si vous savez que la fonction `rand` a un argument par défaut valant `1.0` et que vous écrivez :

```
rand + 5;
```

vous pouvez *pensez* que vous avez écrit la même chose que :

```
rand() + 5;
```

alors qu'en fait vous avez obtenu :

```
rand(+5);
```

---

1. *N.d.T.* :

```
 perl : avertissement : Échec avec la configuration des locales.
    perl : avertissement : Merci de vérifier que votre configuration
                           des locales :
            LC_ALL = "Fr_FR",
            LANG = (non configuré)
        soit supportée et installée sur votre système.
    perl : avertissement : Retour aux locales standards ("C").
```

Donc mettez des parenthèses pour dire ce que vous voulez vraiment dire.

`write() on closed filehandle`

(`write()` sur un handle de fichier fermé)

(W closed) Le handle de fichier dans lequel vous écrivez s'est retrouvé fermé un moment auparavant. Vérifiez le flux logique de votre programme.

`X outside of string`

(`X` en dehors d'une chaîne)

(F) Vous aviez un canevas pour `pack` spécifiant une position relative avant le début de la chaîne en train d'être dépaqueté.

`x outside of string`

(`x` en dehors d'une chaîne)

(F) Vous aviez un canevas pour `pack` spécifiant une position relative après la fin de la chaîne en train d'être dépaqueté.

`Xsub "%s" called in sort`

(Xsub "*%s*" appelé dans un `sort`)

(F) L'utilisation d'un sous-programme externe pour la comparaison d'un `sort` n'est pas encore supportée.

`Xsub called in sort`

(Xsub "*%s*" appelé dans un `sort`)

(F) L'utilisation d'un sous-programme externe pour la comparaison d'un `sort` n'est pas encore supportée.

`You can't use -l on a filehandle`

(Vous ne pouvez pas utiliser `-l` sur un handle de fichier)

(F) Un handle de fichier représente un fichier ouvert et lorsque vous avez ouvert le fichier, il avait déjà passé tous les liens symboliques que vous essayez vraisemblablement de rechercher. Utilisez plutôt un nom de fichier.

`YOU HAVEN'T DISABLED SET-ID SCRIPTS IN THE KERNEL YET!`

(VOUS N'AVEZ TOUJOURS PAS DÉSACTIVÉ LES SCRIPTS SET-ID DANS LE NOYAU !)

(F) Et vous ne le ferez probablement jamais car vous n'avez pas les sources du noyau et probablement que votre fabriquant se moque éperdument de ce que vous voulez. La meilleure solution est de mettre une surcouche setuid en C autour de votre script en utilisant le script *wrapsuid* situé dans le répertoire eg de la distribution Perl.

`You need to quote "%s"`

(Vous devez protéger "*%s*" avec des guillemets ou des apostrophes)

(W syntax) Vous avez affecté un mot simple comme nom d'un gestionnaire de signaux. Malheureusement, vous avez déjà un sous-programme avec ce nom, ce qui veut dire que Perl 5 essayera d'appeler ce sous-programme lorsque l'affectation sera exécutée, ce qui n'est probablement pas ce que vous voulez. (Si *c'est bien* ce que vous voulez, mettez une *&* devant.)

Glossaire

Un mot référencé ailleurs dans le glossaire est présenté en italique à la manière d'un hyperlien.

addresse réseau

L'attribut le plus important d'une socket, comme le numéro de votre téléphone. Typiquement une adresse IP. Voir aussi *port*.

affectation

Un *opérateur* dont la fonction est de modifier la valeur d'une *variable*.

algorithme

Séquence d'instructions clairement définie dans le langage de la machine (virtuelle).

alias

Surnom d'une variable qui se comporte comme le nom original. Par exemple, un alias temporaire est créé implicitement pour la variable de boucle dans l'instruction `foreach`, avec les opérateurs `map` et `grep` dans la variable `$_`, dans les variables `$a` et `$b` lors de l'appel de la fonction de comparaison `sort` ou bien pour les *paramètres réels* d'un appel de fonction dans la variable `@_`. Des alias permanents sont créés explicitement dans les *paquetages* par l'*importation* des symboles ou par affectation dans les *typeglobs*. Des alias à portée lexicale dans les paquetages sont créés avec la déclaration `our`.

alternatives

Une liste de choix possibles parmi lesquels on ne peut en sélectionner qu'un, comme dans : « Préférez-vous la porte A, B ou C ? » Les termes des alternatives dans les expressions régulières sont séparés par une barre verticale : `|`. Les termes des alternatives dans les expressions Perl normales sont séparés par deux barres verticales : `||`. On peut parler d'une alternative d'alternatives. Ou non. À vous de choisir. Les alternatives logiques dans les expressions booléennes sont séparées par `||` ou bien `or`.

analyse lexicale

Terme amusant pour dire *tokeniseur*.

analyse syntaxique

L'art subtil et parfois brutal pour tenter de tranformer votre programme avec ses erreurs potentielles en *arbre syntaxique*.

anonyme

Qualifie un *référent* qui ne peut être accédé par une *variable*, mais indirectement par une *référence dure*, qui disparaît après utilisation.

appel par référence

Un mécanisme de passage d'*arguments* par lequel les *arguments formels* se réfèrent directement aux *arguments réels*, le *sous-programme* pouvant modifier les arguments réels en changeant les arguments

formels. Les arguments formels sont des alias pour les arguments réels. Voir aussi *appel par valeur*.

appel par valeur

Un mécanisme de passage d'*arguments* par lequel les *arguments formels* se réfèrent à une copie des *arguments réels*, le *sous-programme* ne pouvant modifier les arguments réels par les arguments formels. Voir aussi *appel par référence*.

appel système

Un appel direct de *sous-programme* au *système d'exploitation*. Nombre des fonctions et des sous-programmes utilisés ne sont pas des appels systèmes directs, mais constituent des surcouches au-dessus du niveau de l'appel système. En général, les utilisateurs de Perl n'ont pas à se soucier de la distinction. Néanmoins, si vous connaissez les fonctions Perl qui font des appels systèmes, vous savez celles qui affecteront la variable `$!` (`$ERRNO`) lors d'un échec. Malheureusement, les programmeurs Perl emploient souvent le terme « appel système » lorsqu'ils utilisent la fonction `system`, laquelle met en jeu beaucoup d'*appels systèmes*. Pour éviter tout confusion, nous utilisons presque toujours le terme « appel système » pour désigner l'appel indirect à l'aide de la fonction Perl `syscall`, mais jamais pour indiquer un appel de la fonction Perl `system`.

appelant

L'agent responsable de l'appel d'une *méthode*. Dans une méthode de *classe*, l'appelant est un nom de paquetage. Dans une méthode d'*instance* l'appelant est une référence d'objet.

approche boîte à outils

La notion selon laquelle, avec un ensemble complet d'outils simples qui fonctionnent bien les uns avec les autres, il est possible de réaliser ce que l'on veut. Ce qui convient si l'on assemble un tricycle, mais dans le cas d'un microsthéseur rhomboïdal à double inversion de flux, il faut absolument qu'un atelier construise les outils adaptés. Perl est une sorte d'atelier.

arbre d'analyse

Voir *arbre syntaxique*.

arbre syntaxique

Une représentation interne de votre programme dans laquelle les *constructions* de bas-niveau sont incluses dans les blocs contenant.

architecture

Le type d'ordinateur sur lequel on travaille, « type » signifiant que les ordinateurs en faisant partie peuvent faire tourner le même programme *binaire*. Puisque les scripts Perl sont des fichiers texte et non binaires, un script Perl est beaucoup moins sensible à l'architecture sur laquelle il tourne que les programmes d'autres langages (comme C) qui sont compilés en code machine. Voir aussi *système d'exploitation*.

argument

Des données fournies en entrée d'un *programme*, d'un *sous-programme*, d'une *fonction* ou d'une *méthode* servant à lui dire quoi faire. Également appelé « paramètre ».

arguments de la ligne de commande

Les *valeurs* fournies comme paramètres d'un programme dont l'exécution est demandée au *shell*. Ces valeurs sont accessibles en Perl par l'intermédiaire du tableau `@ARGV`.

arguments formels

Nom générique des arguments d'un *sous-programme*. En Perl, les arguments formels d'un programme sont dans un tableau : `$ARGV[0]`, `$ARGV[1]`, ainsi de suite. Ceux d'un sous-programme sont `$_[0]`, `$_[1]`, etc, arguments qu'on peut tranformer localement en noms par l'affectation à une liste `my`. Voir aussi *arguments réels*.

arguments réels

Les *valeurs scalaires* que l'on fournit à une *fonction* ou à un *sous-programme* que l'on appelle. Par exemple, quand on appelle `jeton("bingo")`, la chaîne `"bingo"` est l'argument réel. Voir également *argument* et *arguments formels*.

ARGV

Nom du tableau qui contient le *vecteur d'argument* de la ligne de commande. Avec l'opérateur `<>`, `ARGV` est à la fois le nom du *filehandle* qui utilisera les arguments et le *scalaire* contenant le nom du fichier en entrée.

ASCII

« American Standard Code for Information Interchange » (un ensemble de caractère efficace pour la représentation imparfaite des textes anglais). Utilisé aussi pour désigner les 128 premières valeurs des codages ISO-8859-X, un groupe de codes 8 bits incompatibles entre eux.

assertion

Composant d'une *expression* régulière qui doit être vraie pour le motif recherché, mais qui ne correspond pas forcément avec un caractère. Souvent utilisé pour désigner une assertion de *longueur nulle*, c'est-à-dire évaluée sur une chaîne nulle.

associativité

Détermine lequel des *opérateurs* de droite ou de gauche est exécuté en premier, quand on a « A *opérateur* B *opérateur* C » , si les deux opérateurs ont la même précédence. Les opérateurs comme + sont associatifs à gauche, alors que les opérateurs comme ** sont associatifs à droite. Le chapitre 3, *Opérateurs unaires et binaires*, donne la liste des associativités.

asynchrone

Événements ou activités dont l'occurrence relative est indéterminée, car trop nombreux à se produire dans la même unité temporelle. On peut dire que c'est un événement qui arrivera à l'improviste.

atome

Composant d'une *expression régulière* qui peut correspondre à une *sous-chaîne* qui contient un ou plusieurs caractères et qui est traitée comme une unité syntaxique par tout *quantificateur* qui la suit. À comparer avec l'*assertion* qui s'évalue sur une chaîne de *longueur nulle* et ne peut être quantifiée.

attribut

Nouvelle fonctionnalité qui permet d'ajouter des modificateurs à des déclarations de *variable* ou *fonction* avec modificateurs comme dans `sub foo : locked method`. C'est aussi l'autre nom d'une *variable d'instance* d'un *objet*.

autogénération

Fonctionnalité de l'*opérateur de surcharge* des *objets*, où le comportement de certains *opérateurs* peut être dérivé des opérateurs fondamentaux. Cela suppose bien entendu que les opérateurs surchargés ont les mêmes relations que les opérateurs ordinaires. Voir le chapitre 13, *Surcharge*.

auto-incrémentation

Ajouter automatiquement un à quelque chose. Généralement employé pour décrire l'opérateur ++. L'opération inverse, soustraire 1, est appelé « autodécrémentation ».

autochargement

Chargement à la demande (également appelé « chargement paresseux »), précisément appel d'une fonction `AUTOLOAD` en lieu et place d'un routine indéfinie.

autosplit

Découpage automatique d'une chaîne à la manière du *sélecteur* **`-a`** utilisé avec **`-p`** ou **`-n`** pour émuler la commande *awk*. Voir aussi le module `AutoSplit` (qui n'a rien à voir avec l'autosplit, mais beaucoup à voir avec l'autochargement).

autovivification

Terme d'origine gréco-latine signifiant littéralement « se donner la vie ». En Perl, les places mémoires (ou *lvalues*) se créent suivant le besoin, y compris la création des *références dures* nécessaires pour adresser le niveau suivant de stockage. L'affectation `$a[5][5][5][5][5] = "quintet"` crée cinq scalaires, plus quatre références (dans les quatre premiers scalaires) pointant sur quatre tableaux anonymes. Mais ici le point important est que nous n'avons pas à nous en soucier.

AV

Abréviation de « array value », qui se réfère à l'un des types de données interne de Perl. Une AV est une sorte de `SV`.

avertissement

Un message affiché sur le flux `STDERR` indiquant que quelque chose ne va pas mais que ce n'est pas une raison pour tout casser. Voir `warn` dans le chapitre 29, *Fonctions*, ainsi que le pragma `use warnings` dans le chapitre 31, *Modules de pragmas*.

avide

Un *sous-motif* dont le *quantificateur* cherche à effectuer une correspondance avec le maximum de choses.

awk

Abréviation du mot « étrange ». Aussi un vénérable langage de traitement de textes auquel Perl a emprunté quelques idées de haut niveau.

backtracking

En théorie cela revient à dire « si c'était à refaire, je le referais autrement », et en pratique avec Perl c'est le refaire vraiment. Mathématiquement parlant, il s'agit du retour d'une récursion ayant échoué sur une arborescence de possibilités. Le backtracking, ou retour arrière, se produit en Perl quand il tente de comparer des motifs avec une *expression régulière*, et que ses premiers essais ne correspondent pas. Voir *« Le petit Moteur qui /(ne)?pouvait(pas)?/ »* au chapitre 5, *Recherche de motif*.

bibliothèque

Une collection de procédures. Aux temps héroïques, se référait à une collection de sous-programmes dans un fichier *.pl*. Actuellement, se réfère souvent à l'intégralité des *modules* Perl d'un système.

binaire

Ayant à voir avec les nombres représentés en base deux. Cela veut dire qu'il existe deux nombres, zéro et un. Désigne également un « fichier non-texte », dont le codage a été effectué avec les 8 bits de l'octet. Avec l'arrivée d'*UNICODE*, cette distinction perd toute signification.

bit

Entier dans l'intervalle 0-1, bornes comprises. La plus petite unité de stockage d'information. Un huitième d'*octet* ou de *dollar*. (L'expression *Pieces of eight*, pièces de huit, vient de la possibilité de diviser les vieux dollars espagnols en 8 bits, chacun restant de l'argent valable. C'est là d'où vient l'expression *twobits* pour une pièce de 25 cents.

bits d'autorisation

Les bits utilisés par le *propriétaire* d'un fichier pour limiter les accès à celui-ci. Ces bits font partie du mot *mode* retourné par la fonction `stat` prédéfinie lorsque vous demandez l'état d'un fichier. Sur les systèmes Unix, vous pouvez regarder la page de manuel *ls*(1) pour plus d'information.

bit d'exécution

La marque spéciale qui indique au *système d'exploitation* qu'il peut lancer ce programme. Il existe en fait trois bits d'exécution sous UNIX, et le bit utilisé dépend de ce que l'on possède le fichier individuellement, collectivement, ou pas du tout.

bloc

Un gros morceau de données, d'une taille que le système d'exploitation apprécie (d'habitude une puissance de 2 comme 512 ou 8192). Cela désigne habituellement un morceau de données venant d'un fichier ou y allant.

BLOCK

Une construction syntaxique composée d'une séquence d'instructions Perl délimitées par des accolades. Nous parlons parfois de « bloc » pour une séquence d'instructions qui agissent comme un *BLOC*, même non délimitée par des accolades.

bloqué

État d'un *processus* qui attend une ressource : « ce processus est bloqué car il attend une ressource du disque ».

Booléen

Valeur *vraie* ou *fausse*.

boucle

Une construction qui effectue quelque chose de façon répétée, comme un surfeur sur des rouleaux.

brouillon

C'est une zone mémoire temporaire dans laquelle un sous-programme ou un fichier place temporairement ses valeurs et aussi toute variable à portée lexicale.

BSD

Une drogue psychotrope, populaire dans les années 80, probablement développée à l'U.C. Berkeley ou ses environs. Similaire par de nombreux points à la médication appelée « System V », mais infiniment plus utile (ou du moins, plus amusante). Le nom chimique complet est « Berkeley Standard Distribution » .

bucket

Espace dans une *table de hachage* contenant plusieurs entrées dont les clés ont une valeur de hachage identique. (Cette notion est interne à la gestion des tableaux faite par Perl et ne vous concerne pas, en règle générale.)

bytecode

Un baragouin employé par les androïdes quand ils ne veulent pas révéler leur orientation (voir *boutismes*). Ainsi dénommé d'après certains langages similaires parlés (pour des raisons similaires) entre les compilateurs et les interpréteurs à la fin du vingtième siècle. Ces langages sont caractérisés par leur façon de représenter par une séquence d'octets indépendante de l'architecture.

C

Un langage chéri par de nombreuses personnes pour ses définitions de types inversées, ses règles de *précédence* impénétrables et une *surcharge* importante de ses mécanismes d'appel de fonctions. (En fait, les gens sont d'abord passés au C parce qu'ils trouvaient les identificateurs en minuscules plus faciles à lire que ceux en majuscules.) L'*interpréteur* Perl est écrit en C, et il n'est donc pas surprenant que Perl lui ait emprunté quelques idées.

canal de sortie actuellement sélectionné

Le dernier *handle de fichier* ayant été désigné par `select (FILEHANDLE)` ; `STDOUT` par défaut, si aucun handle n'a été sélectionné.

canonique

Réduction dans une forme standard qui facilite la comparaison.

capture

L'utilisation de parenthèses autour d'un *sous-motif* dans une *expression régulière* pour mémoriser la *sous-chaîne* correspondante comme *référence arrière*. (Les chaînes capturées peuvent aussi être retournées dans un *contexte de liste*.)

caractère

Un petit entier qui code une unité d'écriture. Historiquement, les caractères sont encodés dans un entier de longueur fixe (typiquement un octet, voir deux selon le type à coder), mais avec l'apparition du codage UTF-8, ils sont codés sur une longueur variable. Perl gère ceci d'une manière transparente, en général.

caractère espace

Un *caractère* qui déplace le curseur mais n'affiche rien à l'écran. Se réfère typiquement aux caractères suivants : blanc, tabulation, retour chariot, saut de ligne, saut de page.

caractère nul

Un caractère ASCII de valeur zéro. Il est utilisé par C et certains appels système UNIX pour terminer les chaînes, alors que Perl autorise les chaînes à contenir un nul.

chaîne

Une séquence de caractères telle que « il dit !@#*&%@#*?! ». Une chaîne n'est pas obligatoirement imprimable.

chaîne binaire

Séquence de *bits* interprétée comme un séquence de bits.

chaîne nulle

Une *chaîne* ne contenant aucun caractère, à ne pas confondre avec une chaîne contenant un *caractère nul*, qui a une longueur positive et une valeur *vraie*.

champ

Un bout simple de données faisant partie d'une *chaîne*, d'un *enregistrement* ou d'une *ligne*. Les champs de longueur variable sont généralement séparés par des *délimiteurs* (on utilise `split` pour extraire les champs) alors que les champs de longueur fixe se trouvent à des positions constantes (on utilise `unpack`). Les *variables d'instances* sont aussi appelées « champs ».

chemin

Un nom de fichier entièrement qualifié tel */usr/bin/perl*. Parfois confondu avec `PATH`.

classe

Un *type* prédéfini par l'utilisateur implémenté en Perl par la déclaration *package*, paquetage qui fournit (directement ou par héritage) les *méthodes* (c'est-à-dire les *sous-routines*) pour gérer les *instances* de la classe (ses *objets*). Voir aussi l'*héritage*.

classe de base

Type d'*objet* générique dont la *classe* très abstraite peut être dérivée par *héritage* vers des classes plus spécialisées. Aussi appelé « superclasse » dans la tradition de nos ancêtres.

classe de caractères

Une liste de caractères entre crochets employée dans une *expression régulière* pour indiquer que tout caractère de l'ensemble peut se trouver à cet endroit. Aussi tout ensemble de caractères prédéfinis.

classe dérivée

Une *classe* qui définit certaines de ses *méthodes* dans les termes d'une classe plus générique, appelée *classe de base*. Remarquez que les classes ne sont par classifiées exclusivement en classes de base ou classes dérivées ; une classe peut simultanément fonctionner des deux manières, ce qui est typique.

classe parent

Voir *classe de base*.

clef

L'index de type chaîne d'un *hachage*, utilisé pour chercher une *valeur* associée à une certaine clef.

client

Dans un contexte de réseau, un *processus* qui prend contact avec un processus *serveur* pour échanger des données et éventuellement recevoir un service.

cloisonneur

Un *cluster* utilisé pour restreindre la portée d'un *modificateur d'expression rationnelle*.

cluster

Un *sous-motif* entre parenthèses utilisé pour grouper les parties d'une *expression rationnelle* dans un seul *atome*.

CODE

Mot retourné par la fonction `ref` appliquée à la référence d'une sous-routine. Voir aussi *CV*.

commande

En programmation *shell*, la combinaison syntaxique d'un nom de programme avec ses arguments. Plus généralement, tout ce que l'on entre dans un shell (un interpréteur de commandes) qui fait quelque chose. En programmation Perl, une *instruction*, qui peut démarrer par une *étiquette* et se termine typiquement par un point-virgule.

commentaire

Une remarque qui n'affecte pas la signification du programme. En Perl, un commentaire commence par un caractère # et continue jusqu'à la fin de la ligne.

compatibilité ascendante

Possibilité de faire tourner ses anciens fichiers sources sur les nouvelles versions de l'interpréteur, car nous n'avons cassé ni les fonctionalités, ni les bugs sur lesquels reposent les anciens programmes.

compilateur

Au sens strict, un programme qui en absorbe un autre et recrache un troisième fichier contenant ce programme sous une forme plus exécutable, contenant typiquement des instructions en langage machine

natif. Le programme *perl* n'est pas un compilateur selon cette définition, mais il en contient un qui prend un programme et le métamorphose sous une forme plus exécutable (*arbres syntaxiques*) à l'intérieur du processus *perl* lui-même, que l'*interpréteur* interprète ensuite. Il existe néanmoins des *modules* d'extension qui permettent à Perl d'agir plus comme un vrai compilateur. Voir le chapitre 18, *Compilation*.

compilation

Le moment où Perl essaye de comprendre ce que signifie le programme, par opposition à celui où, croyant savoir ce qu'il signifie, il essaye de faire ce que le programme lui dit de faire. Voir *exécution*.

compositeur

Un « constructeur » pour un *référent* qui n'est pas vraiment un *objet*, comme un tableau ou un hachage. Par exemple, une paire d'accolades agissent comme compositeur de hachage, et une paire de crochets agissent comme compositeur de tableau. Voir *Création de références* au chapitre 8, *Références*.

concaténation

Coller le museau d'un chat à la queue d'un autre. On peut remplacer « chat » par « chaîne ».

condition

Avec des « si ». Voir *contexte booléen*.

connexion

En téléphonie, le circuit électrique temporaire entre les téléphones de l'appelant et de l'appelé. En réseau, le même genre de circuit temporaire entre un *client* et un *serveur*.

consacrer

Perl va classer un *référent* pour qu'il fonctionne comme un *objet*. Voir la fonction `bless` dans le chapitre 29.

constructeur

Toute *méthode de classe*, *méthode d'instance* ou *sous-programme* qui crée, initialise, consacre et retourne un *objet*. Le terme est quelquefois utilisé pour désigner un *compositeur*.

construction

Un ensemble syntaxique composé d'éléments plus petits. Également l'action de créer un *objet* grâce à un *constructeur*.

construire

Créer un objet avec un constructeur.

contexte

L'environnement. Le contexte donné par le code environnant détermine le genre de données qu'une *expression* doit renvoyer. Les trois contextes primordiaux sont le *contexte de liste*, le *contexte scalaire* et le contexte *peu importe*. Le contexte scalaire est parfois subdivisé en *contexte booléen*, *contexte numérique* et *contexte de chaîne*. Il existe aussi un contexte « peu importe » (qui est traité au chapitre 2, *Composants de Perl*, si cela vous importe).

contexte booléen

Un *contexte scalaire* particulier utilisé dans les conditions pour évaluer une *valeur scalaire* à *vrai* ou *faux*. À comparer à l'évaluation d'une *chaîne* ou d'un *nombre*. Voir *contexte*.

contexte de chaîne

La situation dans laquelle l'environnement (le code appelant) attend d'une expression qu'elle renvoie une *chaîne*. Voir aussi *contexte* et *contexte numérique*.

contexte de liste

La situation dans laquelle un environnement (le code appelant) attend d'une *expression* qu'elle renvoie une liste de valeurs au lieu d'une valeur unique. Les fonctions qui désirent une `LIST` d'arguments indiquent à ses arguments qu'ils doivent produire une liste de valeurs. Voir également *contexte*.

contexte de tableau

Ancienne expression pour dire *contexte de liste*.

contexte numérique

La situation dans laquelle l'environnement (le code appelant) attend d'une expression qu'elle renvoie un nombre. Voir aussi *contexte* et *contexte de chaîne*.

contexte scalaire

La situation dans laquelle l'environnement (le code appelant) attend d'une *expression* qu'elle renvoie une *valeur* unique au lieu d'une *liste*. Voir également *contexte* et *contexte de liste*. Un contexte scalaire impose parfois des contraintes supplémentaires sur la valeur de retour, voir *contexte de chaîne* et *contexte numérique*. Nous parlons parfois d'un *contexte booléen* dans les conditions, mais cela n'implique aucune contrainte supplémentaire puisque toute valeur scalaire, qu'elle soit numérique ou *chaîne*, est déjà vraie ou fausse.

contexte vide

Une forme de *contexte scalaire* dans lequel une *expression* ne retourne aucune *valeur* et qui n'est utile que pour ses propres *effets de bord*.

continuation

Le traitement de plusieurs *lignes* physiques comme une seule ligne logique. Les lignes de *Makefile* sont continuées en mettant un antislash avant le *saut de ligne*. Les en-têtes de messages Internet sont continués en mettant un espace ou une tabulation *après* le saut de ligne. Les lignes Perl n'ont pas besoin d'un quelconque forme de marque de continuation, car les *espaces* (y compris les sauts de ligne) sont joyeusement ignorés. En général.

Le cadavre d'un *processus*, sous la forme d'un fichier laissé dans le *répertoire de travail* du processus, généralement résultant de certaines formes d'erreur fatale.

conversion en chaîne

La production d'une *chaîne* pour représenter un objet.

correspondance

Voir *recherche de correspondance*.

CPAN

« Comprehensive Perl Archive Network » ou Réseau détaillé des Archives de Perl. Voir la préface et le chapitre 22, *CPAN*, pour les détails.

cracker

Une personne qui viole la sécurité des systèmes d'exploitation. Un cracker peut-être un vrai *hacker* ou simplement un *script kiddie*.

CV

Une définition de type de « valeur de code » (code value). Un CV est une sorte de *SV*.

datagramme

Un paquet de données, tel un message *UDP*, qui est transmis indépendamment de la couche applicative sur le réseau (ce sont de paquets *IP* encapsulés dans un protocole *TCP* et donc invisible depuis votre programme.)

DBM

Signifie « Data Base Management » , un ensemble de routines qui émulent un *tableau associatif* avec des fichiers sur disque. Les routines emploient un système de hachage dynamique pour localiser n'importe quelle entrée en seulement deux accès disque. Les fichiers DBM permettent à un script Perl de conserver un *hachage* entre plusieurs invocations. Il est possible de lier par `tie` les variables hachage à diverses implémentations DBM ; voir *AnyDBM_File*(3) et l'entrée *DB_File* au chapitre 32, *Modules standards*.

débogueur symbolique

Un programme permettant l'*exécution* pas à pas d'un programme, en arrêtant pour afficher des choses ici et là pour voir si quelque chose a mal tourné. L'adjectif « symbolique » signifie que l'on peut discuter avec le débogueur en employant les mêmes symboles que ceux qui ont été utilisés pour écrire le programme.

décalage

Décalage à droite ou à gauche dans un mot de la mémoire, dont l'effet est de multiplier ou de diviser par une puissance de 2.

décalage à droite

Un *décalage de bit* qui divise un nombre par une puissance de 2.

décalage à gauche

Un *décalage de bits* qui multiplie un nombre par une puissance de deux.

déclaration

Une *assertion* de l'existence de quelque chose, et éventuellement de ce à quoi cela ressemble, sans présumer de ce qui va lui arriver, ni de quelle façon. Une déclaration ressemble à la partie d'une recette énumérant : « deux tasses de farine, un œuf, quatre ou cinq beaux têtards... » Voir son opposé *instruction*. Remarquez que certaines déclarations fonctionnent également comme des instructions. Les déclarations de sous-routines forment aussi une définition si le corps est fourni.

déclencheur

Un événement qui déclenche un *handler*.

décrémenter

Soustraire une valeur d'une variable comme dans « decrement `$x` » qui soustrait 1 ou « décrémenter `$x` de 3 ».

défaut (par)

Une *valeur* qui est choisie pour vous si vous n'en fournissez pas une vous-même.

défini

Possédant une signification. Perl pense que certaines des choses que les gens font n'ont aucune signification : en particulier l'utilisation d'une variable à qui n'a jamais été attribuée de valeur, et effectuer des opérations sur des données qui n'existent pas. Par exemple, si l'on essaye de lire des données après la fin d'un fichier, Perl renvoie une *valeur indéfinie*. Voir également *faux* et l'opérateur `defined` au chapitre 29.

délimiter

Un *caractère* ou *une chaîne* qui délimite un objet textuel de taille indéfinie que l'on ne doit pas confondre avec le *séparateur* ou *terminateur*. « Délimiter » veut simplement dire « entourer » ou « clôturer » (comme ces parenthèses le font).

déplacement

Ou offset : le nombre d'éléments à sauter après le début d'une chaîne ou d'un tableau pour arriver à une position spécifique. L'offset minimal est donc zéro et non un, car il n'y à rien à sauter pour arriver au premier élément.

déréférencer

Un joli terme informatique signifiant « suivre une *référence* vers ce sur quoi elle pointe ». Le préfixe « dé » indique qu'il s'agit d'enlever un niveau d'*indirection*.

descripteur

Voir *descripteur de fichier*.

descripteur de fichier

Le petit numéro qu'utilise le *système d'exploitation* pour garder la trace du *fichier* ouvert dont on s'occupe. Perl dissimule le descripteur de fichier dans un *flux d'E/S standard*, puis attache le flux à un *handle de fichier*.

destruction globale

Le *ramasse-miettes* des globales (et l'action de ces destructeurs) qui se produit quand l'*interpréteur* Perl est arrêté.

destructeur

Une *méthode* spéciale qui est appelée quand un *objet* le demande. La méthode `DESTROY` n'effectue pas la destruction à proprement parler. Elle donne l'occasion à la *classe* de faire un peu de ménage.

détruire

Désallouer la mémoire d'un *référent* (mais en appelant d'abord sa méthode `DESTROY`, si elle existe.)

diffuser

Envoyer un datagramme à plusieurs destinataires simultanément.

directive

Une directive *pod*. Voir le chapitre 26, *POD*.

distribuer

Envoyer une chose à sa destination correcte. Métaphore utilisée pour désigner le transfert d'un programme vers un autre par sélection dans une table de fonction de *références* ou bien comme dans le cas des *méthodes* d'objets, par le parcours d'un arbre d'héritage de classes.

distribution

Une version stable d'un système ou d'un logiciel. Par défaut, le code source est

inclus. Dans le cas inverse, elle est dite « binary-only ».

document ici-même

Ainsi appelé en raison d'une construction similaire des *shells* qui prétendent que les lignes suivant « ici même » la *commande* sont un *fichier* séparé à donner à la commande, jusqu'à une chaîne servant de *délimiteur* de fin. En Perl, il ne s'agit que d'une forme élaborée de protection.

domaine public

Qui n'est possédé par personne. Perl fait l'objet d'un copyright, et n'est donc *pas* dans le domaine public ; il n'est que *librement disponible* et *librement redistribuable*.

drapeau

Nous évitons l'utilisation de ce terme qui veut tout dire : un *sélecteur* de la ligne de commande qui ne prend pas d'argument (tels **-n** et **-p**), un simple bit indicateur (tels `O_CREAT` et `O_EXCL` utilisés dans `sysopen`).

drôle de caractère

Quelqu'un comme Larry, ou les types qui lui servent d'amis. Ou bien, l'un de ces étranges préfixes devant les noms de variables en Perl.

dweomer

Un enchantement, une illusion, un phantasme, ou une acrobatie. Se dit d'un programme Perl qui ne fait pas ce que vous attendez de lui, mais plutôt quelque chose de l'ordre de la sorcellerie.

dwimmer

DWIN est un acronyme pour « Do What I Mean », principe selon lequel une instruction devrait faire ce que vous désirez, sans faire de chichis. Un morceau de code qui fait du « dwimming » est un « dwimmer ». Dwimming nécessite souvent beaucoup de travail en arrière plan et si ça se voit ça devient du *dweomer*.

éclectique

Dérivé de nombreuses sources. Certains diraient trop nombreuses.

édition de lien

Résolution de noms dans un fichier partiellement compilé afin de produire une image (presque) exécutable. L'édition de liens est statique ou dynamique (à ne pas confondre avec la portée statique ou dynamique).

effets de bord

Un effet secondaire qui se produit lors de l'évaluation d'une *expression*. Par exemple l'évaluation d'une simple affectation a pour « effet de bord » d'affecter une valeur à une variable (ce qui était bien votre intention première!). De même, l'affectation d'une valeur à la variable spéciale `$|` (`$AUTOFLUSH`) a pour effet de bord de forcer un vidage après chaque `write` ou chaque `print` sur le handle de fichier courant.

élément

Une brique de base. Quand on parle d'un *tableau*, l'un des items qui le composent.

emplacement d'objet indirect

Position syntaxique entre l'appel d'une méthode et ses arguments quand la notation indirecte est invoquée. La position se distingue des arguments par l'absence de virgule entre elle et l'argument suivant. `STDERR` est dans un emplacement indirect ici :

```
print STDERR "Debout!
  Réveillez-vous!  Peur, Feu, Folie!
  Debout!\n";
```

en passant

Quand on change la *valeur* d'un objet au moment de sa copie (de l'expression, prise du pion en passant aux échecs).

encapsulation

La couche abstraite qui sépare l'*interface* de l'*implémentation*, et qui impose que tout accès à l'état d'un *objet* soit effectué au travers de *méthodes* exclusivement.

enregistrement

Ensemble de données reliées entre elles dasn un *fichier* ou un *flux*, souvent associées avec une *clef* unique. Sous Unix, elles sont souvent rassemblées sur une *ligne* ou un dans un « paragraphe » (suite de lignes

terminée par une ligne vide). Chaque ligne du fichier */etc/passwd* est un enregistrement dont la clef est le login, et qui contient l'information de l'utilisateur.

enrobage

Un programme qui en lance un autre à votre place, en modifiant certaines des entrées ou des sorties pour mieux vous convenir.

entier

Un nombre sans partie fractionnaire. Un nombre pour dénombrer comme 1, 2, 3..., en incluant 0 et les nombres négatifs.

entrée standard

Le flux d'entrée par défaut d'un programme, qui doit si possible ne pas s'occuper du lieu de provenance des données. Représenté dans un programme Perl par le *handle de fichier* `STDIN`.

environnement

L'ensemble des *variables d'environnement* que le *processus* hérite de son parent. Accessible *via* `%ENV`.

EOF

Fin de fichier. Métaphore pour désigner la chaîne indiquant la fin d'un *fichier ici-même*

erreur

Voir *exception* ou *erreur fatale*.

erreur fatale

Une *exception* non traitée, qui provoque la fin du *processus* après impression d'un message sur la console d'*erreur standard*. Les erreurs dans un bloc `eval` ne sont pas fatales : le bloc se termine après avoir placé le message d'exception dans la variable `$@` (`$EVAL_ERROR`). On peut lever explicitement une exception avec l'opérateur `die` qui peut être traitée dynamiquement par le bloc `eval`. S'il ne l'est pas, cela provoque une erreur fatale.

erreur standard

Le *flux de sortie* par défaut qui sert à faire des remarques désobligeantes qui ne sont pas destinées à la *sortie standard*. Représenté en Perl par le *handle de fichier* `STDERR`. Ce flux peut être utilisé explicitement, mais les opérateurs `die` et `warn` le font automatiquement.

errno

Le numéro de l'erreur renvoyé par un *appel système* UNIX quand il échoue. Perl se réfère à l'erreur par le nom `$!` (ou `$OS_ERROR` si l'on utilise le module English).

E/S standard

Une bibliothèque standard du C (« standard I/O ») servant aux entrées et aux sorties i vers le *système d'exploitation* (le « standard » des E/S standard n'a qu'un rapport marginal avec le « standard » des entrées et sorties standard). En général, Perl se base sur l'implémentation propre à chaque système d'exploitation, et les caractéristiques de tamponnage peuvent donc ne pas exactement correspondre d'une machine à l'autre. Cela n'influence normalement que l'efficacité et non la portabilité. Si les fonctions d'E/S standard tamponnent par bloc et qu'il serait souhaitable de *vider* les tampons plus souvent, il suffit de mettre la variable `$|` à une valeur vraie.

espace de noms

Un domaine de noms. Il n'y a pas besoin de se soucier de savoir si les noms d'un tel domaine ont été utilisés dans d'autres. Voir *paquetage*.

est une sorte de

Relation entre deux *objets* dans laquelle l'un est considéré comme une version plus spécifique de l'autre objet générique : « Un chameau est un mammifère ». L'objet générique étant une abstraction au sens platonique, on parle plutôt de relation entre une *classe de base* et une classe spécifique dite *classe dérivée*. Étrangement, les classes abstraites platoniques n'ont pas toujours des relations platoniques — voir la notion d'*héritage*.

étiquette

Nom donné à une *instruction* pour pouvoir y accéder partout ailleurs dans le programme.

étiquette de boucle

Un genre de clef ou de nom rattaché à une boucle pour que les instructions de contrôle de boucle puissent savoir quelle boucle elles doivent contrôler.

exception

Un joli nom pour une erreur. Voir *erreur fatale*.

exec

Stopper le programme du *processus* en cours pour le remplacer par un autre, tout en gardant les ressources en cours dans le processus (sauf son image mémoire).

exécuter

Exécution d'un *programme* ou *sous-programme*. (Rien à voir avec la fonction « tuer » `kill`, sauf si vous mettez en œuvre un *handler de signal*.)

exécution

Le laps de temps pendant lequel Perl essaye de faire ce que le script lui ordonne, par opposition à la période précédente pendant laquelle Perl essayait de savoir si le script avait un sens quelconque à l'exécution. Voir également *compilation*.

exporter

Rendre les symboles d'un *module* disponibles depuis un autre par la directive *import*.

expression

Tout assemblage de *littéraux*, *variables*, *opérateurs*, *fonctions* ou *sous-programmes*, effectué suivant les règles, quand on a besoin d'une valeur.

expression alerte

Une expression qui provoque un arrêt lorsque sa valeur change, ce qui permet de déboguer.

expression régulière

Une entité unique avec diverses interprétations, comme un éléphant. Pour un informaticien, il s'agit de la grammaire d'un petit langage dans lequel certaines chaînes sont légales et d'autres non. Pour les gens normaux, c'est un motif que l'on peut utiliser pour trouver ce dont on a besoin quand la cible varie selon les cas. Les expressions régulières en Perl sont loin d'être régulières au sens mathématique du terme, mais fonctionnent plutôt bien lorqu'on les utilise correctement. Exemple d'une expression régulière : `/Oh, m.*d../`. Ce motif correspond à des chaînes comme « Oh, mais vous étiez dans le train allant à Lourdes » et « Oh, ma foi, il se fait tard ! ». Voir le chapitre 5.

extension

Un module Perl qui comporte du code C ou C++ compilé. Ou plus généralement, une option expérimentale qui peut être compilée et intégrée dans Perl, comme le multithreading.

false

En Perl, toute valeur qui ressemble à `""` ou `"0"` dans un contexte de chaîne. Tout valeur indéfinie est donc faussse, car évaluée à `""`, mais toute valeur fausse n'est pas forcément indéfinie.

FAQ

Questions fréquemment posées (*Frequently Asked Questions*, ou Foire aux Questions) (même les réponses n'y sont pas forcément, surtout dans les FAQ de la version courante qui peuvent dater).

fermeture

Un sous-programme *anonyme* qui, généré à l'exécution, garde trace des identités des *variables lexicales* visibles de l'extérieur même après que ces variables lexicales sont devenues hors de *portée*. Elles sont appelées « fermetures » parce que ce genre de comportement donne aux mathématiciens un sentiment d'isolement.

fichier

Une collection de données nommée, généralement stockée sur disque dans un *répertoire* lui-même dans un *système de fichiers*. Ressemblant vaguement à un document, si on apprécie les métaphores d'ordre bureautique. Sous quelques *systèmes d'exploitation* comme UNIX, un même fichier peut en fait répondre à plusieurs noms. Certains fichiers sont spéciaux

comme les répertoires et les périphériques.

fichiers d'en tête

Un fichier contenant certaines définitions requises qui doivent être incluses « en-tête » pour pratiquer certaines opérations obscures. Un fichier d'en-tête C possède une extension *.h*. Un fichier d'en-tête Perl possède une extension *.ph*. Voir l'opérateur `require` au chapitre 29. (Les fichiers d'en-tête ont été supplantés par le mécanisme de *module*.)

fichier exécutable

Un *fichier* marqué, reconnu par le *système d'exploitation* comme exécutable. Simplement appelé « exécutable ».

fichier normal

Un *fichier* qui n'est pas un *répertoire*, un *périphérique*, un *tube* nommé, une *socket* ou un *lien symbolique*. Perl utilise l'*opérateur de test de fichier* `-f` pour identifier les fichiers « normaux ».

FIFO

Premier entré, premier sorti, file d'attente. Voir aussi *LIFO*. Aussi l'autre nom pour un *tube nommé*.

fileglob

Une correspondance par « joker » sur des *noms de fichier*. Voir la fonction `glob`.

filtre

Transforme un *flux* d'entrée en flux de sortie.

filtre source

Un *module* spécial qui opère un *prétraitement* sur votre script juste avant l'*analyseur lexical*.

flux

Un flux continu de données de type caractères ou octets entrant ou sortant d'un processus sans donner l'impression d'arriver paquet par paquet. C'est une sorte d'*interface* — l'implémentation sous-jacente est invisible, et peut fort bien grouper les données par paquets.

FMTEYEWTK

Plus que vous n'avez jamais espéré savoir. Traité exhaustif sur un sujet pointu, quelque chose comme une super *FAQ*. Voir Tom pour en savoir beaucoup plus.

fonction

Mathématiquement, une relation faisant correspondre un ensemble de valeurs de départ vers un ensemble de valeurs d'arrivée. Dans le langage des ordinateurs, se réfère à un *sous-programme* ou un opérateur, avec ou sans valeurs d'entrée (appelées *arguments*) qui retourne une *valeur*.

fonction de rappel

Un *handler* déclaré dans une autre partie du programme et qui peut être *déclenché* par un code différent lorsqu'un événement se produit.

fork

Créer un *processus* fils identique au processus père, du moins jusqu'à ce qu'il se fasse ses propres idées. Un thread avec sa mémoire locale.

format

Spécification indiquant combien d'espaces, de chiffres et de choses diverses il faut pour que tout ce que vous imprimez donne un résultat agréable à regarder.

freeware

Logiciel mis à libre disposition, parfois avec son code source. On l'appelle maintenant plutôt *logiciel à source ouvert*. Récemment, une tendance est apparue qui oppose les freewares aux logiciels à source ouvert, et qui réserve le terme freeware aux logiciels diffusés sous la licence GPL de la « Free Software Foundation ».

générateur de code

Système qui produit du code machine tel le code final d'un compilateur. Voir *générateur de programme*.

générateur de programme

Un système qui écrit du code pour vous dans un langage de haut niveau. Voir aussi *générateur de code*.

gestionnaire de signal

Un *sous-programme* qui, au lieu de se contenter d'être appelé de façon normale, attend dans son coin qu'un éclair se produise soudainement avant de daigner s'*exécuter*. En Perl, ces éclairs sont appelés des signaux, et elles sont distribuées avec la fonction prédéfinie `kill`. Voir le hachage `%SIG` dans le chapitre 28, *Noms spéciaux*, et la section *Signaux* dans le chapitre 16, *Communication interprocessus*.

GID

Group ID, identifiant de groupe ; sous UNIX, le numéro de groupe que le *système d'exploitation* utilise pour vous identifier, ainsi que tous les utilisateurs de votre groupe.

glob

Au sens strict, le caractère * du shell, qui recherche une correspondance avec un « glob » (une palanquée) de caractères quand on essaye de générer une liste de noms de fichiers. Plus généralement, le fait d'utiliser des globs et des symboles similaires afin d'effectuer des recherches de correspondances. Voir aussi *fileglob* et *typeglob*.

global

Ce que l'on peut voir de partout, concernant en général les *variables* et les *sous-programmes* visibles de partout dans le programme. En Perl, seules certaines variables sont vraiment globales ; la plupart des variables (comme tous les sous-programmes) sont locales au *paquetage* courant. Les variables globales peuvent être déclarées avec `our`. Voir *Déclarations globales* dans le chapitre 4, *Instructions et déclarations*.

granularité

La taille des objets avec lesquels vous travaillez.

grep

Vient de la vieille commande UNIX « Recherche globale d'un expression régulière et impression » (« Global search for a Regular Expression and Print »), utilisé maintenant pour tout type de recherches textuelles. Perl possède une fonction `grep` prédéfinie qui cherche dans une liste la correspondance suivant certains critères, alors que le programme `grep`(1) cherche les lignes qui correspondent à une *expression régulière* dans un ou plusieurs fichiers.

gros-boutiste

Décrit les ordinateurs qui rangent l'*octet* du poids le plus fort d'un mot à une adresse plus basse que l'octet du poids le plus faible, souvent considérés comme supérieurs aux ordinateurs petit-boutiste. D'après Swift : quelqu'un qui mange les œufs à la coque par le grand bout. Voir aussi *petit-boutiste*.

groupe

Un ensemble d'utilisateurs dont on fait partie. Sous certains systèmes d'exploitation (comme UNIX), il est possible de donner certaines permissions d'accès fichier aux autres membres de son groupe.

groupe de sélecteurs

La combinaison de plusieurs sélecteurs d'une commande (i.e `-a -b -c`) en un seul (i.e `-abc`). Un sélecteur avec *argument* doit être placé en dernier.

GV

Une « valeur glob » (*glob value*), signifiant un *typeglob*. Un GV est une sorte de *SV*.

hachage

Une liste nommée de paires *clef/valeur*, arrangée de telle manière qu'il soit facile d'utiliser une clef pour trouver sa valeur associée ; une relation binaire, pour les utilisateurs de bases de données. Ce glossaire ressemble à un hachage, où le mot à définir est la clef et la définition est la valeur. Un hachage est parfois également appelé un « tableau associatif » en six syllabes (ce qui est une bonne raison pour l'appeler un hachage).

hacker

Personne pugnace à résoudre les problèmes techniques, quel que soit le sujet qu'il s'agisse de jouer au golf, de combattre des orques ou de programmer. Hacker est un terme neutre, moralement parlant. Les

bons hackers ne doivent pas être confondus avec les *crackers* fous ou les *script kiddies* incompétents. Si vous les confondez, nous vous considérons comme fou ou incompétent.

handle de fichier

Ce avec quoi on sélectionne un fichier. Ou bien, un nom (qui n'est pas nécessairement le vrai nom du fichier) qui représente une instance donnée d'ouverture d'un fichier jusqu'à ce qu'il soit fermé. Si l'on doit ouvrir et fermer plusieurs fichiers à la suite, il est possible d'ouvrir chacun d'entre eux avec le même handle, ce qui fait que l'on n'a pas à réécrire du code pour traiter chacun d'entre eux.

handle de fichier indirect

Une *expression* dont la valeur est un *handle de fichier* : une *chaîne* (nom de handle de fichier), un *typeglob*, une *référence* de typeglob, ou un objet de bas niveau du module *IO*.

handle de répertoire

Un nom représentant une instance donnée de répertoire ouvert, jusqu'à ce qu'il soit fermé. Voir la fonction `opendir`.

handler

Un *sous-programme* ou *méthode* appelé par Perl quand votre programme doit répondre à un événement interne tel un *signal*, ou bien la rencontre avec un *opérateur de surcharge*. Voir aussi *fonction de rappel*.

héritage

Ce que nous laissent nos ancêtres, en termes génétiques ou autres. Si vous êtes une *classe*, vos ancêtres sont appelés *classes de base* et vos descendants *classes dérivées*. Voir aussi l'*héritage simple* et *héritage multiple*.

héritage multiple

Les caractéristiques de votre mère et de votre père, mélangées de façon imprévisible (voir *héritage* et *héritage simple*). Dans les langages informatiques (y compris Perl), la notion selon laquelle une classe donnée peut avoir plusieurs ancêtres directs ou *classes de base*.

héritage simple

Les caractéristiques héritées de votre mère, si elle vous a raconté que vous n'aviez pas de père (voir également *héritage* et *héritage multiple*). Dans les langages informatiques, la reproduction des *classes* est asexuée, et une classe donnée ne peut avoir qu'un seul ancêtre direct ou *classe de base*. Perl ne connaît pas ces restrictions, même si c'est possible de faire de cette manière.

hexadécimal

Un nombre en base seize, parfois raccourci en « hexa ». Les chiffres allant de dix à quinze sont conventionnellement représentés par les lettres allant de `a` à `f`. Les constantes hexadécimales commencent par `0` en Perl. Voir aussi la fonction `hex` au chapitre 29.

hôte

L'ordinateur sur lequel réside un programme ou des données.

HV

Abréviation de « hash value » (valeur de hachage), qui se réfère à l'un des types de données internes de Perl. Un HV est une sorte de *SV*.

identificateur

Un nom légalement formé pour tout ce qui peut intéresser un programme informatique. De nombreux langage (y compris Perl) autorisent les identificateurs commençant par une lettre et contenant des lettres et des chiffres. Perl considère également le caractère souligné comme une lettre valide. (Perl dispose aussi d'une notation complexe avec les noms *qualifiés*.)

impatience

La colère qui vous saisit quand l'ordinateur paresse. Vous êtes alors poussé à écrire des programmes qui ne se contentent pas de répondre à vos besoins, mais aussi à les anticiper. Idéalement. En conséquence, la deuxième grande vertu du programmeur. Voir également *paresse* et *orgueil*.

implémentation

La manière dont un morceau de code fait

réellement son travail. Les utilisateurs n'ont pas à s'occupper des détails de l'implémentation à moins qu'elle ne soit disponible à travers son *interface*.

importer

Avoir accès à des symboles qui sont exportés depuis un autre module. Voir l'opérateur `use` au chapitre 29.

incrémentation magique

Un opérateur d'*incrémentation* qui sait comment ajouter `1` tant à une chaîne alphanumérique qu'à un nombre.

incrémenter

Augmenter une valeur d'une unité (ou de tout autre nombre, si spécifié).

indexation

À l'origine, rechercher une *clef* dans un index (comme un annuaire), mais de nos jours, il s'agit surtout d'utiliser une clef ou une position pour trouver la *valeur* correspondante, même alors qu'aucun index n'entre en jeu. La situation a tellement dégénéré que la fonction `index` de Perl se contente de localiser la position d'une chaîne dans une autre.

indice

Une *valeur* indiquant la position d'un *élément* d'un *tableau*.

indirection

Quelque chose qui indique où se trouve la valeur cherchée. Ce qui peut se faire avec des *références symboliques* ou des *références dures*.

infixe

Un *opérateur* qui se place entre ses *opérandes*, tel la multiplication dans `24 * 7`.

inséré

Quand une chose est contenue dans une autre, même si c'est parfois surprenant : « J'ai inséré un interpréteur Perl dans mon éditeur ! ».

instance

Raccourci pour « instance de classe » qui signifie un *objet* de cette *classe*.

instruction

Une *commande* à l'ordinateur concernant ce qu'il faut faire ensuite, comme une étape dans une recette : « Ajouter du rhum à la pâte et mélanger ». À ne pas confondre avec une *déclaration*, qui ne dit pas à l'ordinateur de faire quoi que ce soit, mais d'apprendre quelque chose.

instruction de contrôle de boucle

Une instruction dans le corps de la boucle qui peut arrêter le bouclage ou sauter une *itération*. Voir la phrase du milieu de l'entrée précédente. N'essayez pas cela sur un vrai rouleau sans parachute.

instruction pendante

Une *instruction* simple, sans aucune accolade, après une condition `if` ou `while`. Le langage C les autorise mais pas Perl.

instruction switch

Une construction de programmation qui permet d'évaluer une *expression* et, d'après la valeur de l'expression, d'effectuer un débranchement à choix multiple vers le code approprié pour cette valeur. Également appelé une « structure `case` » d'après la construction Pascal similaire. La plupart des instructions switch en Perl sont des énumérations `for`. Voir *Structures de cas* dans le chapitre 4.

interface

Les services qu'un morceau de code promet de fournir, en constraste de son *implémentation*, qu'il peut modifier à son gré.

interpolation

L'insertion d'un scalaire ou d'une liste au milieu d'une autre valeur, sans que cela se voit. En Perl, l'interpolation de variable se produit dans les chaînes entre guillemets doubles et dans les motifs, et l'interpolation de liste se produit dans la construction d'une liste à passer en paramètre à un opérateur de liste ou tout autre construction qui prend une *`LIST`*.

interpolation de variable

L'*interpolation* d'une scalaire ou d'un tableau en une chaîne.

interpréteur

Au sens strict, un programme qui lit un second programme et fait ce que ce dernier lui dit sans d'abord le transformer sous une forme différente, ce qui est le comportement des *compilateurs*. Perl n'est pas un interpréteur selon cette définition, car il contient un compilateur qui prend un programme et en produit un (*arbre syntaxique*), dans le processus Perl lui-même, qu'interprète ensuite la machine virtuelle Perl temps-réel.

invocation

Le fait d'appeler une divinité, un démon, un programme, un sous-programme ou une fonction pour qu'il fasse ce qu'il a à faire. Les fonctions ou sous-programmes sont usuellement « appelés » alors que les méthodes sont « invoquées », ça sonne mieux.

I/O

Entrée depuis, ou sortie vers un *fichier* ou *périphérique*.

IO

Un objet I/O interne. Peut aussi vouloir dire *objet indirect*.

IP

Protocole Internet, ou propriété intellectuelle.

IPC

« Interprocess Communication », communication interprocessus.

itération

Faire une chose à répétition.

itérateur

Un machin de programmation spécial qui garde tout seul la trace de l'itération à l'endroit où l'on se trouve. La boucle Perl `foreach` contient un itérateur ; ainsi que le hachage, ce qui permet de lui appliquer l'opérateur `each` pour le parcourir.

IV

IV veut dire la Valeur interne Entière qu'un type *scalaire* peut porter, à ne pas confondre avec *NV*.

JAPH

« Just Another Perl Hacker », un bout de code trappu, mais illisible, qui lorsqu'exécuté s'évalue à cette chaîne. Souvent utilisé pour illustrer certaine particularités de Perl, et dans les concours de Perl obscurci.

langage rassembleur

Un langage, tel Perl, bon pour rassembler des choses différentes entre elles, qui n'étaient pas prévues pour s'assembler.

le plus à gauche, le plus long

Priorité d'une *expression régulière* pour établir la correspondance sur le *motif* le plus à gauche, et à partir de cette position d'essayer la plus longue (en supposant l'usage d'un quantificateur *avide*). Voir le chapitre 5 pour en savoir beaucoup plus sur le sujet.

lexème

Terme amusant pour dire *token*.

lexeur

Terme amusant pour dire *analyseur lexical*.

lexicale typée

Une *variable lexicale* déclarée avec un type *classe* : `my Poney $bill`.

librement diffusable

Signifie que l'on ne risque rien à en donner des copies piratées à des amis, même si cela se sait. N'hésitez pas à le faire pour Perl.

librement disponible

Signifie qu'il n'est pas besoin de payer pour l'obtenir, mais le copyright peut toujours rester propriété de quelqu'un d'autre (comme Larry).

lien

Utilisé comme nom, un répertoire représentant un fichier. Un fichier donné peut avoir plusieurs liens, comme quand plusieurs noms de l'annuaire se rapportent au même numéro de téléphone.

lien symbolique

Un nom de fichier supplémentaire qui pointe sur le vrai nom de fichier, lequel à son tour pointe sur le fichier. Quand le *sys-*

tème d'exploitation essaye d'analyser un nom de chemin contenant un lien symbolique, il se contente de substituer le vrai nom et continue l'analyse.

lier

Associer une *adresse réseau* à une *socket*.

LIFO

Dernier arrivé, premier sorti. Voir aussi *FIFO*. Une LIFO est plutôt appelé une *pile*.

ligne

Sous UNIX, une séquence de zéro, un ou plusieurs caractères autres que le saut de ligne, terminée par un caractère *saut de ligne*. Sur les machines non UNIX, ce concept est émulé même si le *système d'exploitation* sous-jacent a sa propre idée sur la question.

lisible

1) Fichier dont le bit d'autorisation adéquat est à `1`, ce qui permet l'accès en lecture. 2) Programme informatique assez bien écrit pour que quelqu'un d'autre puisse revenir plus tard en ayant une chance de comprendre ce qu'il fait.

LIST

Une construction syntaxique représentant une liste d'expressions séparées par des virgules, dont l'évaluation produit une *liste de valeurs*. Chaque expression dans une *LIST* est évaluée dans un *contexte de liste* et interpolée en une liste de valeurs.

liste

Une liste ordonnée de valeurs.

liste nulle

Une *valeur de liste* de zéro éléments, représentés en Perl par `()`.

littéral

Un token tel un nombre ou une *chaîne* qui vous donne une *valeur* effective au lieu de simplement contenir une valeur, ce que fait une *variable*.

littéral scalaire

Un nombre ou une *chaîne* entre apostrophes ; une vraie *valeur* dans le texte du programme, par opposition avec une *variable*.

local

Ne signifiant pas la même chose partout. Une variable globale Perl peut être rendue locale à une *portée dynamique* par l'opérateur `local`. Voir *portée dynamique*.

logiciel à source ouvert

Programmes dont le code source est librement disponible et que l'on peut redistribuer sans droit commerciaux attaché. Pour une définition plus détaillée, voir *http://www.opensource.org/osd.html*.

longueur zéro

Une *assertion* qui correspond à une *chaîne nulle* entre deux *caractères*.

lvalue

Terme utilisé par les puristes des langages informatiques (et abhorré par les puristes des langages humains) concernant un endroit auquel une nouvelle *valeur* peut être assignée, comme une *variable* ou un élément de *tableau*. Le « l » signifie « left », pour le côté « gauche » d'une assignation, un endroit typique pour une lvalue. Une fonction ou expression « normale » à laquelle on peut affecter une valeur comme dans `pos($x) = 10`.

magique

Sémantique attachée aux variables telles que `$!`, `$0`, `%ENV`, `%SIG`, ou tout autre variable liée. Des choses magiques se produisent lors de l'utilisation de ces variables.

Makefile

Un fichier qui contrôle la compilation d'un programme. Les programmes Perl n'en ont pas besoin en général, car le compilateur a beaucoup de self-control.

man

Le programme Unix qui affiche la documentation en ligne (pages de manuel).

marqué

Caractéristique de données pouvant provenir des doigts malodorants d'un utilisateur, et donc dangereux du point de vue d'un programme sécurisé. Perl effectue des vérifications d'entachement (« taint checks ») si on lance un programme *setuid* (ou *setgid*) ou si l'on active l'option **-T**.

membre

Voir *variable d'instance*.

mémoire

Concerne toujours la mémoire principale, jamais le disque. La mémoire *virtuelle* laisse croire que la mémoire est plus importante, mais on ne peut la substituer à la mémoire physique. La seule différence en cas de limite mémoire atteinte, est que le système mettra plus de temps à crasher, alors qu'avec la mémoire physique il s'arrêtera tout de suite.

mémoire partagée

Un morceau de *mémoire* accessible par deux *processus* différents qui autrement ne pourraient voir la mémoire de l'autre.

métacaractère

Un *caractère* qui n'est pas censé être traité normalement. Le choix des caractères à traiter de manière spéciale comme métacaractères varie grandement avec le contexte. Le *shell* peut en accepter certains, les chaînes Perl protégées par des apostrophes doubles en ont d'autres, et les motifs d'*expression régulière* connaissent ces derniers plus d'autres encore.

métasymbole

Une séquence de *caractères* dont le premier est un *métacaractère*.

méthode

L'action que fait un *objet* quand on lui demande. Voir le chapitre 12, *Objets*.

méthode d'accès

Une *méthode* permettant de lire ou de modifier une *variable d'instance* d'un objet.

méthode de classe

Une *méthode* dont l'invocant est le nom d'un *paquetage* et non une référence d'*objet*. Cette méthode s'applique à la classe entière.

méthode statique

Ne se dit pas. Voir *méthode de classe*

minimalisme

La croyance selon laquelle « moins c'est mieux ». Paradoxalement, lorsque l'on dit quelque chose dans un petit langage, on obtient un grand discours, et lorsque l'on dit la même chose dans un grand langage on obtient un petit discours. Allez savoir.

mode

Dans le contexte de l'appel système *stat*(2), se réfère au mot contenant les permissions et le type du fichier.

modificateur

Voir *modificateur d'instruction*, *modificateur d'expression régulière*, et *modificateur de lvalue*.

modificateur de lvalue

Pseudo-fonction adjective qui qualifie la portée d'une déclaration de variable. Il y a actuellement 3 modificateurs de lvalue : `my`, `our`, et `local`.

modificateur d'expression régulière

Une option sur un motif ou une substitution telle que `/i` pour rendre le motif insensible à la casse. Voir aussi *cloisonneur*.

modificateur d'instruction

Une *condition* ou une *boucle* que vous mettez après une *instruction* au lieu de la mettre devant, si vous voyez ce que je veux dire.

module

Un *fichier* qui définit un *paquetage* (presque) du même nom, qui peut aussi bien *exporter* des symboles que fonctionner comme une *classe d'objet*. (Le fichier principal *.pm* d'un module peut aussi inclure d'autres fichiers). Voir l'opérateur `use`.

modulo

Un entier, quand on est plus intéressé par le reste que par le quotient.

mongueur

Raccourci pour Perl mongueur, un fournisseur de Perl.

mortel

Une valeur temporaire prévue pour durer le temps d'une instruction.

mot

En « informaticianais », le bout de données de la taille la plus efficace pour un

ordinateur, typiquement 32 bits à quelques puissances de deux près. Dans la culture Perl, se réfère souvent à un *identificateur* (avec ou sans caractère alphanumérique), ou à une chaîne de caractères sans espaces délimitée par des espaces ou des limite de chaîne.

mot-clé

Voir *mots réservés*.

mot simple

Un mot suffisamment ambigu pour que Perl l'estime illégal, lorsque la directive `use strict 'subs'` est en vigueur. Sans cette directive, ce mot isolé est traité comme avec des guillemets.

motif

Modèle utilisé en *reconnaissance de correspondance*.

motif à l'exécution

Un motif qui contient une ou plusieurs variables à interpoler avant que ce motif soit analysé comme une *expression régulière*, et qui ne peut donc pas être analysé à la *compilation*, mais doit être reanalysé à chaque fois que l'opérateur de recherche de correspondance est évalué. Les motifs d'exécution sont utiles mais coûteux.

mots réservés

Un mot-clef possédant une signification interne spécifique pour un *compilateur*, comme `if` ou `delete`. En de nombreux langages (pas en Perl), il est illégal d'utiliser des mots réservés pour nommer quoi que ce soit d'autre (c'est d'ailleurs pourquoi ils sont réservés, après tout). En Perl, il est simplement interdit de les utiliser pour nommer les *étiquettes* et les *handles de fichiers*. Aussi appelés *mot-clé*.

nettoyer

Dernière opération effectuée par un *processus* parent lors de la terminaison d'un processus fils de telle sorte qu'il ne reste pas en mémoire. Voir les appels des fonctions `wait` et `wait`.

NFS

Système de fichiers en réseau qui permet de monter un système de fichier distant comme s'il était local.

nom de commande

Le nom du programme en cours d'exécution, tel qu'il a été entré sur la ligne de commande. En C, le nom de la commande est passé au programme comme premier argument. En Perl, il est connu sous le nom de `$0`, et séparé des arguments.

nom de fichier

Le nom donné à un fichier. Ce nom est listé dans un *répertoire*, et on peut l'utiliser dans une instruction `open` pour indiquer au *système d'exploitation* quel fichier ouvrir exactement, et l'associer à un *handle de fichier* qui servira d'identité pour les accès suivants dans votre programme jusqu'à sa fermeture.

nombre en virgule flottante

C'est la « notation scientique » qui permet de représenter les nombres avec une précision indépendante de l'ordre de grandeur (le point décimal flotte). Perl effectue les calculs numériques avec des nombres à virgule flottante (parfois appelés des « flottants ») lorsque les entiers ne suffisent plus. Ce sont des approxiations des nombres réels.

normale

Qui peut servir de *lvalue*.

numéro de ligne

Le nombre de lignes avant celle-ci, plus `1`. Perl conserve un numéro de ligne séparé pour chaque script ou fichier d'entrée qu'il ouvre. Le numéro de ligne du script courant est représenté par `__LINE__`. Le numéro de ligne d'entrée courant (du fichier qui a été le plus récemment ouvert par `<FH>`) est représenté par la variable `$.` (`$INPUT_LINE_NUMBER`). De nombreux messages d'erreur renvoient les deux valeurs, si elles sont disponibles.

NV

Valeur numérique interne du type virgule flottante qu'un *scalaire* peut représenter, à ne pas confondre avec *IV*.

objet

Une *instance* de classe. Quelque chose qui « sait » à quoi il ressemble, et ce qu'il peut faire car il connaît sa classe. Le programme peut demander à un objet de faire des cho-

ses, mais c'est l'objet qui décide s'il veut le faire ou non. Certains sont plus accomodants que d'autres.

objet indirect

Terme grammatical indiquant le complément d'objet indirect bénéficiaire ou récepteur de l'action. En Perl, l'instruction `print STDOUT "$foo\n";` peut être interprétée grammaticalement comme « verbe objet-indirect objet » où `STDOUT` est le récepteur de l'action `print`, et `"$foo"` est l'objet imprimé. De la même manière, quand on invoque une *méthode* on place l'invocant entre la méthode et les arguments :

```
$golem =
  newCreature::Pathetique"Smeagol";
donner $gollum "Poisson!";
donner $gollum "Tressor!";
```

octal

Un nombre en base huit. Seuls les chiffres allant de zéro à sept sont autorisés. Les constantes octales commencent par un zéro en Perl, comme pour `013`. Voir aussi la fonction `oct`.

octet

Un bout de données long de huit *bits* la plupart du temps.

one-liner

Un programme entier concentré sur une ligne de texte.

opérande

Une *expression* qui donne une *valeur* sur laquelle un *opérateur* agit. Voir aussi *précédence*.

opérateur

Une *fonction*, généralement intégrée au langage, souvent avec une syntaxe ou un symbole spécial, qui transforme des entrées en sorties. Un opérateur donné peut attendre des types de données spécifiques en tant qu'arguments (*opérandes*), ainsi qu'en tant que résultat.

opérateur arithmétique

Un symbole tel que `+` ou `**` qui indique à Perl d'effectuer un calcul arithmétique, tel un de ceux appris à l'école. Voir également *opérateur*.

opérateur binaire

Un *opérateur* qui prend deux *opérandes*.

opérateur circonfixe

Un *opérateur* qui entoure ses *opérandes*, commme l'opérateur d'angle ou les parenthèses, ou une embrassade.

opérateur d'adressage

Certains langages travaillent directement avec les valeurs des adresses mémoire, ce qui peut être dangereux. Perl fournit des gants isolants pour ce genre d'opération. L'opérateur le plus proche en Perl est le backslash qui s'évalue à une *référence dure* beaucoup plus sûre qu'une adresse mémoire.

opérateur d'affectation

Soit une *affectation* régulière, soit un opérateur composé, formé par l'association d'une affectation ordinaire avec un autre opérateur qui modifie la valeur d'une variable en place, c'est-à-dire relativement à son ancienne valeur. Par exemple, `$a += 2` ajoute `2` à `$a`.

opérateur de liste

Un *opérateur* comme `join` ou `grep`qui agit sur une liste de valeurs. Habituellement utilisé pour désigner des opérateurs prédéfinis (tels `print`, `unlink`, et `system` qui ne nécessitent pas de parenthèses autour de leurs liste d'*arguments*.

opérateur de test de fichier

Un opérateur interne de Perl qui permet de déterminer si quelque chose est *vrai* concernant un fichier, tel le test `-o $filename` qui retourne vrai si vous êtes propriétaire du fichier.

opérateur logique

Symboles représentant les concepts « et » , « ou » , « ou exclusif » et « non » .

opérateur relationnel

Un *opérateur* qui indique si une relation d'ordre donnée est *vraie* concernant une paire d'*opérandes*. Perl possède des opérateurs relationnels sur les chaînes et sur les nombres. Voir *séquence de tri*.

opérateur unaire

Un opérateur avec un seul *opérande*,

comme ! ou `chdir`. Ce sont en général des opérateurs préfixés, ils précèdent leurs opérandes. Les opérateurs `++` et `--` sont préfixés ou postfixés. (Leur position modifie leur sens.)

opération atomique

Lorsque Démocrite inventa le mot « atome » pour désigner la plus petite quantité de matière, il voulait dire litéralement insécable *a-* (privatif) suivi de *tomos* (sécable) : insécable. Une opération atomique est une action qui ne peut être interrompue.

options

Voir *sélecteurs* ou *modificateur d'expression rationnelle*.

orgueil

Fierté excessive, le genre de chose qui attire les foudres de Jupiter. Mais aussi une qualité qui fait écrire (et maintenir) des programmes dont les autres ne voudront pas dire de mal. En conséquence, la troisième grande vertu du programmeur. Voir également *paresse* et *impatience*.

pad

Contraction de « scratchpad », voir *brouillon*.

page de manuel

Une « page » des manuels UNIX (« manpage »), à laquelle on accède typiquement par la commande *man*(1). Une page de manuel contient un synopsis, une description, une liste de bogues, etc., et est généralement plus longue qu'une page. Il existe des pages de manuel documentant les *commandes*, les *appels systèmes*, les *fonctions de bibliothèque*, les *périphériques*, les *protocoles*, les *fichiers*, et ainsi de suite. Dans ce livre, toute partie de la docmentation Perl standard telle *perlop* ou *perldelta*, est appelée page de manuel indépendamment du format utilisé par votre système.

paquet

Groupe de modules communs sur *CPAN*. (Dénomme parfois aussi un groupe d'options de la ligne de commande rassemblés en une *groupe d'options*.)

paquetage

Un *espace de noms* pour les *variables* globales, les *sous-programmes* et autres, qui les distinguent de ceux de même noms dans un autre espace. En un sens, seul un paqueage est global puisque on accède à un symbole de sa table en nommant ce paquetage. Mais dans un autre sens, tous les symboles des packages sont globaux, mais organisés comme il faut.

paquetage courant

Le *paquetage* dans lequel le code en cours est compilé. Parcourez votre code en arrière dans la portée *lexicale* courante et les portées englobantes. La première déclaration « package » que vous trouvez est le paquetage courant.

paramètre

Voir *argument*.

paresse

La qualité qui vous pousse à fournir des efforts surhumains dans le but de réduire le travail total. Vous êtes ainsi conduit à écrire des programmes qui, au total, économisent un dur labeur et que d'autres trouvent utiles, et à documenter ce que vous avez écrit pour ne pas devoir répondre sans arrêt aux mêmes questions. En conséquence, la première grande vertu du programmeur. Voir aussi *impatience* et *orgueil*.

patch

Dans le monde des hackers, une liste des différences entre deux versions d'un programme que l'on peut appliquer avec le programme *patch*(1) pour la mise à jour ou la correction d'un bogue.

PATH

La liste de *répertoires* qu'examine le système pour trouver un programme à *exécuter*. La liste est stockée dans une *variable d'environnement*, accessible en Perl par `$ENV{PATH}`.

périphérique

Un super matériel de la mort (comme un disque, une unité à bande, un modem ou un joystick) rattaché à l'ordinateur que le *système d'exploitation* essaye de faire ressem-

bler à un *fichier* (ou à un groupe de fichiers). Sous UNIX, ces faux fichiers tendent à résider dans le répertoire */dev*.

petit-boutiste

Décrit les ordinateurs qui rangent l'*octet* du poids le plus fort d'un mot à une adresse plus haute que l'octet du poids le plus faible, souvent considérés comme supérieurs aux ordinateurs gros-boutiste, d'après Swift : quelqu'un qui mange les œufs à la coque par le petit bout.

phase de compilation

Phase avant l'exécution. Voir aussi *phase d'exécution*. Elle effectue surtout de la *compilation*, mais elle peut également effectuer un peu d'*exécution* dans l'évaluation des blocs `BEGIN`, des déclarations `use` ou des sous-expressions constantes. Le code de démarrage et d'importation des déclarations `use` est aussi exécuté dans cette phase.

phase d'exécution

Dès que Perl commence à exécuter votre programme. Voir aussi *phase de compilation*. La phase d'exécution effectue surtout de l'*exécution*, mais elle peut effectuer de la *compilation* lorsqu'elle exécute les opérateurs `require`, `do FILE`, or `eval STRING`, ou lorsqu'une substitution utilise le modificateur `/ee`.

pile

Un dispositif qui permet d'empiler des objets dessus et plus tard de les récupérer dans l'ordre inverse. Voir *LIFO*.

pipeline

Une série de *processus* en ligne, reliés par des *tubes*, où chacun passe sa sortie au suivant.

plateforme

Le contexte hardware et software dans lequel un programme tourne. Un programme écrit dans un langage qui dépend d'une plate-forme peut crasher si vous changez l'une des choses suivantes : la machine, le système d'exploitation, les bibliothèques, le compilateur ou la configuration du système. L'interpréteur *perl* doit être recompilé à chaque *portage* car il est écrit en C, mais les programmes écrits en Perl sont dans une large mesure indépendants de la plateforme.

pod

Les balises pour inclure la documentation dans votre code. Voir chapitre 26, *POD*.

point d'arrêts

Une ligne marquée dans le source où le déboggueur stoppe l'*exécution*, afin de pouvoir jeter un coup d'œil et voir si quelque chose a commencé à mal fonctionner.

pointeur

Une *variable* dans un langage comme C qui contient l'emplacement exact d'un autre élément. Perl gère les pointeurs en interne et l'on n'a pas besoin de s'en soucier. On utilise des pointeurs symboliques sous la forme de *clefs* et de noms de *variables*, ou de *références en dur*, qui ne sont pas des pointeurs (mais agissent comme tels et, en fait, contiennent des pointeurs).

polymorphisme

Le fait de pouvoir dire à un *objet* de faire quelque chose de générique, et l'objet l'interprétera de diverses façons, en fonction de sa forme (vient du Grec, formes nombreuses).

port

La partie de l'adresse d'une socket TCP ou UDP qui dirige les paquets vers le processus correct après avoir trouvé la bonne machine, un peu comme le numéro de poste de quelqu'un à qui l'on veut téléphoner au bureau.

portable

Il y a longtemps, du code compilable à la fois sur BSD et SysV. En général, du code qui est relativement facile à convertir pour fonctionner sur une autre *plateforme*. On peut considérer que tout code est portable, il suffit d'y mettre du sien.

portage

Se dit aussi de la conversion du code machine vers une autre machine.

portée

Distance à laquelle une variable reste visible, du point de vue d'une autre variable.

Perl comprend deux mécanismes de visibilité : la *portée dynamique* d'une *variable* locale, ce qui veut dire que le reste du *bloc* et tous les *sous-programmes* appelés dans le reste du bloc peuvent voir les variables locales au bloc ; et la *portée lexicale* des variables `my`, ce qui signifie que le reste du bloc peut voir les variables, mais ce n'est *pas* le cas pour les autres sous-programmes appelés par le bloc.

portée dynamique

Les variables à portée dynamique sont visibles dans tout le reste du *bloc* dans lesquelles elles ont été d'abord utilisées, ainsi que dans les *sous-programmes* appelés par le reste du bloc. Les variables à portée dynamique peuvent voir leur valeur temporairement modifiées (et restaurées implicitement plus tard) par une instruction *local*. Comparer avec *portée lexicale*. Employé plus généralement pour décrire la façon dont un sous-programme en train d'en appeler un autre « contient » ce sous-programme à l'*exécution*.

portée lexicale

La zone d'influence du Dictionnaire de l'Académie Française (connue également sous le nom de portée *statique*, vu la période de renouvellement du dictionnaire en question). De même, la propriété de variables rangées dans un dictionnaire propre à chaque bloc (espace de nom), qui ne sont visibles que du point de leur déclaration à la fin du bloc dans lequel elles ont été déclarées. — Syn. *portée statique*. — Ant. *portée dynamique*.

portée statique

Ne se dit pas. Voir *portée lexicale*.

porteur

Une personne qui « porte » le logiciel d'une *plate-forme* vers une autre. Le portage de programmes écrits en C peut être un travail difficile, mais celui d'un interpréteur Perl est très complexe, mais la récompense est en proportion.

POSIX

La spécification des systèmes d'exploitation portables.

postcurseur

Une *assertion* qui cherche la chaîne à droite de la correspondance courante.

postfixe

Un *opérateur* qui suit son *opérande* comme dans `$x++`.

pp

Une abréviation pour le code « push-pop » qui implémente en C la machine à pile de Perl.

pragma

Un module de bibliothèque dont les suggestions et les conseils pratiques sont reçus (et parfois ignorés) par le compilateur. Les pragmas ont des noms en caractères minuscules.

précédence

Les règles de conduite qui, en l'absence d'autres directives, déterminent ce qui doit se produire en premier. Par exemple, en l'absence de parenthèses, la multiplication se produit toujours avant l'addition.

précurseur

Une *assertion* qui cherche la chaîne à gauche de la correspondance courante.

prédéfini

Une *fonction* prédéfinie du langage. Même si elle est *surchargée*, on peut toujours y accéder en *qualifiant* son nom avec le pseudo-paquetage `CORE::`.

préfixe

Un *opérateur* qui précède son *opérande*, comme dans `++$x`.

préprocesseur

Un *processus* d'aide pour préparer des données pour le processus courant. Souvent effectué à l'aide d'un *tube*. Voir aussi *préprocesseur*.

préprocesseur C

La première passe typique du compilateur C, qui traite les lignes qui commencent par un # pour compiler de manière conditionnelle et définir les macros, et qui effectue diverses manipulations du texte du programme en se basant sur les défini-

tions courantes. Également connu sous le nom de *cpp*(1).

procédure

Un *sous-programme*.

processus

Une instance d'un programme en train de tourner. Sous des systèmes multitâches comme UNIX, plusieurs processus peuvent faire tourner le même programme de façon indépendante en même temps ; en fait, la fonction `fork` est conçue pour en arriver à cet heureux état de fait. Sous d'autres systèmes d'exploitation, les processus sont parfois appelés « tâches » ou « jobs » souvent avec de faibles nuances de signification.

propriétaire

L'utilisateur unique (le super-utilisateur mis à part) qui exerce un contrôle absolu sur un *fichier*. Un fichier peut également appartenir à un *groupe* d'utilisateurs qui en partagent la propriété si le propriétaire réel le permet. Voir *bits d'autorisation*.

propriété

Voir *variable d'instance* ou *propriété de caractère*.

propriété de caractère

Une *classe de caractère* prédéfinie retournée par le *métasymbole* `\p`. La plupart de ces propriétés sont définies pour *Unicode*.

protocole

En réseau, une manière convenue d'envoyer et de recevoir des messages qui permet aux deux correspondants de ne pas trop se mélanger les pinceaux.

prototype

Partie optionnelle d'une déclaration de *sous-programme* indiquant au compilateur Perl le type et le nombre de ses *paramètres réels*, qui rend les sous-programmes semblables à des fonctions prédéfinies.

pseudofonction

Une construction qui ressemble à une fonction mais qui n'en est pas une. Utilisé pour les modificateurs de *lvalue* comme `my`, pour les modificateurs de *contexte* comme `scalar`, et pour les constructions comme `q//`, `qq//`, `qx//`, `qw//`, `qr//`, `m//`, `s///`, `y///` et `tr///`.

pseudohachage

Une référence à un tableau dont le premier élément contient une référence à un hachage. Un pseudohachage peut être considéré comme une référence à un tableau ou une référence à un hachage.

pseudo littéral

Un *opérateur* qui ressemble à un *littéral*, comme l'opérateur d'absorption de sortie, *`commande`*.

pumpkin

Un hypothétique « sceptre » qui se transmet d'une personne à l'autre dans la communauté Perl pour savoir qui gouverne tel ou tel domaine de développement.

PV

Abréviation de « pointer value », qui signifie `char*` en Jargon Interne Perl.

qualifié

Nom défini complètement défini. Le symbole `$Foo::bar` est qualifié ; `$bar` ne l'est pas. Un nom de fichier qualifié est le chemin complet depuis la racine des répertoires.

quantificateur

Composant d'une *expression régulière* spécifiant le nombre d'occurrence d'un *atome*.

quartet

La moitié d'un *octet*, équivalent à un chiffre *hexadécimal*, et qui vaut quatre *bits*.

ramasse-miettes

Une fonctionnalité de certains langages de programmation, qui devrait plutôt s'appeler « Maman va ranger la chambre ». Ce n'est pas exactement ce que fait Perl, mais il se fie à un mécanisme de comptage de références pour laisser les lieux aussi propres en sortant qu'en entrant. Cependant, si ça peut vous rassurer quand l'interpréteur sort, un vrai ramasse-miettes est lancé pour s'assurer que des références circulaires (entre autres) ne traînent pas dans les coins.

recherche de correspondance

Prendre un motif, exprimé par une *expression régulière*, et l'essayer de diverses manières sur une chaîne pour voir s'il y a moyen de le faire correspondre. Souvent utilisé pour extraire des informations intéressantes d'un fichier.

reconnaissance de motif progressive

Une *reconnaissance de motif* qui reprend à partir de là où il s'est arrêté .

récursion

L'art de définir un chose (au moins partiellement) en fonction d'elle-même, ce qui dans un dictionnaire est absolument exclu, mais qui dans un programme d'ordinateur équivaut à une boucle infinie avec une condition d'arrêt exceptionnelle, en faisant attention d'atteindre un jour cette condition.

référence

Un endroit où l'on trouve un pointeur vers des informations stockées ailleurs (voir *indirection*). Les références existent sous deux formes, *références symboliques* et *références en dur*.

référence arrière

Une sous-chaîne *capturée* par un motif partiel entre parenthèses simples, à l'intérieur d'une *regex*. Les termes (`\1`, `\2`, etc.) mémorisent une référence pour les occurrences correspondantes des sous-motifs. Hors du motif, les variables nombres `$1`, `$2`, etc.) référencent les mêmes valeurs, tant que le champ dynamique du motif est valable.

référence en dur

Une *valeur scalaire* contenant l'adresse réelle d'un *référent*, telle que la renvoie le compteur de *références*. (Certaines références en dur sont contenues en interne, comme la référence implicite de l'un des emplacements de variable d'un *typeglob* à son référent correspondant). Une référence en dur est différente d'une *référence symbolique*.

référence soft

Voir *référence symbolique*.

référence symbolique

Une variable dont la valeur est le nom d'une autre variable ou d'un sous-programme. *Déréférencer* la première variable permet d'obtenir la seconde. Les références symboliques sont interdites avec le pragma `use strict 'refs'`.

référent

Ce à quoi réfère une référence, qui n'a pas forcément un nom. Les types de référent communs sont les scalaires, le tableaux, le hachages et les sous-programmes.

regex

Voir *expression régulière*.

remplacement

Dissimuler ou invalider une autre définition du même nom (à ne pas confondre avec la *surcharge*, qui se contente d'ajouter des définitions lesquelles sont distinguées par un autre moyen). Comme si ce n'était pas suffisant, nous employons ce mot avec deux définitions surchargées : pour décrire comment il est possible de définir un *sous-programme* personnel cachant une *fonction* interne du même nom (voir *Supplanter des fonctions internes* dans le chapitre 11, *Modules*) et pour décrire comment définir une *méthode* de remplacement dans une *classe dérivée* cachant la méthode d'une *classe de base* du même nom (voir le chapitre 12).

répertoire

Un endroit où résident les fichiers et éventuellement d'autres répertoires. Certains *systèmes d'exploitation* peuvent le nommer « dossier », « *drawer* » (tiroir) ou « catalogue ». Également appelé « directory » dans certains langages humanoïdes.

répertoire courant

Voir *répertoire de travail*.

répertoire de base

Le répertoire dans lequel on se trouve quand on s'est identifié sur le système. Sur un système UNIX, le nom est souvent placé dans `$ENV{HOME}` ou `$ENV{LOGDIR}` par le programme de login, mais on peut également le trouver avec `getp-`

`wuid($<)[7]`. (Certains systèmes sont dépourvus de cette notion.)

répertoire de travail

Votre *répertoire* courant à partir duquel les noms de fichiers relatifs sont calculés. Le *système d'exploitation* le connaît par la fonction `chdir` ou parce que votre *processus* parent a démarré à cet endroit.

RFC

« Request For Comment », appel à commentaires, qui sont des séries de documents standards importants, contrairement à ce que laisse supposer leur dénomination timide.

root

Le super-utilisateur (UID == 0). C'est aussi le répertoire racine du système de fichier.

RTFM

Sigle anglophone pour vous suggérer de bien vouloir lire le superbe manuel.

RV

Une Référence de Valeur interne du type de celle qu'un *scalaire* représente. Voir aussi *IV* et *NV.*

rvalue

Une *valeur* que l'on peut trouver du côté droit d'une *assignation*. Voir également *lvalue*.

saut de ligne

Un caractère unique représentant la fin d'une ligne, d'une valeur de 012 octal sous UNIX (mais 015 sur un Mac), et représenté par \n dans les chaînes Perl. Pour les machines sous Windows et certains périphériques physiques comme les terminaux, il est traduit automatiquement par votre bibliothèque C en saut de ligne et retour chariot, mais normalement aucune traduction n'est effectuée.

scalaire

Une valeur simple, comme un nombre, une *chaîne* ou une *référence*.

script

Un *fichier* texte qui est un programme à *exécuter* directement au lieu d'avoir à le *compiler* sous une autre forme avant l'*exécution*. Aussi, dans le contexte *Unicode*, un système d'écriture pour un langage particulier tel le Grec, le Bengali ou le Klingon.

script kiddie

Un *cracker* qui n'est pas un *hacker*, mais en connaît juste assez pour exécuter des scripts tout faits. En général, les programmes qu'il écrit sont de simples imitations de programmes existants.

sed

Un vénérable éditeur duquel Perl dérive certaines idées.

sélecteur

Une option de la ligne de commande qui modifie le comportement d'un programme, habituellement notée avec le signe moins.

sémaphore

Jolie sorte de signaux mutuels qui empêchent les *threads* ou les *processus* d'utiliser les mêmes ressources simultanément.

séparateur

Un *caractère* ou une *chaîne* qui sépare deux chaînes entre elles. La fonction `split` fonctionne sur les séparateurs. À ne pas confondre avec les *délimiteurs* ou les *terminateurs*. Le « ou » de la première phrase sépare deux alternatives.

séquence d'échappement

Voir *métasymbole*.

séquence de tri

L'ordre dans lequel les caractères sont triés. Utilisé par les routines de comparaison de chaînes pour décider, par exemple, où mettre dans le glossaire « séquence de tri ».

sérialisation

Mettre une *structure de données* en ordre linéaire de façon à pouvoir la stocker comme une *chaîne* sur un fichier disque ou dans une base de données ou bien l'envoyer dans un *tube*.

serveur

En réseau, un *processus* qui peut publier un *service* ou se contenter d'attendre à un endroit donné que des *clients* ayant besoin d'un service se manifestent.

service

Quelque chose que l'on fait à une personne pour lui faire plaisir, comme lui indiquer l'heure. Sur certaines machines, les services les plus connus sont listés par la fonction `getserver`.

setgid

Comme *setuid*, mais concernant les privilèges des *groupes*.

setuid

Un programme tournant avec les privilèges de son *propriétaire* au lieu des privilèges de l'utilisateur (ce qui est en général le cas). Décrit aussi le bit du mot *mode* (flags de permissions) qui implémente cette fonctionnalité. Ce bit doit être explicitement activé par le propriétaire, et le programme doit être écrit de façon à ne pas donner plus de privilèges que nécessaire.

shebang

En Perl c'est la séquence `#!` qui indique au système où trouver l'interpréteur. Le terme vient de la construction du mot anglais « sharp » désignant # et de « bang » pour !.

shell

Un *interpréteur* de ligne de *commande*. Le programme qui donne une invite interactive, accepte une ou plusieurs *lignes* d'entrée et exécute les programmes mentionnés, leur fournissant leurs *arguments* et leurs données d'entrée. Les shells peuvent également exécuter des scripts contenant ces commandes. Sous le *système d'exploitation* UNIX, les shells habituels sont le Bourne shell (*/bin/sh*), le C shell (*/bin/csh*) et le Korn shell (*/bin/ksh*). Perl n'est pas strictement un shell car il n'est pas interactif (bien que des programmes Perl puissent être interactifs).

signal

Un éclair soudain ; ou plutôt un événement déclenché par le *système d'exploitation*, à n'importe quel moment, de préférence au moment où vous ne vous y attendez pas.

slurp

Lire un *fichier* en entier dans une *chaîne* en une opération.

socket

Un point de communication réseau entre deux *processus*, qui fonctionne comme un téléphone ou une boîte à lettres. La chose la plus importante concernant une socket est son *adresse réseau* (comme un numéro de téléphone). Des types différents de sockets ont des types d'adresse différents ; certains ressemblent à des noms de fichiers, d'autres non.

sortie standard

Le flux de sortie par défaut d'un programme, qui ne doit pas se préoccuper de la destination des données. Représenté dans un programme Perl par le *handle de fichier* `STDOUT`.

sous-chaîne

Une partie d'une chaîne, commençant à partir d'une certaine position de *caractère* (*déplacement*), et courant sur un certain nombre de caractères.

sous-classe

Voir *classe dérivée*.

sous-motif

Composant un motif d'une *expression rationnelle*

sous-motif de code

Sous-motif d'une *expression rationnelle* dont la fonction est d'exécuter du code Perl, par exemple les sous-motifs `(?{...})` et `(??{...})`.

sous-programme

Une partie nommée d'un programme qui peut être invoquée d'un autre endroit du programme pour atteindre un but secondaire du programme. Un sous-programme est souvent paramétré pour accomplir des tâches différentes mais en rapport selon les *arguments* fournis en entrée. Si la *valeur* que le programme retourne a une signification.

standard

Inclus dans la distribution officielle de Perl comme module standard, outil standard ou bien *page de manuel* standard.

statique

Variant lentement, comparé à autre chose.

(Malheureusement, tout est relativement stable comparé à autre chose, à part certaines particules élémentaires, et encore). Dans les ordinateurs, où les choses sont censées varier rapidement, « statique » a une connotation péjorative, indiquant une *variable*, une *méthode* ou un *sous-programme* légèrement défectueux. Dans la culture Perl, nous évitons d'utiliser ce mot.

statut

La *valeur* renvoyée au *processus* père quand l'un de ses processus fils meurt. Cette valeur est placée dans la variable spéciale `$?`. Ses huit *bits* de poids fort sont le statut de sortie du processus défunt, et ses huit bits de poids faible identifient le signal (s'il y a lieu) dont est mort le processus. Sur les systèmes UNIX, cette valeur de statut est la même que le mot qui est renvoyé par *wait*(2). Voir `system` dans le chapitre 29.

statut de sortie

Voir *statut*.

STDERR

Voir *erreur standard*.

STDIN

Voir *entrée standard*.

STDIO

Voir *E/S standard*.

STDOUT

Voir *sortie standard*.

struct

Un mot-clef C présentant une définition ou un nom de structure.

structure

Voir *structure de données*.

structure de donnée

La façon dont les données s'articulent entre elles et la forme qu'elles constituent ensemble, comme par exemple un arbre de forme triangulaire ou une table rectangulaire.

structure stat

Un endroit spécial dans lequel Perl garde l'information sur le dernier fichier consulté.

substitution

Pour changer des parties de chaîne avec l'opérateur `s///`. Ne pas confondre avec l'*interpolation de variable*.

sucre syntaxique

Façon différente d'écrire quelque chose : raccourci.

superclasse

Voir *classe de base*.

super-utilisateur

La personne à qui le *système d'exploitation* laisse faire presque tout ce qu'il veut. Typiquement, l'administrateur système, ou quelqu'un prétendant l'être. Sur un système UNIX, l'utilisateur *root*. Sur les systèmes Windows c'est l'administrateur.

surcharge

Donner une nouvelle signification à un symbole ou à une construction. En fait, tous les langages pratiquent la surcharge à un degré ou un autre, car les gens savent distinguer les significations d'après le *contexte*.

surcharge d'opérateur

Un type de *surcharge* que l'on peut effectuer sur les *opérateurs* internes pour les faire fonctionner (de manière syntaxique) sur des *objets* comme s'il s'agissait des valeurs scalaires ordinaires, mais les règles sémantiques étant fournies par la classe. Ceci est activé par le *pragma* de surcharge– voir le chapitre 13.

SV

Abréviation de « scalar value ». Mais dans l'interpréteur Perl, tout *référent* est traité comme une sorte de *SV*, d'une certaine manière orientée objet. Toute *valeur* dans Perl est passée en tant que pointeur `SV*` en C. La `struct` SV connaît son propre type de *référent*, et le code est assez intelligent (espérons-le) pour ne pas essayer d'appeler une fonction de *hachage* sur un *sous-programme*.

symbole

En général, un *token* ou un *métasymbole*. Souvent utilisé pour désigner les noms que l'on trouve dans une *table de symboles*.

synchrone

Séquence d'événnements dont l'ordre d'arrivée est déterminé.

syntaxe

Du grec « avec-arrangement ». Comment des éléments (en particulier des symboles) sont assemblés entre eux.

système d'exploitation

Un programme spécial qui tourne sur la machine elle-même et cache les détails bas niveau de gestion des *processus* et des *périphériques*. Également utilisé dans un sens plus général pour indiquer une culture de programmation particulière. Le sens général peut être employé à divers degrés de spécificité. à un extrême, on peut dire que toutes les versions d'UNIX et de ses clones sont le même système d'exploitation (ce qui énerve beaucoup de gens, surtout les avocats). À l'autre extrême, une version donnée du système d'exploitation d'un fournisseur donné diffère de n'importe quelle autre version de ce fournisseur ou d'un autre. Perl est beaucoup plus portable entre les systèmes d'exploitation que de nombreux autres langages. Voir également *architecture* et *plateforme*.

système de fichiers

Un ensemble de *répertoires* et de *fichiers* résidant sur une partie du disque. Aussi appelé « partition ». Vous pouvez changer le nom d'un fichier ou bien le déplacer dans l'arborescence du système sans déplacer le fichier lui-même, du moins sur le système UNIX.

table de hachage

Une méthode utilisée en interne par Perl pour implémenter les tableaux associatifs (hachages) de manière efficace.

table de symboles

Là où un *compilateur* conserve ses symboles. Un programme comme Perl doit, d'une manière ou d'une autre, se souvenir de tous les noms de *variables*, de *handles de fichiers* et de *sous-programmes* qu'il a utilisé. Il place à cet effet les noms dans une table de symboles, qui est implémentée en Perl par une *table de hachage*. Il existe une table de symboles séparée pour chaque *paquetage*, qui donne à chaque *paquetage* son propre *espace de noms*.

tableau

Séquence de *valeurs* ordonnées de façon à pouvoir y accéder par *indice entier* qui indique la valeur du *déplacement*.

tableau associatif

Voir *hash*, svp.

tableau multidimensionnel

Un tableau avec plusieurs indices pour trouver un élément donné. Perl utilise des *références* à cet effet ; voir le chapitre 9, *Structures de données*.

tampon

Un lieu de stockage temporaire des données. Le *tamponnage par bloc* signifie que les données sont passées à leur destination quand le tampon est plein. Le *tamponnage par ligne* signifie qu'elles sont passées chaque fois qu'une ligne complète est reçue. Le *tamponnage de commandes* signifie qu'elles sont passées après chaque commande `print` (ou équivalente). Si la sortie n'est pas bufferisée, le système procède octet par octet sans utilisation d'une zone temporaire, ce qui est moins efficace.

tampon par bloc

Une méthode efficiente de tamponnage d'entrées/sorties qui utilise un *bloc* de données d'un coup. Perl l'utlise par défaut pour l'accès au fichiers sur le disque. Voir *tampon* et *tampon de commande*.

tampon par commande

Un mécanisme en Perl qui vous permet de stocker la sortie d'une commande, et la vide immédiatement dans la même requête au système d'exploitation. Il est activé en positionnant la variable `$|` (`$AUTOFLUSH`) à une valeur vraie. C'est utilisé lorsque vous ne souhaitez pas que vos données soient là sans rien faire à attendre l'ordre du départ, ce qui peut se produire car, par défaut, un *fichier* ou un tube utilisent un *tampon par bloc*.

tamponnage de ligne

Utilisé par un flux *d'E/S standard* de sortie qui vide (flush) ses *tampons* (buffers) après chaque *saut de ligne*. De nombreuses bibliothèques d'E/S standard activent ainsi automatiquement la sortie allant au terminal.

TCP

Abréviation de « Transmission Control Protocol ». Un protocole au-dessus de *IP*, faisant apparaître un mécanisme de transmission de paquets non fiable comme un flux d'octets fiable (enfin, en théorie).

terme

Court-circuit pour « terminal » c'est-à-dire la feuille d'un *arbre syntaxique*, et qui a le rôle d'un opérande pour les opérateurs de l'expression.

terminateur

Un *caractère* ou une *chaîne* qui indique la fin d'une autre chaîne. La variable `$/` contient la chaîne qui termine l'opération `readline`, ce que la fonction `chomp` enlève de la fin. à ne pas confondre avec les *délimiteurs* ou les *séparateurs*. Le point à la fin de cette phrase est un terminateur.

ternaire

Un *opérateur* prenant trois *opérandes*.

test de la sous-classe vide

Le fait qu'une *classe dérivée* vide doit se comporter exactement comme sa *classe de base*.

texte

Normalement, une *chaîne* ou un *fichier* contenant des caractères imprimables.

thread

Comme un processus dédoublé, mais sans la protection mémoire de l'instruction `fork`. Un thread est plus léger, au sens que les threads peuvent être exécutés ensemble à l'intérieur du même processus en se disputant la même mémoire, à moins de prendre des précautions pour les isoler les uns des autres. Voir le chapitre 17, *Threads*.

tie

Le lien entre une variable magique et sa classe d'implémentation. Voir la fonction `tie` au chapitre 29 et au chapitre 14, *Variables liées*.

TMTOWTDI

« There's More Than One Way To Do It », la devise Perl. L'idée qu'il peut y avoir plusieurs façons de procéder pour résoudre un problème de programmation. (Cela ne veut pas dire que le fait d'avoir plusieurs possibilités est toujours une bonne chose, ouque toutes les possibilités sont aussi bonnes les unes que les autres. Cela veut juste dire qu'il n'y a pas de vérité unique.

token

Un morphème en langage de programmation : la plus petite unité de texte qui possède un sens.

tokenisation

Éclater un programme en mots et en symboles séparés, chacun étant appelé un token (symbole). Également appelé « *lexicaliser* » , auquel cas l'on obtient des « lexèmes » .

tokeniseur

Un module qui sépare le programme en séquences de *tokens* pour l'analyseur syntaxique.

traitement d'exception

La manière dont un programme réagit à une erreur. Le mécanisme de traitement d'exception de Perl est la construction `eval`.

tranche

Une partie des *éléments* d'une *liste*, d'un *tableau* ou d'un *hachage*.

translittération

Transformation d'une chaîne vers une autre par correspondance de chaque caractère vers un autre avec une fonction. Voir l'opérateur `tr///` dans le chapitre 5.

transtypage

Conversion explicite de données d'un type à l'autre. C autorise ceci. Perl n'en a pas besoin. Et n'en veut pas.

troff

Un langage de mise en forme duquel Perl dérive sa variable `$%` secrètement utilisé

dans la production des livres de la série du Chameau.

tronquer

Vider un fichier de son contenu, soit automatiquement en l'ouvrant pour écriture ou explicitement par la fonction `truncate`.

true

Toute valeur scalaire qui ne vaut pas pas 0 ou `""` s'évalue à true dans un contexte booléen.

tube

Une *connexion* directe entre l'entrée d'un *processus* et la sortie d'un autre sans fichier intermédiaire. Une fois qu'il est initialisé, les deux processus concernés peuvent lire et écrire dessus comme si c'était un fichier banal, avec quelques précautions.

tube nommé

Un *tube* avec un nom encapsulé dans le *système de fichiers* de façon à pouvoir être utilisé comme un fichier par le *processus* lecteur, ce qui rend la programmation indépendante des données sources car il suffit de considérer la source comme un fichier.

type

Voir *type de données* et *classe*.

type de données

Un ensemble de valeurs possibles, avec toutes les opérations qui savent les gérer. Par exemple, un type de données numérique comprend un certain ensemble de valeurs sur lesquelles il est possible de travailler, ainsi que diverses opérations mathématiques associées qui n'auraient que peu de sens sur, par exemple, une chaîne comme « argh » . Les chaînes ont leurs opérations propres, comme la *concaténation*. Les types composés d'un certain nombre d'éléments plus petits ont généralement des opérations permettant de les composer et de les décomposer, et peut-être de les réarranger. Les *objets* qui modélisent les choses du monde réel ont souvent des opérations qui correspondent à des activités réelles. Par exemple, si l'on modélise un ascenseur, l'objet en question peut avoir une *méthode* `ouvrir_porte()`.

typedef

La définition d'un type en langage C.

typeglob

Emploi d'un identificateur unique (par exemple `*nom`) pour désigner `$nom`, `@nom`, `%nom`, `&nom` ou simplement `nom`. Son mode d'utilisation détermine s'il est interprété comme tous ces derniers, ou comme un seul d'entre eux. Voir *Typeglobs et handles de fichiers* dans le chapitre 2.

typemap

Dans un module *extension* écrit en *XS* décrit comment transformer un type C vers un type Perl ou réciproquement.

UDP

« User Datagram Protocol », la façon typique d'envoyer des *datagrammes* sur internet.

UID

Un identifiant d'utilisateur. Souvent employé dans le contexte de la propriété d'un *fichier* ou d'un *processus*.

umask

Le masque de *bits d'autorisation* pour créer des fichiers ou des répertoires pour déterminer à qui vous en refusez l'accès. Voir la fonction `umask`.

Unicode

Un ensemble de caractères qui contient tous les caractères du monde, plus ou moins. Voir *http://www.unicode.org*.

unité de compilation

Le *fichier* (ou la *chaîne* dans le cas d'un *eval*) dans le processus de compilation courant.

Unix

Un grand langage en constante évolution avec plusieurs syntaxes largement incompatibles, avec lequel n'importe qui peut définir n'importe quoi n'importe comment, ce dont peu se privent. Les UNIXophones le croient facile à apprendre car on peut facilement le tordre à sa convenance, mais les différences entre les dialectes rendent les communications intertribales quasiment impossibles, et les voyageurs en

sont souvent réduits à utiliser une version petit-nègre du langage. Pour être universellement compris, un programmeur shell UNIX doit étudier son art pendant des années. Nombreux sont ceux qui ont abandonné cette discipline et communiquent à présent *via* une sorte d'espéranto appelé Perl. Dans les temps anciens, UNIX représentait également du code que quelques personnes des Bell Labs avaient écrit pour utiliser un ordinateur PDP-7 qui ne faisait rien d'autre à ce moment-là.

v-chaîne

Une *chaîne* « vecteur » ou « version » spécifiée avec `v` suivie par une série d'entiers décimaux en notation pointée, par exemple, `v1.20.300.4000`. Chaque nombre devient un *caractère* avec sa valeur ordinale. (Le `v` est optionel quand il y a au moins trois caractères.)

valeur

Une donnée réelle, par constraste avec les variables, références, clefs, index, opérateurs ou toute chose pour accéder à une valeur.

valeur de liste

Une liste sans nom de valeurs scalaires temporaires qui peuvent être passées dans un programme par une fonction renvoyant une liste à une expression qui fournit un *contexte de liste*.

valeur de retour

La *valeur* produite par un *sous-programme* ou une *expression* à l'évaluation. En Perl, une valeur de retour doit être soit une *liste*, soit un *scalaire*.

valeur scalaire

Une valeur qui se trouve être *scalaire* par opposition à une *liste*.

variable

Un emplacement de mémoire qui porte un nom et qui peut contenir diverses *valeurs*, quand votre programme en ressent le besoin.

variable d'environnement

Mécanisme par lequel un agent de haut niveau, tel un utilisateur, peut transmettre l'état de ses variables de travail à un *processus* fils. Chaque variable d'environnement a la forme d'une paire *clef/valeur* comme dans un *hachage*.

variable d'instance

L'*attribut* d'un *objet* ; donnée mémorisée par l'instance de l'objet et non par la classe entière.

variable lexicale

Une *variable* ayant une *portée lexicale*, déclarée par `my`. Souvent juste appelé une « lexicale ». (La déclaration `our` déclare un nom à portée lexicale pour une variable globale, qui n'est pas une variable lexicale.)

variable scalaire

Une *variable* préfixée par un `$`, contenant une valeur unique.

variables magiques

Variables spéciales donnant des effets secondaires quand on y accède en lecture ou en écriture. Par exemple, en Perl, la modification des éléments du hachage `%ENV` modifie également les variables d'environnement correspondantes que les sous-processus utilisent. La lecture de la variable `$!` donne le numéro ou le message d'erreur UNIX courant.

variable statique

Rien de tel. Utilisez plutôt une *variable lexicale* pour une portée plus grande que votre *sous-programme*.

variadique

Se dit d'une *fonction* qui reçoit avec bonheur un nombre indéterminé d'*arguments réels*.

vecteur

Jargon mathématique pour désigner une liste de *valeurs scalaires*.

vidage

Le vidage d'un *tampon*, souvent avant qu'il ne soit rempli.

virtuel

Ce qui donne l'illusion de quelque chose qui n'a pas d'existence réelle comme par

exemple la mémoire virtuelle qui simule la mémoire réelle (voir aussi *mémoire*). Le mot inverse est « transparent » qui donne une existence réelle à quelque chose qui n'apparaît pas comme la façon dont Perl gère l'encodage variable UTF-8.

WYSIWYG

« What You See Is What You Get » (Vous obtenez ce que vous voulez). Utilisé pour indiquer quelque chose qui s'affiche comme il est écrit, comme par exemple les déclarations Perl `format`. C'est aussi le contraire de la magie parce que chaque chose apparaît comme elle est réellement, comme dans la fonction `open` à 3 arguments.

XS

eXternal Subroutine, extraordinairement exportée, expéditivement excellente, expressément exécutée en C ou C++ existant, ou dans un nouveau et excitant langage d'extension appelé (de façon exaspérante) *XS*. Examinez le chapitre 21, *Mécanismes internes et accès externes*, pour une explication exacte.

XSUB

Un *sous-programme* externe défini dans *XS*.

yacc

« Yet Another Compiler Compiler ». Un générateur d'*analyseur syntaxique* sans lequel Perl n'aurait pas existé. Voir le fichier *perly.y* dans le source de la distribution Perl.

zombie

Un processus qui est mort mais dont le père n'a pas été informé par la fonction `wait` ou `waitpid`. Si vous utilisez `fork`, vous devez ensuite nettoyer la mémoire avant de sortir sinon la table des processus va se remplir et risque de déborder ce qui risque de fâcher l'administrateur système.

Index

Symboles

Nombres

A

B

C

D

E

F

G

H

I

J

K

L

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

À propos des auteurs

Larry Wall est l'inventeur du langage Perl. « Associé » de O'Reilly & Associated, Larry est aussi fort célèbre pour avoir écrit les logiciels gratuits les plus populaires du monde UNIX, et notamment le lecteur de news *rn*, l'incontournable programme *patch*. Il est également connu pour *metaconfig*, un programme qui écrit des scripts *Configure*, et pour le jeu de guerre de l'espace *warp*, dont la première version a été écrite en BASIC/PLUS à la Seattle Pacific University. Larry est en fait linguiste de formation, il a traîné entre l'U.C. Berkeley et l'U.C.L.A. pendant ses études (assez curieusement, il n'avait rien à voir avec les développements UNIX en cours pendant qu'il se trouvait à Berkeley).

Les années suivantes le virent chez Unisys, JPL, Netlabs et Seagate, où il joua avec des choses aussi diverses que des simulateurs d'événements discrets ou des systèmes de gestion de réseau, sans oublier quelques fusées (il lui arrive aussi de jouer avec ses quatre enfants de temps en temps, mais ils gagnent trop souvent). C'est à Unisys, où il essayait d'assembler un système transcontinental de gestion de configuration sur une ligne chiffrée à 1 200 bauds à partir d'une version trafiquée de Netnews, que naquit Perl.

Tom Christiansen est consultant indépendant spécialisé dans la formation et la programmation Perl. Après avoir travaillé pendant plusieurs années chez TSR Hobbies (célèbre pour Dungeons and Dragons), il repartit au collège en passant un an en Espagne et cinq en Amérique, s'occupant de musique, de linguistique, de programmation et d'une demi-douzaine de langages parlés. Tom repartit de UW-Madison avec des BA d'espagnol et d'informatique, ainsi qu'un MS d'informatique.

Il passa ensuite cinq ans chez Convex comme électron libre, travaillant tant sur l'administration système que sur le développement d'utilitaires et de noyaux, en ajoutant le support utilisateur et la formation pour faire bonne mesure. Tom a également passé deux mandats au Board of Directors de la USENIX Association.

Tom est impliqué dans le développement de Perl depuis 1987. Il est l'auteur de *The Perl Cookbook*, (Perl en action) et co-auteur de *Learning Perl*, (Introduction à Perl) et de *Learning Perl on Win32 Systems*.

Tom présente des séminaires annuels basés sur plus de quinze ans d'expérience d'administration système et de programmation UNIX. Vivant dans les collines de Boulder, Colorado, environné de daims, de mouffettes et à l'occasion de pumas et d'ours bruns, il passe tous les étés à faire des excursions, tailler des arbres, élever des oiseaux, faire de la musique et chasser.

Jon Orwant est le CTO d'O'Reilly & Associates et il est éditeur en chef de *The Perl Journal*. Il est aussi co-auteur de *Mastering Algorithms with Perl* (O'Reilly) et auteur de *Perl 5 Interactive Course* (Macmillan). Avant de rejoindre O'Reilly, il était membre du groupe groupe d'édition électronique à MIT Media Lab. Il a alors effectué des recherches sur le comportement d'utilisateurs, l'automatisation du jeu de programmation et des nouvelles et divertissements personnalisés générés par ordinateur. Jon travaille également avec les comités consultatifs de VerticalSearch.com, Focalex, Inc. et YourCompass, inc.

Jon participe souvent à des conférences, parlant de manifestations comme celles de programmeurs ou de journalistes. Il prend plaisir à écrire du code comme de la prose et ses trois vices sont le jeu, le vin et les mathématiques. Il a également créé le premier jeu sur l'Internet.

À propos des traducteurs

Philippe Bruhat, consultant spécialisé en sécurité des réseaux, programme en Perl depuis 1996. Il fait partie des Mongueurs de Perl depuis les tous débuts du groupe parisien en 1999 et est vice-président de l'association éponyme fondée en 2001. Il est bien connu dans la communauté Perl pour ses exercices de programmation assombrie (qu'il considère comme une forme d'expression artistique).

Kai Carver, ingénieur logiciel basé aux États-Unis et en France, travaille dans les domaines de l'informatique linguistique, d'Internet et des bases de données. Mongueur Perl parisien, il collabore régulièrement avec O'Reilly.

Gérald Sédrati a travaillé pendant 6 ans chez Bull dans le développement de logiciels concernant l'administration de systèmes, réseaux et applications. Actuellement, ses talents sont employés par Xylème, startup issue de chercheurs de l'INRIA, qui développe et met en service un moteur de recherche reposant sur XML.

Il a commencé à pratiquer le développement en Perl il y a près de cinq ans qui, depuis, est devenu son langage informatique de prédilection ; il en apprécie énormément, entre autres, le souci linguistique dans lequel il a été conçu.

Gérald est très attaché à la philosophie du Logiciel Libre qui rejoint quelques-unes de ses autres préoccupations extra-informatiques comme l'engagement auprès d'associations comme Attac qui lutte pour qu'une « autre mondialisation soit possible ».

Il a également publié un recueil de poèmes, intitulé « Embryon ».