Table des matières

Liste des tableaux

Table des figures

Première partie

Présentation de Perl

Chapitre 1

Introduction

Perl est un langage de programmation initialement conçu pour le traitement de fichiers dans le but de produire des rapports. Il a été créé par un seul auteur, Larry Wall , qui depuis assure l'ensemble des développements liés à ce langage.

1.1 L'auteur (de Perl!)

Larry Wall est entre autre l'auteur du lecteur de news **rn**, de l'utilitaire **patch**[1] et du jeu **warp. Perl** trouve son origine vers 1986 dans un projet de maintenance de code. La nécessité de produire des rapports à partir des fichiers journaliers et les limitations de **awk** , en particulier l'impossibilité de manipuler plusieurs fichiers simultanément conduisent alors Lary Wall à développer un nouvel interpréteur: **Perl** , **P**ractical **E**xtraction and **R**eport **L**angage[2]. Cette première version était encore très primitive (le manuel ne comptait guère que 15 pages...), mais était déja plus performant que **sed** ou **awk** .

1.2 Les versions

Perl a rapidement rencontré un grand succès, dû en partie à l'existence de versions sur la plupart des plateformes: diverses saveurs de **Unix** , **MS-DOS**, **Mac OS**, **VMS**, **Atari**, ... mais aussi à l'étendue de ses possibilités. Bien que la version courante de **Perl** soit la version 5, la plupart des scripts que l'on trouve sont encore écrits avec la version 4. Ce petit fascicule est essentiellement consacré à cette version 4, les principales différences seront indiquées en appendice.

1. La légende veut que **patch** ait été crée pour gérer la complexité des mises à jour de **rn**
2. Une autre version est: **P**athologically **E**clectic **R**ubbish **L**ister

1.3 Le langage

Perl est un interpréteur (avec toutefois une pré-compilation des expressions régulières). Il présente en gros les possibilités des divers shells (**sh**, **csh** et leurs divers dialectes), du langage de traitement de listes **awk** , de l'utilitaire de recherche de chaînes de caractères **grep** , de l'éditeur **sed** et du langage **C** . **Perl** présente une syntaxe relativement simple (dans le sens ou le nombre de mots clés est assez limité) et offre des possibilités très étendues pour la manipulation de textes ou de fichiers, la production de rapports. La gestion de processus concurents est aussi très simple, ainsi que la communication entre ces processus, que ce soit par un mécanisme de "sockets" BSD ou par les librairies "IPC" de System V (queue de messages, mémoire partagée et sémaphores). Contrairement aux autres utilitaires, **Perl** ne présente en général pas de limitations arbitraires sur la longueur des lignes, la présence de données binaires, etc... Comparé aux programmes écrits en langage **C** , on trouve les mêmes possibilités, en particulier une bibliothèque complète d'appels système, mais on bénéficie en plus des avantages d'un interpréteur: facilité de mise au point et facilité de gérer des actions complexes. Pour toutes ces raisons, **Perl** a été adopté par les administrateurs de systèmes **Unix** pour l'écriture de leurs programmes de gestion du système[3]. Plus récemment, avec le développement du "Wolrd Wide Web", **Perl** est devenu un des langages favori pour l'écriture des programmes "CGI". Pour terminer sur la "philosophie" de **Perl** , il est à noter qu'il n'existe en général pas "une" manière officielle d'écrire un programme, mais plusieurs qui présentent chacune des avantages (essentiellement liés à la lisibilité du code). Mais comme avec tous les langages qui présentent une certaine souplesse (ici il s'agit même d'une souplesse certaine!), il est très facile d'écrire des programmes illisibles avec **Perl** , ou pire, des programmes dont une version remplit les objectifs fixés, mais sans que plus personne ne sache pourquoi!

1.4 Où trouver Perl

Une caractéristique de **Perl** est d'être disponible gratuitement pour la plupart des plateformes existantes. On peut donc se le procurer par "ftp anonyme" sur un grand nombre de sites. En France, on peut le trouver sur *ftp.ibp.fr* entre autres. Une fois le fichier transféré, l'installation est particulièrement simple, surtout sur les plateformes **Unix** . La génération du fichier *Makefile* est quasiment automatique, les informations sur le système étant collectées automatiquement par un script de configuration (l'exécution de ce script est une mine d'information précieuse sur votre système...). Les binaires générés sont en général installés dans /usr/local/bin, des sources **Perl** fournissant des extensions au langage sont placées dans /usr/local/lib/perl et les manuels sont installés dans les

3. **Perl** est aussi appelé le "Swiss-Army chainsaw of Unix programming"

répertoires /usr/local/man/man*x*. En plus du langage proprement dit, la distribution comprend quelques utilitaires tels que: **a2p** (un convertisseur de programmes **awk** en programmes **Perl**), **find2perl** (qui convertit des commandes **find** en programmes Perl), **c2ph** et **h2ph** qui convertissent des structures du langage **C** en structures pour **Perl**).

1.5 La documentation

La première source d'information sur **Perl** est le manuel **Unix** [1]. Pour la version 4, cela représente déjà plus de cent pages, quand à la version 5, le manuel représente près de 400 pages. C'est toutefois la seule documentation pour cette version. L'auteur de **Perl** a, en collaboration avec R. L. Schwartz, publié un ouvrage remarquable sur le langage [2]. Bien que limité à la version 4, ce livre est le compagnon indispensable du programmeur **Perl** . A noter qu'un petit fascicule résumant les mots clés du langage est fourni avec. Une version limitée de cet aide-mémoire est d'ailleurs fournie sous forme PostScript avec la distribution de **Perl** . Un autre livre écrit par R. L. Schwartz [3] fournit une bonne introduction. Parmi les groupes de News, **comp.lang.perl** est toujours une source précieuse d'information. Enfin, la distribution de **Perl** contient dans les sources du langage la réponse aux questions les plus pointues...

1.6 Cette présentation

Le but de ce petit fascicule est de fournir une première approche du langage. La première partie décrit successivement la forme d'un programme **Perl** , les divers types de variables (variables simples, listes et vecteurs associatifs), les structures de contrôle, les entrées/sorties puis les appels système et autres fonctions. Deux chapitres seront ensuite consacrés aux expressions régulières et aux méthodes de mise au point des programmes. Une deuxième partie présente des petits programmes **Perl** commentés, correspondant soit à des tâches simples, soit à de véritables utilitaires. Il est à noter que cette présentation est très axée sur l'implémentation **Unix** de **Perl** .
En aucun cas, elle ne saurait remplacer ni le manuel, ni le livre de Larry Wall.

Chapitre 2

Structure d'un programme Perl

Dans ce chapitre, nous commencerons par un petit programme (un grand classique!) pour voir comment se présente un programme écrit en **Perl** . Les variables scalaires de **Perl** seront ensuite présentées, ainsi que les opérateurs qui agissent dessus. Nous terminerons par quelques lignes de programmes pour nous familiariser avec les "coutumes" de **Perl** .

2.1 Un premier programme Perl

Pour ne pas déroger à la tradition, commençons pas le classique "hello world!":

```
#!/usr/local/bin/perl
print "Bonjour!\n";
```

Classiquement, la première ligne du programme indique, à la suite du "nombre magique" `#!`, l'interpréteur à utiliser pour le script qui suit. La ligne suivante utilise simplement la fonction **print** pour afficher la chaîne `"Bonjour!\n"`. La chaîne est délimitée par des guillemets et est terminée par le saut de ligne `\n` . Une chaîne de caractères peut aussi être délimitée par des apostrophes, auquel cas il n'y a pas de substitution des variables qu'elle contient éventuellement (c'est le même comportement qu'avec un shell). Ainsi, la ligne:

```
print 'Bonjour!\n';
```

n'affiche pas le saut de ligne, mais les caractères `\n` à la place. Enfin, la ligne doit, comme en **C** , être terminée par un point-virgule. Les chaînes de caractères peuvent comporter d'autres caractères spéciaux , tels que:

- `\r` un retour chariot;
- `\t` une tabulation;
- `\f` un saut de page;

`\b` le caractère de correction ("backspace");

`\0nnn` le caractère dont le code est 0nnn en octal;

`\xnn` le caractère dont le code est nn en hexadécimal.

Il est à noter que **print** est en fait une fonction et que l'on pourrait écrire:

```
print( "bonjour!\n" );
```

En fait, l'argument de **print** est une liste, et les formes suivantes sont toutes autant valides:

```
print( "Bonjour!", "\n" );
print "Bonjour!", "\n";
```

C'est une règle très générale dans **Perl** , l'argument des fonctions est une liste qui peut être écrite avec ou sans parenthèses , tant qu'il n'y a pas d'ambiguïté. Par exemple, la ligne suivante donnerait un résultat erroné:

```
print (2+4)*3, "\n";
```

Elle affiche "6" sans saut de ligne puisque l'interpréteur considère que la parenthèse de droite termine la liste! Pour obtenir le bon résultat, il faut utiliser une des deux formes:

```
print +(2+4)*3, "\n";
print ( (2+4)*3, "\n" );
```

Dans la première forme, on utilise l'opérateur `+` dont la raison d'être est de lever ce genre d'ambiguïté syntaxique. Nous verrons au Chapitre 7 que **print** a une forme plus générale pour écrire sur un autre fichier que la sortie standard.

2.2 Les variables scalaires

Perl utilise trois types de variables:

- les variables scalaires ("scalar");
- les listes ("array");
- les variables associatives ("hash array").

Le nom d'une variable scalaire commence par un `$`, celui d'une liste par un `@` et celui d'une variable associative par un `%`. De plus, les noms des sous-programmes commencent par un `&`, les descripteurs de fichiers n'ont pas de caractère spécial et l'on peut définir des étiquettes ("label") en faisant suivre leur nom de `:` . Pour

améliorer la lisibilité, il est d'usage d'utiliser des majuscules pour les descripteurs de fichiers et les étiquettes.

Les règles pour les noms de variables sont assez souples:

- comme en **C** , les minuscules et majuscules sont distinguées, donc **`$UN`**, **`$Un`** et **`$un`** représentent trois variables scalaires différentes;
- si le premier caractère du nom est une lettre (sachant que le signe `_` est une lettre!), on peut utiliser n'importe quel lettre ou chiffre à la suite.
- si ce n'est pas une lettre, le nom est limité à un seul caractère supplémantaire. En fait, beaucoup de variables de ce type sont prédéfinies sous le nom de "variables spéciales" (voir le Chapitre 5).

Parmi les variables spéciales dont il vient d'être question, une est particulièrement importante: **`$_`** . Cette variable est en effet la variable par défaut pour toutes les lectures et les recherches ou substitutions de chaînes. De même, de nombreuses fonctions l'utilisent par défaut, par exemple **print** . Ainsi:

```
print;
```

est équivalent à:

```
print $_;
```

Chaque type de variable ayant son propre espace de nommage, on peut sans risque utiliser le même nom pour des variables scalaires, des listes ou des vecteurs associatifs, ils seront distingués par leur préfixe. Ainsi, les variables **`$deux`**, **`@deux`** et **`%deux`** peuvent coexister dans un programme, même si celui-ci comprend un sous-programme appelé par **`&deux`**. De même, les mots réservés du langage peuvent être utilisés dans les noms des variables, la distinction étant faite par le préfixe. Le contenu d'une variable scalaire peut être:

- un nombre: soit entier (`12345` par exemple), soit flottant (`123.45` par exemple), soit flottant en notation scientifique (`123E45` par exemple), soit octal (`0377` par exemple) ou encore hexadécimal (`0xFF` par exemple).
- une chaîne de caractères délimitée par des guillemets ou des apostrophes comme on l'a vu plus haut.
- le résultat d'une commande que l'on place entre des accents graves. Voir le Chapitre 2 de la deuxième partie.

Les affectations aux variables se font bien sûr par le signe `=`. Quand à la distinction entre nombres et chaînes de caractères, elle se fait suivant le contexte. Par exemple, les lignes suivantes:

```
$a = 0377;
```

```
$b = "0377";
$c = 1;
$d = $b + $c;
```

affectent le nombre octal `0377` (soit `255`) à la variable `$a`, la chaîne de caractères `0377` à la variable `$b`, le nombre `1` à la variable `$c` et le nombre `378` à la variable `$d`. En effet, le zéro en tête ne désigne un octal que dans un nombre écrit littéralement. La chaîne `"0377"` est évaluée dans un contexte numérique à la quatrième ligne et prend la valeur numérique `377`.

2.3 Les opérateurs sur les variables scalaires

Perl dispose d'une panoplie impressionnante d'opérateurs sur les variables scalaires. Ils peuvent bien sûr être combinés en utilisant des parenthèses.

2.3.1 Les opérateurs numériques

On trouve bien sûr l'addition (`+`) , la soustraction (`-`) , la multiplication (`*`) , la division (`/`) , le modulo (`%`) , l'exponentiation (`**`) , mais aussi les opérateurs d'auto-incrémentation (`++`) et d'auto-décrémentation (`--`) . En ce qui concerne ces deux derniers, ils peuvent être placés avant ou après la variable suivant que l'on souhaite que l'opération ait lieu avant ou après l'affectation. Ainsi:

```
$a = 10;
$b = ++$a;
```

donne `$a = 11, $b = 11`, alors que:

```
$a = 10;
$b = $a++;
```

donne `$a = 11, $b = 10`.

2.3.2 Les opérateurs logiques

Classiquement, avec Perl, la condition FAUX est représentée par 0 et la condition VRAI par n'importe quel autre nombre. Les opérateurs logiques sont les mêmes qu'en C: le AND (`&&`) , le OR (`||`) et la négation (`!`) .

2.3.3 Les opérateurs bit-à-bit

On trouve classiquement le AND (`&`) , le OR (`|`) et le XOR (`^`) ainsi que le complément (`~`) et les opérations de décalage à gauche (`<<`) ou à droite (`>>`).

2.3.4 Les fonctions mathématiques

On y trouve tout d'abord la partie entière (**int**) et la racine carrée (**sqrt**), le logarithme (**log**), l'exponentielle (**exp**), les fonctions trigonométriques sinus (**sin**) , cosinus (**cos**) et arc tangente (**atan2 ($x,$y)** qui renvoie l'arc tangente de `$x/$y`), la conversion d'un nombre hexadécimal (**hex**) , ou octal (**oct**) en un nombre décimal, un générateur de nombres aléatoires (**rand ($x)** qui renvoie un nombre entre `0` et `$x`) et son initiateur (**srand ($x)** qui initialise le germe à `$x`).

2.3.5 Les opérations sur les chaînes de caractères

La concaténation de chaînes se fait à l'aide de l'opérateur: `.` . Ainsi, le programme:

```
$a = "un";
$b = "deux";
$c = $a . "," . $b;
```

assigne à la variable `$c` la chaîne `"un,deux"`. Un opérateur plus original est l'opérateur de répétiton: `x` . Le programme:

```
print "+" x 80,  "\n";
```

affiche une ligne de 80 signes `+` et un saut de ligne (attention à ne pas oublier la virgule!).

Les autres opérations sont plus classiques: la longueur d'une chaîne (**length**), le sous-ensemble d'une chaîne (**substr ($s, $n, $l)** donne la sous-chaîne de `$s` commençant en `$n` et de longueur `$l`), la position d'une chaîne dans une autre (**index ($s, $t, $n)** donne la position de la chaîne `$t` dans la chaîne `$s` en commençant la recherche à partir de la position `$n` si ce paramètre est présent, de `0` sinon) et **rindex** qui effectue la même opération en partant de la fin de la chaîne. A noter que pour la fonction **substr** , la position peut être négative, indiquant ainsi que l'on compte à partir de la fin de la chaîne.

2.3.6 Les opérateurs d'affectation

En plus de l'affectation par le signe égal, on peut, comme en **C** , combiner une affectation et une opération. Par exemple:

```
$a += $b;
```

incrémente `$a` de `$b`. On obtient ainsi des opérateurs pour incrémenter (`+=`) , décrémenter (`-=`) , multiplier (`*=`) , diviser (`/=`) , exponentier (`**=`) , prendre un modulo (`%=`) , concaténer (`.=`) , répéter (`x=`) et finalement effectuer des opérations

bit-à-bit AND, OR, XOR et décalages à gauche et à droite (`%=` , `|=` , `^=` , `<<=` , `>>=`). Quelques exemples pour éclaicir l'usage de ces fonctions:

```
$a = 1;
$a x= 5;
$a *= 4;
```

La deuxième ligne remplace `$a` par une chaîne formée de cinq `1` et la troisième l'évalue dans un contexte numérique et la multiplie par `4`. A la fin, `$a` vaut donc `44444`.

2.3.7 Les opérateurs de recherche de chaîne

Les opérateurs de recherche de chaîne (`//`) , de substitution (`s`) et de translation (`tr`) (voir le Chapitre 9) utilisent par défaut la variable spéciale `$_` comme espace de travail. L'opérateur `=~` permet de les faire opérer sur une variable scalaire au lieu de `$_`. Par exemple:

```
$a = "undeuxtrois";
if( $a =~ /trois/ ) { print "vrai\n" } else { print "faux\n" }
```

affiche `vrai` puisque la recherche `$a =~ /trois/` est positive (la variable `$a` n'est pas affectée). Le programme:

```
$a = "undeuxtroisquatre";
$a =~ s/deux/trois/;
```

affecte à la variable `$a` la chaîne `"untroistroisquatre"` en substituant la chaîne `"trois"` à la chaîne `"deux"` et le programme:

```
$a = "AbcDEfg";
$a =~ tr/a-z/A-Z/;
```

affecte à la variable `$a` la chaîne `"ABCDEFG"` en transcrivant les minuscules (`a-z`) en majuscules (`A-Z`).

2.3.8 Les opérateurs de comparaison

Perl dispose de deux jeux d'opérateurs de comparaison:

- un pour les valeurs numériques;

- un pour les chaînes de caractères.

Numérique	Chaine	
`==`	`eq`	égal
`!=`	`ne`	différent
`>`	`gt`	supérieur
`>=`	`ge`	supérieur ou égal
`<`	`lt`	inférieur
`<=`	`le`	inférieur ou égal
`<=>`	`cmp`	différent, avec le signe

TAB. 2.1 - *Les opérateurs de comparaison*

On dispose ainsi de 14 opérateurs de comparaison qui sont résumés dans le Tableau 2.1.

Ces opérateurs donnent 1 si le test est positif, la chaîne vide (`''`) sinon, sauf pour `<=>` et `cmp` qui donnent -1 si le premier terme est inférieur au deuxième et +1 s'il est supérieur. En ce qui concerne la comparaison des chaînes de caractères, l'ordre est celui des codes `ASCII` et les éventuels espaces en fin de chaîne sont pris en compte.

2.3.9 Les tests sur les fichiers

La série d'opérateurs qui suit opère différents tests sur les fichiers. Par défaut, ils testent le fichier dont le nom est contenu dans la variable spéciale `$_`, mais ils peuvent aussi opérer sur un nom de fichier ou un descripteur de fichier. Par exemple:

```
print "le fichier .profile ";
if ( -e ".profile" ) { print "existe\n" }
else                 { print "n'existe pas\n" }
```

Après un de ces tests, on peut refaire un test sur le même fichier en utilisant le descripteur spécial `_` :

```
print ".profile est ";
print "lisible " if -r ".profile";
print "modifiable " if -w _;
print "executable " if -x _;
print "\n";
```

(noter au passage la manière de faire les tests! c'est une pratique très courante avec **Perl** ...).

Ces opérateurs donnent 1 si le test est positif, la chaîne vide sinon, sauf `-s` qui donne la taille en octets et `-M`, `-A` et `-C` qui donnent un nombre de jours. Ces différents tests sont décrits dans le Tableau 2.2

-r	le fichier est lisible par l'uid effectif du processus
-w	le fichier est modifiable par l'uid effectif du processus
-x	le fichier est exécutable par l'uid effectif du processus
-o	le fichier appartient à l'uid effectif du processus
-R	le fichier est lisible par l'uid réel du processus
-W	le fichier est modifiable par l'uid réel du processus
-X	le fichier est exécutable par l'uid réel du processus
-O	le fichier appartient à l'uid réel du processus
-e	le fichier existe
-z	le fichier a une taille nulle
-s	donne la taille du fichier en octets
-f	le fichier est un fichier disque
-d	le fichier est un répertoire
-l	le fichier est un lien symbolique
-p	le fichier est un "pipe" nommé (FIFO)
-S	le fichier est un socket
-b	le fichier est un fichier spécial de blocs
-c	le fichier est un fichier spécial de caractères
-u	le fichier a le bit "setuid" positionné
-g	le fichier a le bit "setgid" positionné
-k	le fichier a le bit "sticky" positionné
-t	le descripteur de fichier pointe sur un terminal
-T	le fichier est un fichier de texte
-B	le fichier est un fichier binaire
-M	nombre de jours depuis la dernière modification
-A	nombre de jours depuis le dernier accès
-C	nombre de jours depuis le dernier changement de l'inode

TAB. 2.2 - *Les opérateurs de test sur les fichiers*

A noter que les trois derniers tests donnent le nombre de jours depuis le début de l'exécution du programme. Cette valeur est exprimée en jour et est généralement fractionnaire. Ces tests remplacent l'appel système **stat** de **C**[1] et sont particulièrement utiles dans tous les programmes chargés du "nettoyage" dans les systèmes de fichiers.

1. Il existe cependant une fonction réalisant cet appel, voir le Chapitre 8.

2.4 Un autre programme

Assez de présentation, passons à un "vrai" programme!

```
#!/usr/local/bin/perl
while(<>) {   # tiens, une boucle!
      print if /bonjour/;
}
```

Il apporte quelques nouveautés. A la suite de la boucle **while** , le signe `#` indique que tout ce qui suit est un commentaire. Dans la boucle, on voit apparaître l'instruction de lecture `<>`. Utilisée ainsi, elle lit sur l'entrée standard `stdin`. Comme on l'a déjà dit, le résultat de cette lecture est placé dans la variable spéciale `$_`. Deux remarques à ce sujet: la lecture se fait ligne par ligne[2] et le contenu complet de la ligne est fourni, y compris un éventuel saut de ligne. Cette lecture du fichier alimente une boucle **while** qui s'exécute donc jusqu'à la fin du fichier. Comme toutes les structures de contrôle de **Perl** , le corps de la boucle est constitué d'un bloc délimité par des accolades `{ }`[3] Dans la boucle on teste la présence d'une chaîne de caractères (sous-entendu, dans la variable `$_`), et si ce test est positif, on imprime la ligne. Noter encore cette manière un peu particulière de faire les tests[4] *action if condition* plutôt que le traditionnel *if condition action* . L'ordre **print** étant sans argument, il utilise la variable `$_` par défaut. Le caractère de saut de ligne ayant été préservé, il n'y a pas à le rajouter! On obtient donc une sorte de **grep** qui filtre les lignes de son entrée standard et affiche celles qui contiennent "bonjour". Si on l'appelle "bgrep", on pourrait l'utiliser par:

```
cat hello | bgrep
```

En fait, si on tape:

```
bgrep hello
```

on obtient le même résultat! En effet, si la ligne de commande contient des noms de fichiers, ils sont successivement ouverts pour alimenter l'ordre de lecture par défaut `<>`. On peut donc utiliser ce petit programme pour rechercher dans toute une série de fichiers:

```
bgrep fichier_1 fichier_2 ... fichier_n
```

2. Ce comportement peut être modifié en changeant la variable spéciale $/, voir le Chapitre 5.

3. On verra au Chapitre 6 que, contrairement au langage **C** , ce bloc est requis pour toutes les structures de contrôle.

4. On peut bien sûr utiliser un ordre plus classique, mais c'est presque une tradition avec **Perl** .

Cette boucle simple permettant de balayer un ensemble de fichiers est souvent la raison d'être d'un programme **Perl**[5] et on peut la remplacer par une des options de l'appel de **Perl** :

```
#!/usr/local/bin/perl -n
print if /bonjour/;
```

L'option `-n` crée implicitement la boucle de lecture des fichiers cités dans la ligne d'invocation du script. Une autre option utile est `-e` qui permet de placer directement le script sur la ligne de commande. Notre programme devient:

```
perl -ne 'print if /bonjour/' fichier_1 fichier_2 ... fichier_n
```

Ne pas oublier les apostrophes qui encadrent le "corps" du programme! Pendant que l'on parle des options, en voici quelques autres:

-p entoure comme `-n` le programme d'une boucle, mais de plus imprime chaque ligne lue.

-c vérifie juste la syntaxe d'un programme sans l'exécuter.

-v affiche la version de **Perl** .

-w affiche des messages quand l'interpréteur rencontre des instructions suspectes[6].

Revenons à notre programme. Tel quel, il ne trouve pas les lignes qui contiennent `Bonjour` ou `BONJOUR`. Il suffit de rajouter un ordre de traduction:

```
perl -ne 'tr/A-Z/a-z/;print if /bonjour/' fichier_1 ...
```

pour corriger cela. Cette traduction se fait très simplement par la fonction `tr` avec comme arguments la liste des majuscules `A-Z` et la liste des minuscules `a-z`. A noter qu'alors les lignes affichées ont "bénéficié " de cette traduction et sont en minuscules. Un dernier raffinement: on peut afficher le nom du fichier dans lequel on a trouvé l'expression en utilisant la variable spéciale `$ARGV` qui contient le nom du fichier en cours:

```
perl -ne 'tr/A-Z/a-z/;print "$ARGV: $_" if /bonjour/' fichier_1 ...
```

Nous verrons au Chapitre 3 un programme de recherche et de comptabilisation de mots bien plus puissant.

5. Qui au départ a été conçu pour celà!
6. Larry Wall considère que cette option devrait être obligatoire!

Chapitre 3

Les listes

Dans le langage **Perl** , ces objets sont en fait nommés "array", mais les possibilités qu'ils offrent quand à leur manipulation les apparentent plus à des listes. C'est donc sous ce nom que nous les désignerons ici.

3.1 Création des listes

Comme on l'a vu précédemment, les listes sont désignées par un nom qui commence par le signe `@`. Elles contiennent des variables scalaires qui peuvent donc être soit numériques, soit des chaînes de caractères. On les construit en plaçant la liste des éléments qui la constituent, séparés par des virgules, entre parenthèses:

```
@bool = ( "true", "false" );
@jours = ( 0, 1, 2, 3, 4, 5, 6 );
```

A noter qu'une virgule supplémentaire est tolérée en fin de liste, ce qui est bien agréable quand on la présente comme cela:

```
@bool = (
   "true",
   "false",
);
```

On peut bien sûr placer dans la liste des variables scalaires:

```
$true = "TRUE";
$false = "FALSE";
$bool = ( $true, $false);
```

C'est l'occasion d'introduire un nouvel opérateur: l'intervalle `..`

```
@jours = ( 0 .. 6 );
```

génère la même liste que précédemment. On peut aussi générer des chaînes de caractères ainsi:

```
@letter = ( 'A' .. 'Z' );
@hex = ( '0' .. '9' , 'A' .. 'F' );
```

génère la liste des 26 lettres majuscules de l'alphabet et la liste des 16 codes hexadécimaux. Pour accéder aux éléments d'une liste, on utilise le signe `$` (le résultat est un scalaire!), le nom de la liste suivi du numéro de l'élémemt entouré par des crochets `[]`:

```
$jours[2]
```

vaut 2. Le numéro des éléments commence par 0, sauf si on a modifié la variable spéciale `$[` . Le numéro du dernier élément de la liste est donné par:

```
$#nom_de_la_liste
```

Si on place une liste dans la définition d'une autre liste, cela inclus tous les éléments de la première liste dans la seconde:

```
@un = ( 'un', 'deux' );
@deux = ( @un, 'trois' );
```

donne pour `@deux` la liste: `( 'un', 'deux', 'trois' )`. Si l'on affecte une liste à une variable scalaire, cela affecte le dernier élément de la liste à la variable scalaire:

```
@un = ( 'un', 'deux' );
$deux = @un;
```

donne: `$deux = 'deux'`. En utilisant des crochets, on peut aussi extraire un sous-ensemble d'une liste:

```
@jours = ( 0 .. 6 );
@test = @jours[ 3, 2, 4 ];
```

donne `@test = ( 3, 2, 4 )`. Enfin, on peut faire des affectations entre listes:

```
@un = ( 'un', 'deux', 'trois' );
( $i, $j ) = @un;
```

donne `$i = 'un'` et `$j = 'deux'`, mais aussi:

```
@un = ( 'un', 'deux', 'trois' );
( $i, @j ) = @un;
```

donne `$i = 'un'` et `@j = ( 'deux', trois' )`.

Une liste particulière est la liste `@ARGV` dont les éléments sont les paramètres qui ont été passés au script. Le programme:

```
#!/usr/local/bin/perl
foreach $i ( @ARGV ) { print "$i\n"; };
```

affiche simplement les arguments qu'on lui passe. On peut bien sûr accéder directement à ces paramètres en utilisant `$ARGV[0]`, `$ARGV[1]`,... et le numéro du dernier paramètre est donné par `$#ARGV`, le nombre de paramètres étant `$#ARGV+1`.

3.2 Fonctions sur les listes

3.2.1 Les listes utilisées comme des piles

Quatre fonctions permettent d'empiler ou de dépiler des éléments d'une liste:

- **shift** dépile un élément à gauche;
- **unshift** empile un élément à gauche;
- **pop** dépile un élément à droite;
- **push** empile un élément à droite;

Par exemple:

```
@list = ( 'un', 'deux', 'trois' );
$a = shift( @list );
```

donne:

```
$a = 'un'
@list = ( 'deux', trois' )
```

```
@list = ( 'un', 'deux', 'trois' );
$a = 'in';
unshift( @list, $a );
```

donne:

```
@list = ( 'in', 'un', 'deux', 'trois' )
```

```
@list = ( 'un', 'deux', 'trois' );
$a = pop( @list);
```

donne:

```
$a = 'trois'
@list = ( 'un', 'deux'' )
```

```
@list = ( 'un', 'deux', 'trois' );
$a = 'in';
push( @list, $a );
```

donne:

```
@list = ( 'un', 'deux', 'trois', 'in' ).
```

Par défaut, **shift** opère sur la liste `@ARGV` des arguments du programme, ou, si l'on est dans un sous-programme, sur la liste `@_` des paramètres.

3.2.2 La commande splice

La commande la plus générale pour manipuler une liste est **splice** . Sa syntaxe est:

```
splice( liste, i, n [, iliste ] );
```

Elle donne comme résultat les `n` éléments de `liste` à partir de l'élément d'indice `i` et, si le paramètre `iliste` est présent, insère cette liste à la place des éléments supprimés. Par exemple:

```
@list = ( 'un', 'deux', 'trois', 'quatre' );
@ilist = ( 1, 2, 3 );
@nlist = splice( @list, 1, 2, @ilist );
```

donne:

```
@list = ( 'un',  1, 2, 3, 'quatre' )
@nlist =  'deux', 'trois' )
```

3.2.3 Les tris

Perl propose deux fonctions pour trier des listes:

- **sort** donne une liste triée en utilisant l'ordre des codes ASCII sans modifier la liste initiale.

- **reverse** donne une liste dont les éléments sont en ordre inverse de ceux de la liste initiale. La liste initiale n'est pas modifiée.

Par exemple:

```
@list = ( 'un', 'deux', 'trois', 'quatre' );
@slist = sort @list;
@rlist = reverse @list;
```

donne:

```
@slist = ( 'deux', 'quatre', 'trois', 'un' )
rlist = ( 'quatre', 'trois', 'deux', 'un' )
```

On peut aussi fournir à **sort** sa propre fonction de comparaison avec la syntaxe:

```
sort subroutine list
```

Les paramètres sont passés à la fonction dans les variables `$a` et `$b`. La fonction doit retourner un nombre inférieur, égal ou supérieur à zéro suivant que `$a` est inférieur, égal ou supérieur à `$b`. Par exemple, on peut faire un tri numérique avec le programme:

```
sub numerically { $a <=> $b; }
$sortedbynumber = sort numerically @list;
```

Un autre exemple sera donné au Chapitre 6 en faisant un tri suivant des clés.

3.2.4 La commande grep

La fonction **grep** crée une liste contenant les éléments d'une liste pour lesquels une expression est vraie avec la syntaxe:

```
grep( expression, list )
```

Si par exemple une liste contient comme éléments les lignes d'un fichier, on peut créer une nouvelle liste contenant les lignes qui contiennent une certaine expression. En anticipant un peu sur la suite:

```
open( PASSWD, "</etc/passwd" );              # open passwd file
@list = <PASSWD>;                            # pump it into list
@tcsh = grep( /\/usr\/bin\/tcsh/ , @list ); # grep tcsh entries
foreach $1 (@tcsh) {                         # loop on new list
   print $1;                                 # print them
}
```

affiche les entrées dans le fichier `/etc/passwd` des utilisateurs qui utilisent le shell tcsh. Noter au passage la manière plutôt concise de lire le fichier dans une liste!

3.2.5 Les conversions scalaire/liste

Perl propose deux fonctions pour transformer une chaîne en liste et réciproquement: **split** découpe une chaîne en éléments suivant une liste de séparateurs. La syntaxe en est:

```
split( /pattern/, variable, limit );
```

où `/pattern/` définit les séparateurs, `variable` la chaîne à découper et `limit` le nombre maximum d'éléments à retourner. Tous les paramètres sont facultatifs: la chaîne traitée par défaut est `$_` et le délimiteur par défaut est tout ce qui ressemble à un espace, soit en termes d'expressions régulières: `/[ \t\n]+/` (voir le Chapitre 9). L'exemple précédent peut être modifié en:

```
open( PASSWD, "</etc/passwd" );              # open passwd file
@list = <PASSWD>;                            # pump it into list
@tcsh = grep( /\/usr\/bin\/tcsh/ , @list ); # grep tcsh entries
foreach $1 (@tcsh) {                         # loop on new list
   ( $user, $rest ) = split( /:/, $1, 2 );   # extract user field
   print $user, "\n";                        # print it
}
```

La commande **split** découpe les champs des entrées (ils sont séparés par des :) en limitant le nombre de champ à deux. On obtient donc une liste de deux éléments: le champ user et le reste de l'entrée. Le programme affiche donc le logon des utilisateurs qui utilisent le shell tcsh. **join** fait le contraire de **split** : il génère une chaîne de caractères en concaténant les éléments d'une liste avec le séparateur spécifié. La syntaxe est:

```
join( separator, list )
```

Par exemple:

```
@addr = ( '134', '158', '69', '66' );
$addr = join( '.', @addr );
```

donne: `$addr = '134.158.69.66'`.

Chapitre 4

Les vecteurs associatifs

Les vecteurs associatifs sont une des structures les plus puissantes du langage **Perl** qu'il a emprunté au langage **awk** .

4.1 Définition des vecteurs associatifs

Les vecteurs associatifs sont des vecteurs dont les éléments, au lieu d'être identifiés par des indices, sont identifiés par une chaîne de caractères associée (d'où leur nom). Cette chaîne de caractères est appellée la clé et l'élément correspondant la valeur. Ils sont particulièrement utiles dans un programme qui collecte des données arbitraires pour les stocker. Par exemple, un programme qui traite les statistiques d'un routeur ip ne peut pas prévoir un vecteur pour toutes les adresses (il y en a beaucoup trop!). Par contre, un vecteur associatif dont l'indice est la paire d'adresse ip (les statistiques donnent le nombre d'octets échangés entre deux adresses) convient parfaitement: il est construit dynamiquement et croît au fur et à mesure que des nouvelles paires d'adresses se présentent. C'est aussi la structure idéale pour stocker une petite base de données dans un programme. Les éléments de ces vecteurs sont accédés par une technique de "hashing", ce qui a deux conséquences: l'accès est très rapide et si on explore un vecteur, les éléments sont fournis dans un ordre apparemment aléatoire.

En **Perl** , les vecteurs associatifs sont distingués par un `%` devant leur nom. On peut remplir un vecteur en lui fournissant une liste de paires (clé - valeur). Par exemple:

```
%colors = (
   "red",   "rouge",
   "green", "vert",
   "blue",  "bleu",
);
```

Ici, les couleurs en Anglais sont les clés, les couleurs en Français sont les valeurs. Comme pour les listes, on peut mettre une virgule excédentaire en fin de liste.

Pour accéder aux éléments, on donne la clé entourée par des accolades `{}`. Ainsi, `$colors{"red"}` est la chaîne de caractères `"rouge"`. Il faut bien comprendre que ces structures ne demandent aucune initialisation: si la clé a déjà été utilisée, l'élément existe et on peut l'utiliser comme n'importe quel scalaire, si la clé n'existe pas encore, l'élément correspondant est automatiquement crée. Dans l'exemple précédent:

```
$colors{ "yellow" } = "jaune";
```

crée un quatrième élément dont la clé est `"yellow"` et la valeur `"jaune"`.

4.2 Fonctions sur les vecteurs associatifs

La première fonction qui sert pour les vecteurs associatifs est **keys** qui fournit une liste de toutes les clés du vecteur. Dans l'exemple précédent:

```
@list = keys( %colors );
```

donne:

```
@list = ( "yellow", "red", "blue", "green" );
```

Notez que l'ordre n'est pas celui dans lequel le vecteur a été créé. Il est cependant facile de la trier avec **sort** . Pour explorer tous les éléments d'un vecteur associatif, on utilise souvent une boucle dans le genre:

```
foreach $i (sort keys(%colors)) {
   print "$i\n";
}
```

Notez que les parenthèses sont facultatives pour beaucoup de fonctions et ont ici été omises pour la fonction **sort** .

Le pendant de **keys** est la fonction **values** qui donne l'ensemble des valeurs des éléments du vecteur. En reprenant notre exemple:

```
@list = values( %colors );
```

donne:

```
@list = ( "jaune", "rouge", "bleu", "vert" )
```

Pour obtenir successivement tous les éléments d'un vecteur associatif, on peut aussi utiliser la fonction **each** . Elle fournit à chaque appel une liste constituée de la clé suivante et de la valeur correspondante. Quand le vecteur est entièrement exploré, elle retourne une liste vide:

```
while ( ( $key, $value ) = each( %colors ) ) {
   print "$key \t: $value\n";
}
```

produit le résultat:

```
yellow  : jaune
red     : rouge
blue    : bleu
green   : vert
```

Quand on arrive au dernier élément, **each** donne un vecteur vide qui produit une valeur fausse pour l'affectation à la liste, ce qui arrête la boucle.

Enfin, pour supprimer un élément d'un vecteur associatif, on utilise la fonction **delete** qui supprime l'élément associé à une clé et retoune la valeur correspondante.

```
$last = delete $colors{"yellow"};
```

supprime l'entrée `"yellow"` du vecteur `%colors` et assigne à `$last` la valeur `"jaune"`.

4.3 La simulation de vecteurs multi-dimensionnels

Perl ne définit pas véritablement de vecteur multi-dimensionnel, mais les vecteurs associatifs permettent de les simuler. Il suffit d'utiliser des clés formées de la juxtaposition des indices. On pourrait par exemple écrire:

```
$matrix{ "1, 1" } = 123;
```

mais il y a plus simple. La forme:

```
$matrix{ 1, 1 } = 123;
```

est interprétée par **Perl** comme un raccourci pour:

```
$matrix{ join( $;, 1, 1 ) };
```

Il réalise donc automatiquement la juxtaposition des indices avec comme séparateur ce qui est contenu dans la variable spéciale `$;`. Cette variable est par défaut le caractère dont le code ASCII est 28 ("Field Separator"). En fait, rien n'oblige à ce que les indices soient numériques, on peut parfaitement écrire:

```
$matrix{ "un", "un" } = 123;
```

mais cela présente peu d'intérêt...

4.4 Un exemple

Un exemple typique d'utilisation de ces structures est la constitution d'une petite base de données à partir d'un fichier. Par exemple, on peut constituer une base donnant pour chaque logon du fichier `/etc/passwd` le uid correspondant:

```
open( PASSWD, "</etc/passwd" );
@list = <PASSWD>;
foreach (@list) {
   split(/:/ );
   $uid{ $_[0] } = $_[2];
   push( @group, $_[0] );
}
```

On lit le fichier dans la liste `@list` comme on l'à déjà vu, puis on explore cette liste. Chaque ligne est découpée en champ par un **split** avec le séparateur : et on remplit le vecteur associatif `uid` dont les éléments sont les logons (premier champ) et les clés les uid (troisième champ). Au passage, on crée aussi une liste des logons en les empilant dans la liste `@group`. En aurait tout aussi bien pu faire:

```
@group = keys( %uid );
```

après la boucle, le résultat est (presque[1]) le même. Le vecteur associatif `%uid` peut maintenant être utilisé par exemple comme clé de tri, comme on le verra au Chapitre 6.

1. Il y a quand même une différence: dans le premier cas, s'il y a des entrées dupliquées, elles apparaîtront, dans le deuxième cas, on ne les verra pas!

Chapitre 5

Les variables spéciales

Perl utilise un nombre important de variables spéciales, dotées d'un nom "cryptique"[1]. Une sélection de ces variables spéciales est présentée ici. Pour plus de détails, se reporter à [1] ou [2].

Ces variables peuvent être classées en trois groupes:

- les variables spéciales globales;
- les variables spéciales locales;
- les variables spéciales pour les descripteurs de fichiers.

5.1 Les variables spéciales globales

Elles déterminent le comportement général de **Perl** et peuvent être utilisées pour l'adapter à un usage particulier. Certaines fournissent un accès simple à certains appels système tels que **getpid** , **setuid** , **signal** , ...

`$_` C'est la variable par défaut pour les lectures et la recherche de chaînes de caractères.

`$.` Le numéro de la dernière ligne lue lors de la dernière lecture.

`$/` Le caractère de séparation des enregistrements en lecture. Peut être mis à `"\n\n"` ou à la chaîne vide `""` pour lire paragraphe par paragraphe.

`$\` Le caractère ajouté après chaque **print** . Par défaut, rien, mais peut être mis à `"\n"` pour obtenir un comportement similaire à celui de **awk** .

`$,` Le caractère qui sépare les champs dans la fonction **print** . Par défaut, rien.

1. La version 5 de **Perl** permet d'utiliser des noms plus descriptifs (mais aussi plus longs à saisir...), voir Appendice A à ce sujet

`$"` Comme le précédent, mais à l'intérieur des chaînes de caractères. C'est par défaut un espace.

`$#` Le format d'impression des nombres pour la fonction **print** . Par défaut, **%.20g**.

`$$` Le numéro du processus exécutant le script.

`$?` Le statut rendu par la dernière commande externe (voir le Chapitre 2 de la Partie II). Les huit bits de poids fort donnent la valeur rendue par **exit** , les huit bits de poids faible le numéro du signal qui a interrompu le processus. Cette variable est utilisée après un appel à **wait** .

`$*` Mis à 0 (par défaut), indique à **Perl** que les chaînes de caractères contiennent une seule ligne dans un but d'optimisation. Mis à 1, indique qu'une chaîne peut comporter plusieurs lignes.

`$0` Contient le nom du fichier en cours d'interprétation (donc le nom sous lequel le processus apparaît). Peut être changé pour que le processus apparaîsse sous un nom déterminé dans la commande **ps** .

`$[` Contient le numéro du premier élément des vecteurs (0 par défaut).

`$]` Donne la version et le niveau de patch de **Perl** .

`$;` Le caractère qui sépare les indices dans la simulation des vecteurs multidimensionnels (voir le Chapitre 4).

`$!` Dans un contexte numérique, la valeur courante de **errno** , dans un contexte alphanumérique, le message d'erreur correspondant.

`$@` Le dernier message d'erreur généré par **Perl** dans une évaluation externe (**eval** , **require** ou **do**).

`$<` Le **uid** réel du processus. Permet d'effectuer les appels système **getreuid** et **setreuid** .

`$>` Le **uid** effectif du processus. Permet d'effectuer les appels système **getuid** et **setuid** .

`$(` Le **gid** réel du processus et éventuellement la liste des groupes autorisés. Permet d'effectuer les appels système **getgroups** , **getgid** et **setgid** .

`$)` Le **gid** effectif du processus et éventuellement la liste des groupes autorisés. Permet d'effectuer les appels système **getgroups** ,**getegid** et **setegid** .

`$^D` La valeur de l'indicateur de debug (-D). Sa valeur peut être modifiée en cours d'exécution (voir le Chapitre 10).

`$^T` La date et l'heure à laquelle l'exécution du script a commencé.

`$^W` La valeur de l'indicateur de warning (-w).

`ARGV` Cette variable est un descripteur de fichier qui pointe successivement sur tous les fichiers énumérés sur la ligne de commande (voir le Chapitre 7). C'est le descripteur sur lequel on lit par défaut quand on utilise `<>`.

`$ARGV` Contient le nom du fichier en cours quand on lit des fichiers successifs depuis `<ARGV>`.

`@ARGV` La liste qui contient les arguments de la ligne de commande. Attention, contrairement aux shells, `$ARGV[0]` contient le premier argument, pas le nom du script, celui-ci se trouve dans `$0`. De même, `$#ARGV` contient le nombre d'arguments `-1`.

`DATA` Le descripteur de fichier qui pointe sur ce qui suit la ligne **__END__** dans le script. Permet de placer des données dans le script lui-même (voir le Chapitre 7).

`@INC` Le vecteur qui contient la liste des répertoires où chercher les fichiers à inclure (par **do** ou **require**).

`%INC` Le vecteur associatif qui contient pour chaque fichier inclus par **do** ou **require** le nom complet du fichier retenu.

`%ENV` Le vecteur associatif qui contient les valeurs des variables d'environnement du processus.

`%SIG` Le vecteur associatif qui contient les routines de traitement des différents signaux. Sa modification est équivalente à l'appel **signal** en **C** . Les valeurs `'DEFAULT'` et `'IGNORE'` sont valides.

`STDERR` Le descripteur de fichier pointant sur la sortie d'erreur standard.

`STDIN` Le descripteur de fichier pointant sur l'entrée standard.

`STDOUT` Le descripteur de fichier pointant sur la sortie standard.

5.2 Les variables spéciales locales

Ces variables ont une portée limitée au bloc courant et sont toutes liées à la dernière recherche de chaîne de caractères (voir le Chapitre 9).

`\$1..\$9` Contiennent, après une recherche de chaînes de caractères, les sous-chaînes correspondantes aux diverses paires de parenthèses de la chaîne de recherche.

$& Contient la sous-chaîne de caractères qui remplit le critère de la dernière recherche.

$` Après une recherche de chaînes de caractères, contient la partie de la chaîne initiale qui précède la chaîne trouvée.

$' Après une recherche de chaînes de caractères, contient la partie de la chaîne initiale qui suit la chaîne trouvée.

$+ Contient, après une recherche de chaîne de caractères, la dernière des variables `$1..$9` qui a été affectée.

Par exemple, la recherche:

```
/\/(.*)\/(.*)\//;
```

effectuée sur la chaîne:

```
$_ = "/usr/local/bin/perl";
```

Donne comme résultat:

```
$1 = "usr/local"            # premiere chaine entre //
$2 = "bin"                  # derniere chaine entre //
$+ = "bin"                  # derniere variable $i
$& = "/usr/local/bin/"      # l'ensemble de la chaine trouvee
$` = ""                     # ce qui la precede
$' = "perl"                 # ce qui reste apres
```

5.3 Les variables spéciales liées aux descripteurs de fichiers

Chaque descripteur de fichier a un jeu de variables affecté. Ces variables reflètent les valeurs correspondant au descripteur de fichier sélectionné par **select** .

Une utilisation typique est la suivante:

```
select((select(FILE), $= = 44) [0]);
```

Un brin d'explication? `select(FILE)` sélectionne le descripteur `FILE` et retourne le descripteur précédemment sélectionné. `$= = 44` détermine une longueur de page de 44 lignes pour le descripteur sélectionné (FILE). Le résultat de ces deux commandes est placé dans une liste de deux éléments dont le premier est extrait par `(...)[0]`. Le résultat est l'ancien descripteur qui est passé à **select** pour

rétablir la sélection originale. Autrement dit, on modifie le comportement du descripteur `FILE` sans changer le descripteur sélectionné.

$| Quand cette variable a une valeur non nulle, elle impose un flush après chaque écriture par **write** ou **print** . Elle sert en particulier quand le descripteur auquel elle est attachée est un "pipe" ("hot pipe"). Elle vaut 0 par défaut.

$% C'est le numéro de la page courante pour un descripteur utilisé en écriture.

$= C'est le nombre de lignes par pages pour un descripteur utilisé en écriture.

$- C'est le nombre de lignes restantes à imprimer pour un descripteur utilisé en écriture. `$- = 0` provoque un saut de page.

$~ C'est le nom du **format** utilisé pour l'impression d'un rapport.

$^ C'est le nom du **format** utilisé pour les en-têtes de page pour l'impression d'un rapport.

Chapitre 6

Les structures de contrôle

Perl propose une panoplie impressionnante de structures de contrôle. On y trouve toutes les structures des autres langages, avec en plus quelques instructions plus exotiques.

6.1 If...Else...

La forme de **if** est des plus classiques:

```
if ( condition ) {
   action
}
```

avec les variations **else** :

```
if ( condition ) {
   action
} else {
   action
}
```

ou **elif**:

```
if ( condition ) {
   action
} elif ( condition ) {
   action
...
} else {
   action
}
```

Contrairement au langage **C** , le bloc délimité par les accolades est requis. La ligne suivante est incorrecte:

```
if( $i > 0 ) print "$i\n";
```

il faut l'écrire:

```
if( $i > 0 ) { print "$i\n }
```

if a une variation: **unless** .

```
unless ( condition )
```

est équivalent à:

```
if( !condition )
```

if et **unless** admettent toutefois une forme plus originale où ils sont considérés comme des modificateurs placés en fin de ligne:

```
action if condition;
action unless condition;
```

Cela permet d'avoir un code plus lisible en écrivant par exemple:

```
print "$i\n" if $i > 0;
```

On trouve aussi cette construction bien pratique du langage **C**:

```
condition ? case1 : case2
```

qui prend la valeur `case1` si la condition est vraie, `case2` sinon. Par exemple:

```
print "bit 2 is ", ($a & 0x04) ? "set" : "not set", "\n";
```

on sélectionne la chaîne à imprimer en fonction du test directement dans la fonction **print** .

6.2 Les boucles

Commençons par la boucle **while**:

```
LABEL: while ( condition ) {
   action
}
```

Le `LABEL:` est facultatif (il sert pour les ruptures de boucle) et il peut y avoir en plus une clause **continue** :

```
LABEL:
while ( condition ) {
   action
} continue {
   action
}
```

auquel cas le block qui suit **continue** est systématiquement exécuté avant le test.

On peut remplacer **while** par **until** auquel cas, le test est inversé (mais effectué en début de boucle).

De la même manière, on a une boucle **for** :

```
LABEL:
for ( initial; test; next ) {
   action
}
```

`initial` est exécuté avant la première itération, `action` est effectuée tant que la condition `test` est vraie et `next` est effectué à la fin de chaque itération. `LABEL:` est facultatif comme pour **while** .

Une variation est la boucle **foreach** qui permet de balayer une liste:

```
LABEL:
foreach variable ( liste ) {
   action
}
```

La variable prend successivement la valeur des éléments de la liste et `action` est effectuée. Si une variable n'est pas précisée, c'est la variable **$_** qui est utilisée. Par exemple:

```
foreach (1..10) { print "$_\n" }
```

affiche les nombres de 1 à 10. Comme pour **if** et **unless** , **while** et **until** peuvent aussi être utilisés comme des modificateurs, ce qui donne des boucles:

```
action while condition;
action until condition;
```

Cela permet d'écrire par exemple:

```
print $i++, "\n" while i <= 10;
```

Une variation de cette forme peut être faite avec **do** :

```
do {
   action
} until condition;
```

6.3 Les ruptures de boucle

Trois commandes permettent d'interrompre l'exécution du bloc d'une boucle: **next** , **last** et **redo** . Suivies d'un label, elles interrompent la boucle indiquée, sinon, la boucle la plus interne.

- **next** passe à l'itération suivante de la boucle (c'est similaire au **continue** du langage **C**);

- **last** arrête l'exécution de la boucle (c'est similaire au **next** du langage **C**);

- **redo** recommence l'itération courante.

Deux fonctions réalisent une rupture beaucoup plus brutale:

- **exit** interrompt le programme avec comme code de retour l'argument qu'on lui a fourni;

- **die** interrompt le programme en affichant la liste qu'on lui passe en paramètre et comme code de retour celui donné par la dernière erreur (dans la variable spéciale **$?**). C'est la manière normale de sortir après une erreur fatale. Par exemple:

  ```
  die "Could't change directory" unless chdir "/usr/spool/news";
  ```

6.4 Il y a même un GOTO!

Mais oui, **goto** fonctionne, mais il ne faut pas l'utiliser!

6.5 Les tests logiques

Les tests logiques `||` et `&&` ont un emploi bien particulier en **Perl** dans le sens où au lieu de donner vrai ou faux, ils donnent la dernière valeur qu'ils ont évaluée. Cela permet des raccourcis dans les expressions conditionnelles. Par exemple:

```
$arg = shift || 'a.out';
```

assigne à `$arg` le premier élément de `$_` s'il y en a un, ou `a.out` s'il n'y en a pas. C'est un moyen commode d'affecter une valeur par défaut. Cela sert aussi beaucoup pour garder les commandes qui, si elles échouent, constituent des erreurs fatales, comme:

```
chdir "/usr/local/news" || die "Could't change directory";
```

6.6 La simulation de Case

Il n'y a pas de construction réservée pour **case** dans **Perl** , mais les fonctions existantes permettent plusieurs variantes basées sur le fait qu'un bloc par lui-même est considéré comme une boucle qui s'exécute une seule fois. En utilisant **last** , on peut donc sortir quand on a trouvé la condition remplie. Une possibilité est:

```
CASE: {
   test_1 && do { action_1 ; last CASE };
   ...
   test_n && do { action_n ; last CASE };
   action_default;
}
```

par exemple:

```
CHARTYPE: {
    /^[a-zA-Z]$/ && do { print "a letter\n"; last CHARTYPE };
    /^[0-9]$/    && do { print "a number\n"; last CHARTYPE };
   print "other\n";
}
```

6.7 Les sous-programmes

Les sous-programmes peuvent se trouver n'importe où dans le code. Ce sont simplement des blocs précédés de **sub**:

```
sub NAME {
```

```
   action
}
```

Ils sont invoqués en faisant précéder leur nom de `&`. Leurs arguments leurs sont passés par la liste locale `@_` , donc dans les variables `$_[0]`, `$_[1]`...). La valeur qu'ils retournent est la valeur de la dernière expression qu'ils ont évaluée, mais on peut aussi retourner une liste par **return** . Il n'y a pas de distinction entre fonction et sous-programme: ces structures retournent toujours une valeur, si elle est utilisée, c'est une fonction, sinon, un sous-programme...

La manière dont sont traitées les variables mérite quelques explications:

- les variables externes au sous-programme ne sont pas protégées! Aussi il est FORTEMENT recommandé de ne travailler que sur des variables locales. Celles-ci peuvent être créées par la fonction **local** qui génère une liste dont la portée est limitée au bloc courant.

- les paramètres sont passés par reférence. Il est donc tout aussi recommandé d'en faire une copie locale par **local** .

- si on passe une liste en paramètre, c'est l'ensemble des éléments qui est passé au sous-programme. Si l'on veut passer une liste (ou un vecteur associatif) pour que le sous-programme puisse la modifier (la trier par exemple), il faut passer son nom. Cela se fait en préfixant son nom par `*` . Cette notation désigne tous les types de variables (`$`, `@` et `%`).

- il est toutefois possible de créer des ensembles de sous-programmes qui utilisent des espaces de variables indépendants par la commande **package** . Par défaut, un script se déroule dans l'espace **main** . L'instruction **package** commute dans le nouvel espace de nommage jusqu'au prochain changement. En général, on la place en début d'un fichier constitué d'une librairie de sous-programmes que l'on utilisera par **require** :

  ```
  #!/usr/local/bin/perl
  package private;
  sub sub1 {
     bloc
  }
  sub sub2 {
     bloc
  }
  ```

  Dans un **package** , on peut quand même accéder aux variables des autres espaces! Il suffit de faire précéder le nom de la variable par le nom du package

et une apostrophe, par exemple, dans le **package** `private` précédent, on pourrait accéder à la variable `$i` de l'espace **main** par: `$main'i`[1].

La fonction **caller** fournit l'endroit d'où un sous-programme a été appelé sous la forme d'une liste contenant le nom du **package** , le nom du fichier en cours d'exécution et le numéro de la ligne.

Enfin, les sous-programmes peuvent être appelés récursivement.

Quelques exemples pour préciser:

```
sub test {
   local ( $a, $b ) = @_;
   ( $pack, $file, $linenum ) = caller;
   print "called from package: $pack, file: $file, line:$linenum\n";
   $i++;
   $a + $b;
}
$i = 1;
print "$i\n";
print &test ( 2, 4 ), "\n";
print "$i\n";
```

donne comme résultat:

```
1
called from package: main, file: t1, line:11
6
2
```

dans le sous-programme, on travaille sur des copies locales (`$a` et `$b`) des deux paramètres. La variable externe `$i` EST modifiée! La dernière expression évaluée est la somme des paramètres qui est le résultat.

Autre exemple, on veut multiplier par deux tous les éléments d'une liste:

```
sub dbllist {
   local ( *list ) = @_;
   foreach $i ( @list ) {
      $i *= 2;
   }
   print "$list\n";
   $list = 456;
}
```

1. Remarquez que l'on n'est plus très loin des concepts des langages orientés objet!

```
$t = 123;
@t = ( 1, 2, 3, 4, 5, );
&dbllist( *t);
print '@t = ( ';
foreach $i ( @t ) { print "$i ,"; }
print ")\n";
print "$t\n";
```

donne:

```
123
@t = ( 2 ,4 ,6 ,8 ,10 ,)
456
```

quand on appelle `&dbllist` avec le paramètre `*t`, on passe les noms des variables `$t`, `%t` et `@t` (mais on ne se servira pas de tous). Dans le sous-programme, **local** assigne les noms des variables locales `$list` à `$t`, `%list` à `%t` et `@list` à `@t`. C'est donc chaque élément de `%t` qui est doublé et la variable `$t` qui est affichée. Quand `$list` est modifiée dans le sous-programme, celà se reflète sur `$t`.

Un autre exemple: un sous-programme qui additionne deux listes:

```
sub addlist {
   local( *a, *b ) = @_;
   local( $max ) = $#a > $#b ? $#a : $#b;
   local( @c );
   for( local($i) = 0; $i <= $max; $i++ ) {
      $c[$i] = $a[$i] + $b[$i];
   }
   @c;
}
$i = 123;
print "i: $i\n";
@l1 = ( 1, 3, 5, 7, 9, );
@l2 = ( 2, 4, 6, );
@l3 = &addlist( *l1, *l2 );
print "l3 = ( ";
foreach $i ( @l3 ) { print "$i, "; }
print ")\n";
print "i: $i\n";
```

donne:

```
i: 123
l3 = ( 3, 7, 11, 7, 9, )
i: 123
```

Ici, on passe au sous-programme une liste de deux éléments: les noms des variables (`$%@`)l1 et (`$%@`)l2. Dans le sous-programme, le premier **local** fait pointer `@a` sur `@l1` et `@b` sur `@l2`. On crée une variable locale `$max` qui est le numéro du dernier élément de la liste la plus longue et une liste locale `@c`. Il ne reste plus qu'à faire une boucle pour additionner les éléments (noter toutefois que l'on prend la précaution d'utiliser un indice de boucle `$i` local, de sorte que la variable globale `$i` n'est pas modifiée).

On peut illustrer la récursivité par l'exemple classique des "Tours de Hanoï":

```
#!/usr/local/bin/perl
sub hanoi {
   local ( $n, $i, $j ) = @_;
   if( $n == 0) { return };
   local( $k ) = 6 - $i - $j;
   &hanoi( $n-1, $i, $k );
   print "move from: $i to $j\n";
   &hanoi( $n-1, $k, $j );
   return;
}

&hanoi( $ARGV[0], 1, 3 );
```

Pour terminer, une fonction de comparaison pour la commande **sort** : si la liste `@group` comprend une liste de logons et le vecteur associatif `%uid` contient les uid de chaque logon, on peut avoir:

```
sub byuid {
   $uid{$a} <=> $uid{$b};
}
@sortedbyuid = sort byuid @group;
```

après quoi, la liste `@sortedbyuid` contient la liste des comptes triés par uid.

6.8 L'appel de modules externes

On a vu que l'on pouvait construire des librairies de sous-programmes travaillant dans leur propre espace de variables par l'instruction **package** . Ces fichiers sont nommés avec une extension `.pl` par convention. Pour les utiliser dans un programme, on utilise l'instruction **require** qui exécute le contenu du fichier, rendant ainsi accessible les sous-programmes qui s'y trouvent. L'instruction **require** recherche les fichiers dans les répertoires dont la liste se trouve dans la variable spéciale `@INC` . On peut avoir à la modifier pour y ajouter ses propres répertoires si l'on utilise des librairies privées.

Perl est livré avec plusieurs de ces librairies:

abbrev.pl fournit toutes les abréviations non ambiguës d'une liste de mots.

bigint.pl permet de faire du calcul en entier avec une précision arbitraire.

bigfloat.pl permet de faire du calcul en flottant avec une précision arbitraire.

bigrat.pl permet de faire du calcul sur les fractions avec une précision arbitraire.

cacheout.pl permet d'ouvrir simultanément plus de fichiers que le système ne le permet.

complete.pl fournit (comme `tcsh` quand on tape `<escape>`) le nom de fichier qui complète le mieux la chaîne fournie.

ctime.pl fournit les dates sous une forme lisible.

dumpvar.pl affiche les variables d'un **package** .

find.pl fournit une simulation de la commande **find** .

finddepth.pl comme la précédente, mais avec un ordre de recherche différent.

flush.pl effectue un **flush** sur un fichier.

getopt.pl implémente **getopt** comme en **C** .

getopts.pl comme le précédent, mais avec des regroupements d'options.

importenv.pl récupère les variables d'environnement.

look.pl effectue une recherche dans un fichier par comparaison à une clé.

pwd.pl permet de garder une variable d'environnement `PWD` à jour.

shellwords.pl découpe une ligne en mots de la même manière que le ferait un shell.

stat.pl définit les variables qui permettent d'utiliser la fonction **stat** .

syslog.pl effectue l'appel **syslog Unix** .

termcap met en œuvre les routines de la librairie **termcap** du **C** .

timelocal l'inverse de **localtime** .

validate effectue une série de test sur des fichiers.

Chapitre 7

Les entrées/sorties

7.1 L'accès aux fichiers

L'accès aux fichiers se fait par un descripteur ("filehandle") qui est une variable sans caractère particulier en préfixe. L'usage veut que ces descripteurs soient écrits en majuscules. Un descripteur est lié à un fichier par la commande **open** :

```
open( FILE , "/etc/passwd" );
```

ouvre le fichier `/etc/passwd` en lecture.

Par défaut, le fichier est ouvert en lecture. Si on veut préciser le mode, on fait précéder son nom de < pour la lecture, > pour l'écriture et >> pour ajouter à la fin d'un fichier. Si on ouvre en écriture ou en ajout un fichier qui n'existe pas, il est automatiquement créé.

```
open( TMPFILE, ">/tmp/prog$$" );
open( LOGFILE, ">>/tmp/log$$" );
```

ouvrent en écriture un fichier dont le nom est `prog` suivi du numéro du processus et ajoutent à la suite du fichier dont le nom est `log` suivi du numéro du processus, tout cela dans le répertoire `/tmp`.

L'accès en lecture/écriture se fait en précédent le nom du fichier par `+>` ou `+<`.

La commande **open** permet aussi de dupliquer un descripteur de fichiers, comme le font les appels **dup** et **dup2** de **Unix** . Il faut pour cela ajouter un `&` après le signe donnant le mode d'ouverture. C'est comme cela que sont effectuées les re-directions en shell:

```
open( NEWOUT, ">test.file" );
close( STDOUT );
open( STDOUT, ">&NEWOUT" );
system( "ls" );
```

on ouvre un descripteur `NEWOUT` sur le fichier `test.file` en écriture, on ferme la sortie standard et on duplique `NEWOUT` sur la sortie standard. La commande `ls` exécutée par **system** écrivant sur le descripteur `STDOUT`, elle écrit en fait dans le fichier `test.file`. On obtient la même chose qu'avec la commande:

```
ls > test.file
```

Pour dissocier un descripteur de fichier du fichier (et par la même, terminer toutes les opérations en cours, en particulier vider les tampons), on utilise bien sûr la fonction **close** sous la forme:

```
close( FILE );
```

Une fois le fichier ouvert, on peut le lire par `<>` ou y écrire par **print** ou **printf** (il existe des ordres **read** et **write** , mais le premier sert à faire des lectures sans structure de lignes et le second sert à générer des fichiers tabulés). Ainsi:

```
open( SERV, "/etc/services" );
while( <SERV> ) { print; };
```

ouvre le fichier `/etc/services` le lit ligne par ligne et affiche ces lignes à l'écran, comme le ferait la commande:

```
cat /etc/services
```

Par défaut, une ligne est délimitée par des sauts de lignes, mais ce comportement peut être changé en modifiant la variable spéciale `$/` .

La fonction **print** écrit sur le fichier décrit par son premier paramètre si ce n'est pas une liste. La liste des valeurs à imprimer suit. Attention à ne pas mettre de virgule entre le descripteur de fichier et la liste:

```
print FILE, "test\n";
```

est incorrect, le descripteur de fichier est pris comme faisant partie de la liste à imprimer!

On peut obtenir, comme en **C** , une sortie formattée par **printf** . Par exemple:

```
$a = 1.2345;
print "a = $a\n";                # with print
printf "a = %6.2f\n", $a;        # printf standard
printf "a = %06.2f\n", $a;       # zeroes padded
printf "a = %+6.2f\n", $a;       # force sign
printf "a = %-6.2f\n", $a;       # left align
printf "a = % -6.2f\n", $a;      # left aligne + sign place
```

affiche:

```
a = 1.2344999999999999
a =    1.23
a = 001.23
a =   +1.23
a = 1.23
a =   1.23
```

Les formats sont similaires à ceux du langage **C** , à savoir:

`%fm.nt`

où:

f est un ou plusieurs des "flags" suivants:

- **-** alignement à gauche;
- **+** force l'affichage du signe;
- *espace* ajoute un blanc à gauche s'il n'y a pas de signe;
- **0** bourrage à gauche par des zéros.

m est le nombre de caractères du format;

n est le nombre de chiffres après la virgule;

t est un des types qui sont décrits dans le Tableau 7.1

Comme dans le langage **C** , on a une fonction **sprintf** qui au lieu d'écrire le résultat formaté sur un fichier l'affecte à une variable. Par exemple:

```
$a = 64;
$a = sprintf( "0x%x", $a );
print "$a\n";
```

affiche:

```
0x40
```

Il n'y a pas de fonction analogue à **scanf**! Elle serait en effet inutile, les variables étant traitées suivant leur contexte, la lecture de données numériques se fait naturellement, ainsi que celle de données en hexadécimal ou en octal.

Par défaut, un script **Perl** a plusieurs descripteurs de fichiers ouverts:

– `STDIN` l'entrée standard;

s	Chaine de caractères
c	Caractère
d	Nombre décimal
ld	Nombre décimal long
u	Nombre non signé
lu	Nombre non signé long
x	Nombre hexadécimal
lx	Nombre hexadécimal long
o	Nombre octal
lo	Nombre octal long
e	Nombre flottant exponentiel
f	Nombre flottant virgule fixe
g	Nombre flottant

TAB. 7.1 - *Les formats de* **printf**

- `STDOUT` la sortie standard;
- `STDERR` la sortie d'erreur standard.

Un descripteur de fichier un peu particulier est `DATA` qui pointe sur la partie du script qui suit l'ordre `__END__` . Par exemple:

```
while ( <DATA> ) { print ;};

__END__
C'est ici que les donnees
de ce petit programme
seront lues!
```

affiche:

```
C'est ici que les donnees
de ce petit programme
seront lues!
```

Comme on l'a déjà vu, les lignes qui sont lues contiennent à la fin le caractère saut de ligne. C'est souvent pratique, mais on doit aussi souvent s'en débarrasser. On se sert donc souvent de la fonction **chop** qui supprime le dernier caractère d'une variable ou de la variable spéciale `$_` par défaut.

La fonction **eof** permet de tester si l'on a atteint la fin du fichier. On peut l'utiliser dans l'opérateur intervalle `..` , comme par exemple dans:

```
while(<>){ print if /__END__/ .. eof;};
```

Appliqué au listing du programme précédent, celui-ci affiche les lignes à partir de `__END__` jusqu'à la fin.

Si l'on désire lire un fichier sans notion d'enregistrements, on peut utiliser **read** :

```
read( FILE, buffer, n, delta );
```

essaye de lire `n` octets sur le fichier décrit par `FILE` et place le résultat dans la variable `buffer`. Si le quatrième paramètre est présent, l'écriture dans la variable commence à l'octet `delta`. Le résultat de la fonction est le nombre d'octets lus.

Pour certaines fonctions, on a besoin du numéro associé à un descripteur de fichier. On l'obtient par la fonction **fileno** .

Enfin, pour se positionner dans un fichier, on peut utiliser la fonction **seek** dont la syntaxe est:

```
seek( file, delta, start );
```

Cela déplace le "pointeur" dans le fichier de `delta` octets à partir:

- du début du fichier si `start` vaut 0;
- de la position courante si `start` vaut 1;
- de la fin du fichier si `start` vaut 2.

`delta` peut bien sûr être négatif dans les deux derniers cas.

7.2 L'exécution d'une commande par open

L'ordre **open** permet aussi d'exécuter une commande. Suivant le cas, ce que l'on écrira sur le descripteur sera passé (par un **pipe**) à la commande, ou ce que l'on lira sur le descripteur sera le résultat de la commande. La syntaxe est:

```
open( REF, "|commande" );
```

pour passer des données à une commande et:

```
open( REF, "commande|" );
```

pour lire le résultat d'une commande. Par exemple:

```
open( LS, "ls |" );
while(<LS>) { print ;};
```

exécute la commande `ls`, lit le résultat et l'affiche. Voir le Chapitre 2 de la deuxième partie pour plus de détails.

7.3 L'écriture de données tabulées

Perl possède des commandes très puissantes pour produire des rapports tabulés. Le principe consiste à décrire un format d'en-tête et un format de page pour un type de rapport. Ces descriptions sont faites à l'aide de la commande **format** . On y décrit des champs et les variables qu'ils doivent contenir. Chaque fois qu'on a un nouveau jeu de valeurs pour ces variables, il suffit d'appeler **write** et... **Perl** se charge de tout: il imprime la nouvelle ligne en effectuant s'il le faut un saut de page avec un nouvel en-tête. Plusieurs variables spéciales sont concernées par cette fonction:

- `$%` contient le numéro de la page courante;
- `$=` contient le nombre de lignes par page;
- `$-` contient le nombre de lignes restant à imprimer sur la page courante;
- `$~` contient le nom du format;
- `$^` contient le nom du format d'en-tête.

Le format utilisé sur un descripteur est par défaut celui qui porte le même nom que le descripteur. Celui d'en-tête est par défaut le nom du descripteur suivi de `_TOP`. On peut changer cela en changeant les variables `$~` et `$^` pour le descripteur donné (voir le Chapitre 5). L'ordre **write** s'effectue sur le descripteur qui a été choisi par **select** . Dans la commande **format** , on peut mettre:

- du texte en format libre;
- des descripteurs de champ sous la forme d'un `"@"` suivi d'autant de `"<"`, `">"` ou `"|"` que l'on veut de caractères. Les `"<"` indiquent un alignement à gauche, les `">"` un alignement à droite et les `"|"` des données entrées.
- des variables qui sont placées sous les descripteurs de champs.

le **format** est terminé par une ligne contenant juste un point. Par exemple:

```
#!/usr/local/bin/perl
require "ctime.pl";
format TEST_TOP =
Date: @<<<<<<<<<<                                       Page @>>
      $date                                                  $%

Nom                 Logon      uid    gid home
-----------------------------------------------------------------
.
format TEST =
```

```
@<<<<<<<<<<<<<<< @||||||||| @>>>>> @>>>>> @<<<<<<<<<<<<<<<<<<<<<<
$name            $logon     $uid   $gid   $home
.
$date = &ctime(time);
open(TEST, ">test" ) || die "Could't open test file";
open(PASSWD, "/etc/passwd") || die "Could't open passwd file";
select TEST;
while(<PASSWD>) {
   ($logon, $passwd, $uid, $gid, $name, $home, $shell ) =
                                                      split(/:/);
   write;
}
```

Dans ce programme, on commence par inclure `ctime.pl` par un **require** (on en aura besoin pour la date). On décrit ensuite le format de l'en-tête: il sera constitué d'une ligne contenant deux champs justifiés respectivement à gauche et à droite dans lesquels on placera les variables `$date` et `$&` (le numéro de page). On a ensuite une ligne vide, une ligne contenant le nom des colonnes et une ligne de séparation. On décrit ensuite le format des colonnes: un champ justifié à gauche pour le nom, un champ centré pour le logon, deux champs justifiés à droite pour les uid et gid et enfin un champ justifié à gauche pour le répertoire home. Ces formats sont appelés `TEST` et `TEST_TOP` car ils vont servir sur un descripteur nommé `TEST`. On obtient la date par le sous-programme `ctime` (dans la librairie `ctime.pl`). Puis on ouvre le fichier `test` en écriture et le fichier `/etc/passwd` en lecture. On fait un **select** sur le fichier `test` pour qu'il soit utilisé par **write** et il ne reste plus qu'à lire ligne par ligne le fichier `/etc/passwd`, à récupérer les champs par un **split** et à appeler **write**: tous les champs sont remplis avec les valeurs des variables correspondantes. A la fin, on trouve dans le fichier `test` quelque chose comme:

```
Date: Wed Jan 24                                          Page   1

Nom                 Logon      uid    gid home
-----------------------------------------------------------------
Wojciek Wojcik     wojcik      319    202 /home/ccin2p3/wojcik
Noel Giraud        giraud      309    201 /home/ccin2p3/giraud
Pierre Larrieu    larrieu      311    202 /home/ccin2p3/larrieu
```

7.4 Les entrées-sorties de bas niveau

Perl propose enfin deux fonctions permettant d'effectuer des entrées/sorties directes (similaires aux fonctions **read** et **write** du langage **C**):

```
sysread( FILE, buffer, n, delta );
```

essaye de lire `n` octets sur le fichier décrit par `FILE` et place le résultat dans la variable `buffer`. Si le quatrième paramètre est présent, l'écriture dans la variable commence à l'octet `delta`. Le résultat de la fonction est le nombre d'octets lus. Contrairement à **read** qui utilise la fonction **fread** , **sysread** fait directement un appel à **read** . De même:

```
syswrite( FILE, buffer, n, delta );
```

essaye d'écrire `n` octets de la variable `buffer` sur le fichier décrit par `FILE`. Si le quatrième paramètre est présent, les octets sont pris dans `buffer` à partir de l'octet `delta`. La fonction retourne le nombre d'octets effectivement écrits.

Ces fonctions peuvent servir par exemple pour manipuler des données sur bandes magnétiques (ou cartouches Exabyte, ou DLT,...).

Chapitre 8

Commandes diverses

Ce chapitre décrit les fonctions plus particulièrement liées à l'interface entre **Perl** et le système où il s'exécute. Certaines sont très dépendantes de ce système et n'existent que dans l'implémentation **Unix** de **Perl** (certaines ne sont même pas implémentées sur toutes les versions de **Unix**).

8.1 Les appels système

Perl offre une interface avec le système **Unix** comparable à celle du langage **C** . Souvent, les paramètres de ces fonctions sont plus simples que l'équivalent en **C** grâce à l'utilisation de listes au lieu de structures. Par exemple, la fonction **times**, qui donne le temps utilisateur et système consommé par un processus et ses fils, donne en retour une liste de quatre éléments donnant directement ces valeurs en secondes:

```
($user, $system, $cuser, $csystem) = times;
```

alors que l'équivalent en **C** rend une structure de type `tms` contenant ces valeurs, mais exprimées en fractions de secondes définies par la variable `CLK_TCK`, elle-même obtenue par un appel à la fonction `sysconf`! Par contre, certaines de ces fonctions utilisent en paramètres des structures similaires à celles du **C** et sont parfois peu documentées. Il est alors nécessaire de rechercher la forme de ces structures dans les fichiers d'en-tête (.h) correspondants pour les traduire en **Perl** avec les fonctions **pack** et **unpack** .

La suite de cette section est juste un catalogue de ces fonctions, classées par fonctionnallités.

– l'accès au système de fichiers:

chmod change les autorisations d'accès d'une liste de fichiers.

```
@list = ( 'file1', 'file2' );
chmod 0755, @list;
```

chown change le propriétaire et le groupe d'une liste de fichiers.

```
$uid = 309;
$gid = 203;
@list = ( 'file1', file2' );
chown $uid, $gid, @list;
```

fcntl examine ou modifie les propriétés d'un fichier ouvert.

```
$F_GETFL = 3; # get file flags
open( TEST, ">test" );
print fcntl( TEST, $F_GETFL, $flags ), "\n";
close TEST;
open( TEST, "+>test" );
print fcntl( TEST, $F_GETFL, $flags ), "\n";
close TEST;
open( TEST, "<test" );
print fcntl( TEST, $F_GETFL, $flags ), "\n";
```

affiche:

```
1
2
0 but true
```

on obtient les flags correspondant aux divers modes d'ouverture du fichier.

flock gère les verrouillages sur les fichiers.

```
$LOCK_SH = 1;   # read lock
$LOCK_EX = 2;   # write lock
$LOCK_NB = 4;   # poll
$LOCK_UN = 8;   # unlock
open( TEST, ">test" );
flock( TEST, $LOCK_EX);
sleep 60;
flock( TEST, $LOCK_UN);
```

verrouille le fichier `test` en écriture pendant 60 secondes. Un autre processus demandant un verrouillage en écriture pendant ce temps restera bloqué jusqu'au déverrouillage.

getc lit un caractère sur un fichier.

ioctl examine ou modifie les caractéristiques d'un fichier (particulièrement les fichiers spéciaux: terminaux, bandes,...).

```
$IOCINFO = 0xff01;        # get device info modifier
```

```
open( TAPE, "/dev/rmt0" ) || die "Could't open tape";
$ret = "a" x 1024;                  # force long buffer
ioctl( TAPE, $IOCINFO, $ret );
( $typ, $flags, $subtyp, $pad1, $ttyp, $pad2, $blk ) =
                    unpack( "C C C C S S I", $ret );
printf "type      : %c\n", $typ;
printf "flags     : %02x\n", $flags;
printf "subtype   : %02x\n", $subtyp;
printf "tape type : %04x\n", $ttyp;
printf "blocksize : %d\n", $blk;
```

affiche:

```
type      : m          # SCSI tape
flags     : 00         # no flags
subtype   : 00         # no sub-type
tape type : 0001       # streamer
blocksize : 512
```

Cet exemple illustre parfaitement quelques pièges: la zone `$ret` qui reçoit la réponse doit être initialisée pour avoir une longueur suffisante et la structure retournée doit être soigneusement définie en tenant compte des alignements de mots (ce qui rend ce genre de code non portable).

link crée un lien ("hard-link") sur un fichier.

```
link( $filename, $lfilename );
```

lstat donne les propriétés d'un lien symbolique.

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,$atime,
  $mtime,$ctime, $blksize,$blocks) = lstat( $lfilename );
```

readlink donne la valeur d'un lien symbolique.

```
readlink( $lfilename );
```

rename renomme un fichier.

```
rename( $oldfilename, $newfilename );
```

stat donne les caractéristiques d'un fichier. La syntaxe est la même que pour **lstat**.

symlink crée un lien symbolique ("soft-link").

```
symlink( $filename, $lfilename );
```

tell donne la position du pointeur dans un fichier.

```
tell( FILE );
```

unlink efface une liste de fichiers.

```
unlink( LIST );
```

utime change les dates d'accès et de modification d'une liste de fichiers.

```
utime( $accesstime, $modtime, @LIST );
```

`$accesstime` et `$modtime` doivent être en format numérique (comme ce que donne la fonction **time**).

- l'accès aux répertoires:

closedir clos un répertoire ouvert par **opendir**.

mkdir crée un répertoire.

opendir ouvre un répertoire pour le consulter par **readdir**.

readdir lit un répertoire ouvert par **opendir**.

rewinddir repositionne un répertoire à son début.

rmdir supprime un répertoire.

seekdir se positionne dans un répertoire.

telldir donne la position courante dans un répertoire.

Par exemple, le programme suivant donne la liste des fichiers du répertoire `/usr`:

```
#!/usr/local/bin/perl
opendir( DIR, "/usr" );
@file = readdir( DIR );
foreach $f (@file) { print "$f\n";};
```

- l'environnement des processus (les appels concernant les uid et gid du processus sont réalisés par la consultation ou la modification des variables spéciales `$<`, `$>`, `$(` et `$)`, voir le Chapitre 5).

chdir change de répertoire de travail.

```
chdir( $dirname );
```

chroot établit un répertoire comme la racine (`/`) pour le processus, limitant ainsi son accès à ce qui se trouve en-dessous.

```
$chroot( $dirname );
```

getlogin donne le logon du processus.

```
$login = getlogin;
```

getpgrp donne le numéro de groupe de processus d'un processus donné.

```
$pgrp = getpgrp( $pid );
$pgrp = getpgrp( 0 ); # current process
```

getppid donne le numéro de processus du père.

```
$ppid = getppid;
```

getpriority donne la priorité d'exécution d'un processus.

```
$my_priority = getpriority( 0, 0 );
         # current process priority
$proc_priority = getpriority( 0, $pid );
         # process priority
$pgrp_priority = getpriority( 1, $pgrp );
         # process group priority
$user_priority = getpriority( 2, $uid );
         # user priority
```

setpgrp fixe le numéro de groupe de processus d'un processus.

```
setpgrp( $pid, $pgrp );
         # set process group of process $pid to $pgrp
setpgrp( 0, $pgrp );
         # set process group of current process to $pgrp
```

setpriority fixe la priorité d'exécution d'un processus.

```
setpriority( 0, 0 , $priority );
         # set priority of current process
setpriority( 0, $pid , $priority );
         # set priority of process $pid
setpriority( 1, $pgrp, $priority );
         # set priority of process group $pgrp
setpriority( 2, $uid, $priority );
         # set priority of user $uid
```

umask fixe les autorisations d'accès par défaut des fichiers créés par le processus.

```
umask( $mask );
```

– le contrôle des processus. Des exemples seront donnés dans la seconde partie.

exec exécute un autre programme dans le même processus.

fork crée un processus fils.

kill envoie un signal à une liste de processus.

```
kill( $signal, @LIST );
```

wait attend la fin de l'exécution d'un processus fils. Le statut de fin du fils est dans la variable spéciale `$?`.

waitpid attend la fin de l'exécution d'un fils particulier.

```
waitpid( $pid, $flags );
```

voir `/usr/include/sys.h` pour les valeurs de `$flags`.

- la gestion du temps:

alarm programme l'envoi d'un signal `SIGALRM` après `$n` secondes.

```
alarm( $n );
```

convertit un temps du format donné par **time** en une liste pour l'heure GMT:

gmtime
```
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)
                                        = gmtime(time);
```

localtime comme **gmtime**, mais pour l'heure locale.

sleep suspend l'exécution du processus pendant `$n` secondes.

```
sleep( $n );
```

time donne le nombre de secondes depuis le premier Janvier 1970 pour être converti avec **gmtime** ou **localtime**.

times donne une liste contenant les temps (en secondes) utilisateur et système consommés par le processus et ses fils.

```
($user, $system, $cuser, $csystem) = times;
```

- l'accès aux fichiers `/etc/passwd` et `/etc/group`:

crypt encode une chaîne de caractères comme dans le fichier `/etc/passwd`.

getgrent fournit successivement les entrée du fichier `/etc/group`. Pour explorer tout le fichier:

```
while( ($name, $passwd, $gid, $members ) = getgrent ) { };
```

getgrgid fournit une entrée du fichier `/etc/group` pour un `gid` donné.

```
($name, $passwd, $gid, $members ) = getgrgid( $gid );
```

getgrnam fournit une entrée du fichier `/etc/group` pour un nom de groupe donné.

```
($name, $passwd, $gid, $members ) = getgrnam( $name );
```

getpwent fournit successivement les entrées du fichier `/etc/passwd`. Pour explorer le fichier complet:

```
while (
   ($name,$passwd,$uid,$gid,$quota,
         $comment,$gcos,$dir,$shell)
           = getpwend ) {};
```

getpwnam donne une entrée du fichier `/etc/passwd` pour un nom de logon donné.

```
($name,$passwd,$uid,$gid,$quota,
         $comment,$gcos,$dir,$shell)
           = getpwnam( $name );
```

getpwuid donne une entrée du fichier `/etc/passwd` pour un `uid` donné.

```
($name,$passwd,$uid,$gid,$quota,
         $comment,$gcos,$dir,$shell)
           = getpwuid( $uid );
```

– les autres appels système:

  pipe crée un "pipe" pour la communication entre processus (pour un exemple, voir le Chapitre 2 de la deuxième partie).

  select attend qu'une série de descripteurs de fichiers soient prêts en lecture, en écriture ou rencontrent une condition d'exception, le tout avec un mécanisme de time-out.[1]

  syscall permet d'effectuer les appels systèmes non intégrés dans **Perl** .

  system exécute une commande dans un autre processus.

– les "sockets" Berkeley BSD. En plus des fonctions de manipulation, on a toute une série de fonctions pour la résolution des noms de machines, de réseau, etc... Un exemple est donné au Chapitre 4 de la deuxième partie.

  – manipulation des "sockets":

    accept attend l'arrivée d'une connexion.

    bind enregistre un socket.

    connect ouvre une connexion avec un serveur.

    getpeername donne le nom et le port de la machine connectée.

    getsockname donne le nom et le port locaux du socket.

    getsockopt donne les options d'un socket.

1. Il y a deux **select** en **Perl** , l'un sélectionne un fichier pour **write** , l'autre est l'appel système

listen enregistre un serveur.

recv reçoit un message sur un socket.

send envoie un message sur un socket.

setsockopt positionne les options d'un socket.

shutdown ferme un socket de manière contrôlée.

socket crée un socket.

socketpair crée deux sockets (avant un **fork** par exemple pour permettre aux deux processus de communiquer).

– informations sur les noms de machines, les réseaux, les protocoles et les services.

gethostbyaddr donne le nom d'une machine à partir de son adresse.

```
($name,$aliases,$addrtype,$length,@addrs)
                 = gethostbyaddr( $addr, $addrtype );
```

`$addrtype` est toujours `$AF_INET = 2` et les adresses sont sous forme numérique (il faut donc les manipuler avec **pack** , **unpack**).

gethostbyname donne l'adresse d'une machine à partir de son nom.

```
($name,$aliases,$addrtype,$length,@addrs)
                 = gethostbyname( $name );
```

gethostent fournit successivement les entrées du fichier `/etc/hosts`. Pour explorer tout le fichier:

```
while (
   ($name,$aliases,$addrtype,$length,@addrs)
                 = gethostent ) { };
```

getnetbyaddr donne le nom d'un réseau à partir de son adresse.

```
($name, $aliases, $addrtype, $net )
                 = getnetbyaddr( $addr, $addrtype );
```

getnetbyname donne l'adresse d'un réseau à partir de son nom.

```
($name, $aliases, $addrtype, $net )
                 = getnetbyname( $name );
```

getnetent fournit successivement les entrées du fichier `/etc/networks`. Pour explorer le fichier:

```
while (
   ($name, $aliases, $addrtype, $net )
                 = getnetent ) { };
```

getprotobyname donne le numéro d'un protocole à partir de son nom.

```
($name, $aliases, $proto )
                 = getprotobyname( $name );
```

getprotobynumber donne le nom d'un protocole à partir de son numéro.

```
($name, $aliases, $proto )
                 = getprotobynumber( $proto );
```

getprotoent fournit successivement les entrées du fichier `/etc/protocols`. Pour explorer le fichier:

```
while (
   ($name, $aliases, $proto )
                 = getprotent ) { };
```

getservbyname donne le numéro d'un service à partir de son nom.

```
$proto = 'tcp';
($name, $aliases, $port, $proto )
                 = getservbyname( $name, $proto );
```

getservbyport donne le nom d'un service à partir de son numéro de port.

```
$proto = 'tcp';
($name, $aliases, $port, $proto )
                 = getservbyport( $name, $proto );
```

getservent fournit successivement les entrées du fichier `/etc/services`. Pour explorer tout le fichier:

```
while (
   ($name, $aliases, $port, $proto )
                 = getservent ) { };
```

Ces fonctions peuvent obtenir leurs informations ailleurs que dans les fichiers `/etc/...`. Par exemple, les résolutions d'adresses ip peuvent être faites en consultant un serveur de nom. L'exemple suivant obtient l'adresse ip d'une machine:

```
#!/usr/local/bin/perl
( $name, $aliases, $addrtype, $length, @addrs )
                          = gethostbyname( $ARGV[0]);
($a, $b, $c, $d) = unpack( 'C4', $addrs[0] );
print "$a.$b.$c.$d\n";
```

- la communication entre processus (IPC System V). Elle comprend trois services:

  - la mémoire partagée:

    shmctl contrôle la mémoire partagée.

    shmget accède à la mémoire partagée.

    shmread lit une zone de mémoire partagée.

shmwrite écrit une zone de mémoire partagée.

- les queues de messages:

  msgctl contrôle la queue de messages.

  msgget accède à la queue de messages.

  msgrcv lit un message en queue.

  msgsnd met un message en queue.

- les sémaphores:

  semctl contrôle des sémaphores.

  senget accès à un sémaphore.

  semop manipulation d'un sémaphore.

- l'accès aux bases de données dbm. **Perl** permet d'accéder à une base de données dbm en lui associant un vecteur associatif. Les modifications du vecteur seront répercutées automatiquement sur la base. De même, les recherches sur des clés du vecteur associatif seront traduites en recherches indexées dans la base. Notez que la base n'est PAS chargée entièrement dans le vecteur. **Perl** maintient simplement un cache. On a ainsi un procédé très simple à mettre en œuvre qui permet de manipuler des bases de taille importantes.

  dbmclose met fin à l'accès d'une base de données dbm.

  ```
  dbmclose( %array );
  ```

  dbmopen établit une relation entre une base de données dbm et un vecteur associatif .

  ```
  dbmopen( %array, file, mode );
  ```

  `%array` est le vecteur associatif, `file` est le nom de la base de données (sans les extensions `.pag` ou `.dir`) et `mode` est le mode d'accès, par exemple 0600 pour un accès en lecture/écriture par le propriétaire. La base peut alors être manipulée par l'intermédiaire de ce vecteur.

8.2 L'interprétation de commandes Perl

Il est possible de faire interpréter à **Perl** une expression comme un programme par la fonction **eval** . A la sortie, la variable spéciale `$@` contient le message d'erreur s'il y en a un (**eval** donne alors la valeur **undef**). S'il n'y a pas de message d'erreur, **eval** donne la dernière expression évaluée. Cette fonction a de nombreuses applications:

- générer du code dynamiquement et l'interpréter;

– écrire un petit shell interactif (voir un exemple au Chapitre 1 de la deuxième partie);

– protéger une section de code qui peut générer des erreurs fatales (par exemple tester une fonctionnalité qui n'existe pas sur toutes les implémentations).

Pendant l'exécution, on a accès à toutes les variables et sous-programmes du script principal. L'expression évaluée peut être sur plusieurs lignes en utilisant l'opérateur "here document" >> . Par exemple:

```
#!/usr/local/bin/perl
for($i=0;$i<=10;$i++){$list[$i] = $i;};
$sub =  <<'EndOfEval';
   for( local($i)=0; $i<=$#list; $i++){
      $list[$i] *= 2;
   }
EndOfEval
eval $sub;
for($i=0;$i<=$#list;$i++){print $i, ": ", $list[$i], "\n";};
```

Ce programme construit une variable `$sub` qui contient un programme qui multiplie les éléments d'une liste par 2. `eval $sub` exécute alors ce programme.

8.3 La manipulation des structures binaires

Pour constituer les structures qui servent dans certains appels, il est nécessaire de pouvoir manipuler des données binaires. Cela se fait à l'aide des fonctions **pack** et **unpack**:

```
$struct = pack( $format, @liste );
@list = unpack( $format, $struct );
```

pack utilise la description `$format` pour constituer une structure binaire `$struct` à l'aide des éléments de la liste `@liste`. **unpack** réalise l'opération inverse, extrayant les données binaires de la variable `$struct` suivant le format spécifié et les place dans la liste.

Les différents descripteurs pour le format sont indiqués dans le Tableau 8.1.

La génération des formats pour manipuler les structures des fichiers d'entête peut être facilitée par l'utilitaire **h2ph** qui convertit un fichier `.h` en un fichier d'extension `.ph` contenant des variables **Perl** correspondant aux diverses constantes et structures. Ce fichier peut alors être inclus par un **require** pour disposer de ces variables. On verra quelques exemples d'utilisation de **pack** et **unpack** dans le Chapitre 4 de la deuxième partie.

a	une chaîne de caractères mis à 'NULL'
A	une chaîne de caractères mis à 'SPACE'
b	une suite de bits avec les bits de poids faible d'abord
B	une suite de bits avec les bits de poids fort d'abord
h	une chaîne héxadécimale avec les 4 bits de poids faible d'abord
H	une chaîne héxadécimale avec les 4 bits de poids fort d'abord
c	un octet signé
C	un octet non signé
s	un entier court signé
S	un entier court non signé
i	un entier signé
I	un entier non signé
l	un entier long signé
L	un entier long non signé
n	un entier court dans l'ordre "réseau"
N	un entier long dans l'ordre "réseau"
f	un flottant simple précision
d	un flottant double précision
p	un pointeur sur une chaîne de caractères
x	un octet nul
X	recule d'un octet
@	remplit avec des 'NULLs' n octets
u	"uuencode" une chaîne de caracteres

TAB. 8.1 - *Les formats de* **pack**

Une autre fonction utile pour la manipulation de données binaires est **vec**:

```
$bits = vec( $var, $delta, $width );
```

prend la variable `$var`, la considère comme un vecteur de groupes de `$width` bits et en extrait le groupe `$delta`. Par exemple:

```
$a = pack( "C", 11);
print vec( $a, 0, 1 ), "\n";
print vec( $a, 1, 1 ), "\n";
print vec( $a, 2, 1 ), "\n";
print vec( $a, 3, 1 ), "\n";
print vec( $a, 0, 2 ), "\n";
print vec( $a, 1, 2 ), "\n";
print vec( $a, 0, 4 ), "\n";
```

affiche:

```
1
1
0
1
3
2
11
```

Quelques explications: tout d'abord, pour constituer une donnée binaire, il est nécessaire de passer par **pack** , `$a = 11` ne donnerait pas du tout le résultat escompté! Les quatre premières impressions affichent le nombre bit à bit, les deux suivantes par groupes de deux bits et la dernière imprime quatre bits. Cette fonction est souvent utilisée avec l'appel système **select** qui utilise des masques de bits pour décrire les descripteurs de fichier à surveiller.

8.4 La gestion des variables

Trois fonctions servent à la gestion des variables:

reset réinitialise toutes les variables dont le nom commence par un des caractères de la chaîne passée en argument. Par exemple:

```
reset 'a-z';
```

réinitialise toutes les variables qui commencent par une minuscule.

undef supprime une variable. Sans argument, il donne la valeur **undef** qui peut par exemple être retournée par un sous-programme.

defined permet de tester si une variable existe.

Chapitre 9

Les expressions régulières

Il s'agit d'un des mécanismes les plus puissants de **Perl** et comme tel, un de ceux qui est le plus difficile à maitriser...

9.1 Recherche et substitution de chaînes

Perl propose trois mécanismes pour manipuler les chaînes de caractères:

- `/PATTERN/` recherche les chaînes correspondant à `PATTERN`. L'option `i` permet de ne pas distinguer majuscules et minuscules et l'option `g` permet de faire une recherche globale. Une variante: `m/PATTERN/` permet d'utiliser n'importe quel caractère à la place de `/` pour délimiter le motif de recherche. Le résultat de la recherche peut être testé pour savoir si le motif a pu être trouvé.

- `s/PATTERN1/PATTERN2/` substitue les expressions `PATTERN2` dans les chaînes qui correspondent à `PATTERN1`. L'option `i` permet de ne pas distinguer majuscules et minuscules, l'option `g` permet de faire une recherche globale et l'option `e` permet de considérer `PATTERN2` comme une commande à exécuter.

- `tr/LIST1/LIST2/` translate de `LIST1` vers `LIST2`. S'il n'y a plus de caractères dans `LIST1` que dans `LIST2`, le dernier caractère de `LIST2` est utilisé pour la compléter (sauf avec l'option `d`). Si `LIST2` est vide, une copie de `LIST1` est utilisée (sauf avec l'option `d`). Avec l'option `d`, les caractères de `LIST1` qui ne sont pas dans `LIST2` sont supprimés. Avec l'option `c`, on utilise le complément de `LIST1` à la place de `LIST1`. Enfin avec l'option `s`, si des substitutions consécutives génèrent le même caractère, un seul est utilisé. Le résultat de `tr` est le nombre de caractères transformés (donc `tr/A/A/` compte les `A`).

Toutes ces opérations s'appliquent par défaut à la variable `$_`. Pour les appliquer à une autre variable, on utilise l'opérateur `=~`.

9.2 Les expressions régulières

Perl utilise une variété d'expressions régulières qui est un sur-ensemble de celles des autres utilitaires (**sed** , **awk** , shells, ...). Les règles (un peu simplifiées!) sont les suivantes:

- une chaîne de caractères doit correspondre à ces caractères exactement;
- parmi ces caractères, on peut avoir:
  - des caractères "spéciaux" comme `\t` pour indiquer une tabulation;
  - des listes de caractères comme `[a-z]`;
  - des caractères génériques comme `\d` pour un chiffre;
  - des caractères de positionnement comme `^` pour indiquer le début;
  - des caractères de répétition comme `+` pour indiquer une fois au moins;
- plusieurs chaînes peuvent être proposées en alternative en les séparant par des `|`;
- des motifs peuvent être isolés par des parenthèses. Les chaînes de caractères qui leur correspondent sont alors mémorisées dans `\1`, `\2`, ... et sont affectées aux variables `$1`, `$2`,... pour être utilisées par la suite.

Le tableau 9.1 résume ces différentes possibilités.

9.3 Quelques exemples

Nous avons déjà rencontré quelques expressions régulières, en voici quelques autres usages:

```
$_ = "Date: 24 Jan 96 22:32:15 GMT";
/^Date: (\d+) (\w+) (\d+) (\d+):(\d+):(\d+) (.*)$/;
print "$1,$2,$3,$4,$5,$6,$7\n";
```

affiche:

```
24,Jan,96,22,32,15,GMT
```

En effet, on cherche la chaîne `DATE:` (avec un espace!) au début (à cause du `^`), suivi d'un ou plusieurs chiffres (que l'on retrouvera dans `$1`), puis un espace, plusieurs lettres (que l'on retrouvera dans `$2`), ... jusqu'à un espace, puis 0 ou plusieurs caractéres jusqu'à la fin (`$`) que l'on mettra dans `$7`.

On peut faire des regroupements dans les parenthèses:

```
$_ = "Date: 24 Jan 96 22:32:15 GMT";
/^Date: ((\d+) (\w+) (\d+)) ((\d+):(\d+):(\d+)) (.*)$/;
print "$1,$2,$3,$4,$5,$6,$7,$8,$9\n";
```

<table>
<tr><td>.</td><td>N'importe quel caractère sauf un retour chariot</td></tr>
<tr><td>[a-z0-9]</td><td>N'importe quel caractère d'un ensemble</td></tr>
<tr><td>[^a-z0-9]</td><td>N'importe quel caractère qui n'est pas dans l'ensemble</td></tr>
<tr><td>\d</td><td>Un chiffre</td></tr>
<tr><td>\D</td><td>Un caractère qui n'est pas un chiffre</td></tr>
<tr><td>\w</td><td>Un caractère alphanumérique</td></tr>
<tr><td>\W</td><td>Un caractère non alphanumérique</td></tr>
<tr><td>\s</td><td>Un caractère "blanc"</td></tr>
<tr><td>\S</td><td>Un caractère qui n'est pas un blanc</td></tr>
<tr><td>\n</td><td>Un saut de ligne</td></tr>
<tr><td>\r</td><td>Un retour chariot</td></tr>
<tr><td>\t</td><td>Une tabulation</td></tr>
<tr><td>\f</td><td>Un saut de page</td></tr>
<tr><td>\b</td><td>Une "correction" (^H) (entre [])</td></tr>
<tr><td>\0</td><td>Un "NULL"</td></tr>
<tr><td>\nnn</td><td>Le caractère de code ASCII nnn en octal</td></tr>
<tr><td>\xnn</td><td>Le caractère de code ASCII nn en héxadécimal</td></tr>
<tr><td>\cX</td><td>Le caractère de contrôle ^X</td></tr>
<tr><td>\meta</td><td>Le méta caractère (\|, \.,...)</td></tr>
<tr><td>(abc)</td><td>Mémorise la chaîne trouvée</td></tr>
<tr><td>\1</td><td>Ce qui a été mémorisé par la première parenthèse</td></tr>
<tr><td>\2</td><td>Ce qui a été mémorisé par la deuxième parenthèse</td></tr>
<tr><td>\3</td><td>Ainsi de suite...</td></tr>
<tr><td>x?</td><td>Zéro ou une fois le motif x</td></tr>
<tr><td>x*</td><td>Zéro ou plusieurs fois motifs x</td></tr>
<tr><td>x+</td><td>Une ou plusieurs fois motifs x</td></tr>
<tr><td>x{m,n}</td><td>Au moins m fois mais moins que n fois le motif x.</td></tr>
<tr><td>abc</td><td>La chaîne abc</td></tr>
<tr><td>abc|def</td><td>Une des chaînes abc ou def</td></tr>
<tr><td>\b</td><td>Une limite de mot alphanumérique (en dehors de [])</td></tr>
<tr><td>\B</td><td>Une limite de mot non alphanumérique</td></tr>
<tr><td>^</td><td>En début de chaîne</td></tr>
<tr><td>$</td><td>En fin de chaîne</td></tr>
</table>

TAB. 9.1 - *Les motifs des expressions régulières*

donne:

```
24 Jan 96,24,Jan,96,22:32:15,22,32,15,GMT
```

Notez l'ordre dans lequel sont affectées les variables.

Le résultat de la recherche est vrai ou faux dans un contexte scalaire (si on le teste par exemple), mais la liste des variables `$i` dans un contexte vectoriel (si on l'affecte à une liste par exemple):

```
$_ = "Date: 24 Jan 96 22:32:15 GMT";
@d = /^Date: (\d+) (\w+) (\d+) (\d+):(\d+):(\d+) (.*)$/;
foreach $i (@d) { print "$i,";}; print "\n";
```

donne:

```
24,Jan,96,22,32,15,GMT,
```

La recherche peut être rendue globale avec l'option `g`:

```
$_ = 'un deux trois quatre';
@w = /\s*(\w+)/;
print @w, "\n";
@w = /\s*(\w+)/g;
foreach $i (@w) { print "$i ";};print "\n";
```

donne:

```
un
un deux trois quatre
```

Ici, on cherche un nombre arbitraire de caractères blancs suivi d'une ou plusieurs lettres que l'on mémorise. La recherche simple ne donne que la première occurence et la recherche globale fournit la liste de toutes les occurences. Si on utilise la recherche globale dans un contexte scalaire, elle devient itérative: la dernière occurence est mémorisée et la recherche suivante part de là:

```
while( /\s*(\w+)/g ) { print "$1 " ;};print "\n";
```

donne:

```
un deux trois quatre
```

En effet, la recherche est faite dans un test, donc un contexte scalaire, elle est effectuée, donne vrai ou faux suivant le résultat, la position est mémorisée et l'on recommence jusqu'à la fin où la recherche donne le résultat faux. On peut faire quelque chose de similaire en utilisant l'option `e` de la substitution:

```
s/\s*(\w+)/print "$1 "/eg;print "\n";
```

donne le même résultat: on fait une substitution globale (option **g**) et à chaque occurence, on exécute (option **e**) l'impression de la chaîne trouvée.

Quelques exemples avec **tr** maintenant:

```
$a = "Noel Giraud";
$a =~ tr/A-Z/a-z/;
print "$a\n";
```

transpose en minuscules:

```
noel giraud
```

On peut utiliser l'option c pour traduire tout ce qui n'est pas dans la première liste:

```
$a = 'un %&(*(deux tr%%^%ois 12#$3576';
$a =~ tr/A-Za-z0-9/ /c;
print "$a\n";
```

donne:

```
un      deux tr    ois 12  3576
```

Tous les caractères qui ne sont ni des lettres, ni des chiffres sont transformés en blanc (la seconde liste est complétée en répétant son dernier élément, soit un blanc). Avec l'option s, on peut en plus compacter les blancs:

```
$a = 'un %&(*(deux tr%%^%ois 12#$3576';
$a =~ tr/A-Za-z0-9/ /cs;
```

donne:

```
un deux tr ois 12 3576
```

Avec l'option d, on peut supprimer les caractères non alphanumériques:

```
$a = 'un %&(*(deux tr%%^%ois 12#$3576';
$a =~ tr/A-Za-z0-9 //cd;
print "$a\n";
```

donne:

```
un deux trois 123576
```

La deuxième liste étant vide, tout ce qui n'est ni une lettre, ni un chiffre, ni un blanc (on l'a rajouté à la première liste!) est supprimé.

Chapitre 10

Mise au point des programmes

Perl offre deux aides pour la mise au point des programmes:

- une option permettant de tracer l'exécution;
- une option permettant d'exécuter le programme dans le debugger intégré.

10.1 Tracer l'exécution d'un programme

Cela se fait à l'aide de l'option `-D` lors de l'appel de **Perl** ou sur la première ligne du programme[1]. On peut aussi modifier le niveau de trace au cours de l'exécution en changeant la valeur de la variable `$^D` . Le niveau de debug est indiqué par les bits de cette variable. Leur signification est donnée dans le tableau 10.1 (en Anglais!).

Une valeur raisonnable pour `-D` est 8 qui permet de suivre l'exécution sans une avalanche d'informations incompréhensibles (mais je vous concède volontiers que celà pourrait être plus clair[2]...).

10.2 Le debugger intégré

Il est invoqué par l'option `-d` sur la ligne de commande ou en option dans la première ligne du programme. Cette option entraîne l'exécution du programme sous le contrôle d'un debugger interactif. L'exécution du programme est suspendue à la première ligne et on obtient une invite. Le programme est alors exécuté sous votre contrôle. Les commandes de ce debugger sont résumées dans le Tableau 10.2. Cette possibilité est de loin plus efficace pour la mise au point de programmes.

1. Cette option n'est disponible que si **Perl** a été compilé avec l'option `-DDEBUGGING`
2. N'ayez pas trop de regrets si votre version de **Perl** n'est pas compilée avec cette option, elle est vraiment réservée au debug de **Perl** plus que des scripts!

1	Tokenizing and Parsing
2	Command Linkage
4	Label Stack Processing
8	Trace Execution
16	Operator Node Construction
32	String/Numeric Conversions
64	Print Preprocessor Command for -P
128	Memory Allocation
256	Format Processing
512	Regular Expression Parsing
1024	Syntax Tree dump
2048	Tainting Check
4096	Memory Leaks
8192	Hash Dump–Usurps Values()

TAB. 10.1 - *Valeurs du niveau de trace*

h	Affiche l'aide
T	Trace la pile
s	Exécution pas-à-pas
n	Exécution jusqu'au dernier retour de sous-programme
r	Termine l'exécution du sous-programme en cours
c *line*	Exécute jusqu'à *line*
`<CR>`	Recommence s ou n
l *min+inc*	Affiche *inc+1* lignes à partir de *min*
l *line*	Affiche la ligne *line*
l	Affiche une "page"
f *file*	Change de fichier
-	Liste la "page" précédente
w *line*	Affiche la "page" autour de la ligne *line*
l *name*	Affiche le sous-programme *name*
/*pattern*/	Recherche *pattern*
?*pattern*?	Recherche *pattern* en arrière
L	Affiche les point d'arrêt
S	Affiche le nom des sous-programmes
t	Bascule du mode trace
b *line condition*	Établit un point d'arrêt
d *line*	Supprime un point d'arrêt
D	Supprime tous les points d'arrêt
a *line command*	Établit un point d'action
A	Supprime tous les points d'action
< *command*	Action avant l'invite
> *command*	Action après l'invite
V *package vars*	Affiches les variables d'un **package**
X *vars*	Affiche les variables du **package** courant
! *num*	Recommence une commande
! -*num*	Recommence une commande
H -*num*	Ré-affiche les dernières commandes
q	Sortie
command	Commande **Perl**

TAB. 10.2 - *Commandes du debugger integré*

Deuxième partie

Quelques exemples de programmes

Avertissement

Dans cette partie, nous allons étudier quelques petits programmes **Perl** . Ils ne sont pas du tout écrits dans le but d'obtenir les meilleurs programmes possibles, mais pour montrer les diverses techniques utilisables en programmation **Perl** . D'autre part, tous les vocables de **Perl** étant des mots Anglais, les commentaires dans le code des programmes le sont aussi! Notez aussi un changement de typographie: dans la première partie, les mots clé du langage **Perl** étaient notés **comme cela**, ici, ils le seront `comme ceci` pour les harmoniser avec la typographie des énoncés des programmes.

Chapitre 1

Un shell Perl

Bien que présentant de nombreuses caractéristiques des shells, **Perl** ne peut cependant pas être utilisé tel quel comme un shell interactif. La fonction `eval` fournit une manière simple de pallier ce manque. Le programme suivant:

```
#!/usr/local/bin/perl
while (<>) { eval; print $@; }
```

constitue un petit interpréteur **Perl** . Comme beaucoup de scripts **Perl** , il utilise abondamment les valeurs par défaut.

L'instruction `while` exécute le bloc d'instructions suivant tant que l'ordre de lecture `<>` fournit un résultat. La source de cette lecture est par défaut l'entrée standard. Le résultat de cette lecture est placé dans la variable spéciale `$_` . L'instruction `eval` utilise par défaut le contenu de cette variable spéciale, l'évalue, affiche un éventuel résultat. En cas d'erreur, le message se trouve dans la variable spéciale `$@` qui est affichée en fin de bloc.

Quelques remarques sur ce programme: tout d'abord, l'expression entrée doit être complète, c'est à dire que si elle comporte des blocs, par exemple à la suite de tests, tout doit figurer dans la même ligne. Par exemple, les lignes suivantes sont valides:

```
@num = 0..6;
for (@num) { print; }
```

Ensuite, ce simple programme est particulièrement utile pour effectuer des petits test, spécialement pour apprendre l'usage des expressions régulières.

Ce programme vous semble encore un peu trop long? Essayez:

```
perl -ne 'eval;print $@'
```

Chapitre 2

Exécution d'une commande

Contrairement aux shells, **Perl** ne permet pas de placer directement des commandes **Unix** au milieu du script (elles déclenchent des erreurs de syntaxes). **Perl** fournit cependant quatre procédés pour exécuter des commandes et en traiter éventuellement le résultat.

2.1 La fonction system

C'est la manière la plus simple d'exécuter une commande: la commande est exécutée dans un autre processus et l'interpréteur **Perl** attend la fin de ce processus pour continuer. La valeur retournée par `system` est celle donnée par la fonction wait. Le code retour de la commande exécutée se trouve donc dans le deuxième octet. Par contre, le résultat de la commande n'est pas accessible au programme **Perl** , il est éventuellement affiché sur le terminal. Ainsi:

```
$a = system( "ls" );
$a /= 256;
```

affiche la liste des fichiers au terminal et affecte à `$a` la valeur `0` (en divisant par 256, on récupère le code de retour qui est dans le deuxième octet). Et:

```
$a = system( "ls -w" );
$a /= 256;
```

affiche:

```
Usage: ls [-1ARadCxmnleogrtucpFbqisfL] [File...] [Directory...]
```

et affecte à `$a` la valeur 2. Une autre fonction similaire est `exec` qui exécute la commande dans le même processus qui est ainsi terminé (son code est remplacé par celui de la commande). Ainsi:

```
system("ps");
exec("ps");
print "Jamais\n";
```

affiche quelque chose comme:

```
  PID    TTY  TIME CMD
 10640  pts/2  0:00 perl t1
 12326  pts/2  0:00 -usr/bin/tcsh
 17297  pts/2  0:00 ps
  PID    TTY  TIME CMD
 10640  pts/2  0:00 ps
 12326  pts/2  0:00 -usr/bin/tcsh
```

Quand on exécute `ps` avec `system`, on voit le processus du shell initial, le processus qui interprète le programme **Perl** et le processus qui exécute `ps`, mais quand on utilise `exec`, le processus **Perl** exécute maintenant ps. Quand à la ligne `print`, elle n'est bien sûr jamais atteinte. Les arguments de ces fonctions présentent quelques subtilités: si le premier argument est une variable et que l'on donne un deuxième argument sans virgule entre les deux, le deuxième argument est utilisé comme nom de processus.

```
$a = "ps";
system $a "tralala";
```

donne quelque chose comme:

```
  PID    TTY  TIME CMD
 10683  pts/2  0:00 perl t1
 12326  pts/2  0:00 -usr/bin/tcsh
 17340  pts/2  0:00 tralala
```

C'est bien la commande `ps` qui est exécutée, mais sous le nom `tralala`.

2.2 Les accents graves ‘‘

Une manière plus utile est, comme dans les shells, de placer la commande entre des accents graves et de l'affecter à une variable. La commande est exécutée comme précédemment dans un autre processus et le résultat est affecté à la variable. Si l'on veut examiner le code de retour, il se trouve dans le deuxième octet de la variable spéciale `$?`. Ce qui est affecté à la variable est ce que la commande écrit sur sa sortie standard. Si elle écrit sur `stderr`, cela s'affiche sur le terminal. Le résultat de la commande est traité comme une chaîne de caractères contenant toutes les lignes de la réponse avec les sauts de ligne. Par exemple:

```
$a = `ls`
@a = split(/\n/, $a)
foreach $a (@a) {print "Fichier: $a\n"}
```

La première ligne affecte à `$a` une chaîne constituée des noms de fichiers séparés par des sauts de ligne. La seconde crée une liste `@a` constituée des noms de fichiers et la dernière affiche chaque nom de fichier précédé de `Fichier:` . Remarquez qu'il n'y a pas d'inconvénient à utiliser le même nom pour la variable simple et la liste (si ce n'est que cela n'améliore pas la lisibilité du programme...).

2.3 La fonction open

La fonction `open` permet, comme il a été signalé au Chapitre 7, d'exécuter une commande avec un "pipe" qui pointe soit sur l'entrée standard, soit sur la sortie standard du processus qui exécute la commande. Dans le premier cas, il faut précéder le nom de la commande par `|`, dans le deuxième cas, le faire suivre de `|`. Quelques exemples:

```
open( LS, "ls -l |");
while(<LS>) {
   split;
   print "$_[0] $_[8]\n";
}
```

On ouvre un decripteur de fichier qui pointe sur un pipe relié à la sortie standard de la commande `ls -l`. Quand on lit sur ce descripteur, on récupère le résultat de la commande ligne par ligne dans la variable `$_`. Ici, on affiche donc les protections et le nom de chaque fichier.

```
$a = "giraud@frcpn11";
open( MAIL, "|mail $a" );
print MAIL "Salut l'ami...";
close MAIL
```

Cette fois, on ouvre un descripteur de fichier qui pointe sur l'entrée standard de la commande `mail`. Notez au passage que le nom de la commande peut être construit dynamiquement à partir de variables. Quand on écrit sur ce descripteur, on fabrique le corps du courrier. La fermeture du descripteur déclenche alors l'envoi de ce courrier.

2.4 Les fonctions pipe , fork et exec

Comme en **C** , on peut utiliser les appels système `pipe`, `fork` et `exec` pour créer un nouveau processus et lui faire exécuter une commande en lui passant des données par un premier `pipe` et en lisant le résultat sur un second `pipe`. Cela ne présente d'intérêt que si l'on doit simultanément écrire et lire sur la commande, sinon, `open` fait tout aussi bien. La mise en œuvre n'est pas des plus simple...

Pour l'illustrer, voici un programme qui utilise la commande `sort` de **Unix** pour trier une liste d'adresse ip (aussi bizarre que cela puisse paraître, cet exemple est inspiré d'un programme qui sert vraiment!).

```
#!/usr/local/bin/perl
@a = ('1.2.3.4',
      '20.2.3.4',
      '11.2.3.4',
      '10.2.3.4',
      '100.2.3.4');          # the list to sort
pipe( RE1, WR1);             # pipe to sort
pipe( RE2, WR2);             # pipe from sort
if ( fork() == 0 ){          # in the child
   close(STDIN);             # close STDIN
   open(STDIN, "<&RE1");     # duplicate RE1 on STDIN
   close(RE1);               # close RE1
   close(WR1);               # and WR1
   close(STDOUT);            # close STDOUT
   open(STDOUT, ">&WR2");    # duplicate WR2 on STDOUT
   close(WR2);               # close WR2
   close(RE2);               # and RE2
                             # now STDIN is RE1 and STDOUT is WR2
                             # sort will read RE1, write WR2
   exec("sort -t. +0 -1n +1 -2n +2 -3n +3 -4");    # do it!
} else {                     # in the parent
   close(WR2);               # close WR2
   close(RE1);               # close RE1
   foreach $a (@a) {         # write the list on WR1
      print WR1 "$a\n";
   }
   close(WR1);               # close WR1, this will start sort!
   while(<RE2>) {            # read result on RE2
      print "$_";            # and print it
   }
}
```

Quelques mots d'explication: la fonction `pipe` crée deux descripteurs de fichiers mis "bout-à-bout", l'un en écriture, l'autre en lecture. On lit sur le deuxième ce que l'on a écrit sur le premier. On commence donc par créer deux `pipes`. On fait ensuite un `fork`, ce qui duplique le processus, le code continuant à s'exécuter dans les deux. Dans le processus fils, `fork` donne `0`, dans le père, il donne le numéro du processus du fils. Dans le fils, on commence par clore `STDIN`, puis on duplique `RE1` sur `STDIN` (par l'option `&` de `open`). On peut alors fermer les descripteurs `RE1` et `WR1` devenus inutiles. On procède ensuite de la même manière pour `STDOUT`: on le

clos, on duplique `WR2` dessus et on ferme les descripteurs inutiles. Cela a l'air un peu compliqué, mais le but est simplement d'avoir dans le fils `STDIN` et `STDOUT` qui pointent sur `RE1` et `WR2`, de sorte que `sort` utilise ces `pipes`. Il ne reste plus qu'à appeler `sort`[1] par un `exec`. Dans le père, les choses sont plus simples: on commence par fermer les descripteurs inutiles, puis on écrit le contenu de la liste sur `WR1`. Ce descripteur étant un `pipe` sur `RE1`, sort lit sur son entrée standard ce qu'on y écrit. Quand on clos `WR1`, `sort` commence son tri et écrit le résultat sur sa sortie standard qui est le `pipe` `WR2`. Il suffit donc de lire `RE2` pour trouver la liste triée... La figure 2.1 éclaircira peut-être un peu tout ça:

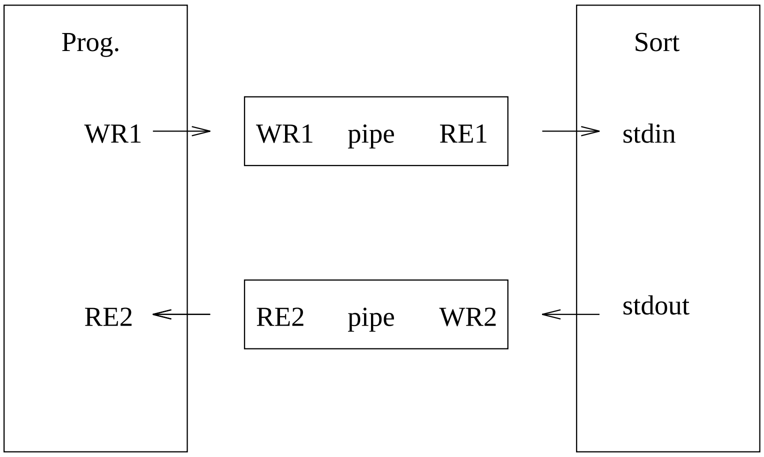

FIG. 2.1 - *Exécution de la commande sort*

L'exécution de ce programme donne :

```
1.2.3.4
10.2.3.4
11.2.3.4
20.2.3.4
100.2.3.4
```

1. Amusant les arguments de `sort` pour trier des adresses ip, non?

Chapitre 3

Compter les mots

3.1 Compter les mots d'un fichier

Compter les mots d'un fichier nécessite trois précautions:

- le même mot pouvant être écrit avec des majuscules, il convient de tout transformer dans une représentation unique, par exemple en minuscules.
- les mots en fin de ligne ont pu être coupés par un trait d'union. Il convient donc de traiter le fichier non pas ligne à ligne, mais paragraphe par paragraphe.
- la manière de découper les paragraphes en mots doit être soigneusement choisie.

3.2 Le programme

Cet exemple est une version légèrement modifiée de celui donné dans [2] (pour compter correctement les mots précédés d'un article élidé).

```
#!/usr/local/bin/perl
$/ = "";
$* = 1;
while(<>){
   s/-\n//g;
   tr/A-Z/a-z/;
   @words = split(/\s*\W+\s*/, $_);
   foreach $word (@words){
      $wordcount{@word}++;
   }
}
```

```
foreach $word (sort keys(%wordcount)){
   printf "%20s %d\n", $word, $wordcount{$word};
}
```

La première chose à faire est de lire paragraphe par paragraphe: on positionne pour cela le séparateur d'enregistrements `$/` à la chaîne vide (c'est équivalent à `\n\n`). De même, on positionne la variable `$*` à 1 pour indiquer que les chaînes de caractères peuvent être réparties sur plusieurs lignes. On peut alors commencer la boucle de lecture qui affecte chaque paragraphe à la variable `$_`. Les mots coupés en fin de ligne sont "recollés" en supprimant les traits d'union suivis d'un saut de ligne par la fonction **s** . Les paragraphes sont alors convertis en minuscules par la fonction **tr** . Il ne reste plus qu'à découper les paragraphes en mots: la commande **split** constitue une liste `@words` de ces mots. Le motif utilisé pour **split** décrit le séparateur comme étant constitué d'au moins un caractère non alpha-numérique (`\W+`) éventuellement entouré par des caractères "blancs" (`\s*`), c'est-à-dire des espaces, des tabulations, des sauts de ligne ou des retours chariot. La liste constituée est alors explorée par une boucle **foreach** et les compteurs associés à chaque mot sont incrémentés dans le vecteur associatif `%wordcount`. Une fois le ou les fichiers explorés, on affiche le résultat en explorant toutes les clés préalablement triées du vecteur associatif.

Chapitre 4

Un serveur de base de données

Dans ce chapitre, nous allons construire progressivement une véritable application en **Perl** . Il s'agit d'un petit serveur de base de données. La première version se contentera de la constituer. Nous la complèterons ensuite pour que l'on puisse l'interroger depuis le terminal. La version suivante sera elle un véritable serveur Internet multitâche. Nous terminerons par quelques "embellissements".

4.1 Description du problème

Le programme que nous allons écrire est un petit annuaire. En partant d'un fichier de texte contenant les noms et numéros de poste, on voudrait réaliser un petit serveur donnant le numéro de poste des personnes demandées. Le format du fichier sera le suivant:

- le nom sera donné sur une ligne commençant par **`*name:`** suivi du nom;
- le numéro de poste correspondant sera sur la prochaine ligne commençant par **`*phone:`**;
- le nom sera écrit en minuscules;
- le fichier peut contenir d'autres informations sur des lignes commençant par **`*anything:`**;
- toutes les autres lignes seront considérées comme des commentaires et ignorées.

Notre fichier aura donc une forme du genre:

```
sample data
Noel Giraud entry
*name: giraud
*firstname: noel
```

```
*group: system
*phone: 308
Pierre Larrieu entry
*name: larrieu
*firstname: pierre
*group: user_support
*phone: 204
```

4.2 La lecture de la base

Dans un premier temps, on se contente de lire le fichier. Les lignes commençant par `*names:` et `*phone:` sont utilisées pour remplir un vecteur associatif `%phone`. Une fois le fichier lu, on affiche ce qu'on a trouvé: le nombre d'entrées lues et la liste triée des entrées.

```
#!/usr/local/bin/perl
# read database
open(DATA, "data") || die "Could't open data file";
while(<DATA>){
   /^\*name:/ && do { split ; $name = $_[1] ; next };
   /^\*phone:/ && do { split ; $phone{$name} = $_[1] ; next };
};
close DATA;
# report the result
@name = sort keys(%phone);   # build sorted keys list
print $#name+1, " entries read:\n";
foreach $name (@name) {
   printf "%-18s, phone number: %4s\n" , $name , $phone{$name};
};
```

Pour l'ouverture du fichier, on utilise la construction traditionelle *action* `||` *die* en cas d'erreur. Dans la boucle **while** de lecture, on utilise une construction du type **case** par des lignes: *cas* `&& do {` *action*`}`. Les lignes contenant les mots clés `*name:` et `*phone:` sont recherchées par `//` en précisant qu'ils doivent se trouver en début de ligne (par `^`). Les lignes ainsi sélectionnées sont découpées en mots par **split** de manière à ce que la valeur du champ se trouve dans `$_[1]`. Ces valeurs sont entrées dans un vecteur associatif `%phone`, le nom servant de clé. Une fois le fichier exploré, on constitue une liste triée des noms `@name` en triant les clés du vecteur associatif `%phone`. Pour terminer, on affiche le nombre d'entrées trouvées (`$#name+1`), puis toutes les entrées par une boucle `foreach` sur les éléments de la liste `@name`.

4.3 Le serveur pour le terminal

Une fois la base de données installée dans le vecteur associatif, on peut l'utiliser pour un annuaire. La seule difficulté consiste dans la saisie des questions:

```
# all is ready, loop on queries
$query = 0;   # queries number
print "Welcome to this small perl program. Enter q to quit\n> ";
while (<>) {   # try to answer queries
                                          # cleanup and lowercase
   $_ =~ tr/[ \t\r\n]//d ;$_ =~ tr/A-Z/a-z/;
   last if ( $_ eq "q" );                          # quit ?
   $query++;
   print  $_ ,
       " : " , $phone{$_} ? $phone{$_} : "unknown" , "\n>";
   }
print  $query , " queries, Goodbye...\n";   # game over...
exit;
```

Avant de commencer, on prépare un compteur pour le nombre de questions posées et on affiche une petite invite, ne serait-ce que pour indiquer comment sortir du programme! Le terminal est lu par une boucle sans fin `while(<>)`. Les lignes lues doivent d'abord subir un petit traitement: il convient de supprimer les espaces, les tabulations, les sauts de ligne et retours chariot. Une possibilité est d'utiliser l'instruction `tr` avec l'option `d` (ce qui est fait ici), mais un `s/[ \t\r\n]//g` ferait tout aussi bien l'affaire. Il faut ensuite tout traduire en minuscules, à nouveau avec `tr` . La variable `$_` contient maintenant une clé utilisable. On vérifie si l'utilisateur ne veut pas sortir du programme, auquel cas on sort de la boucle par `last`, on affiche le compteur des questions posées et on sort. Sinon, on incrémente le compteur et on essaye de répondre: on affiche d'abord le nom demandé, puis on essaye de trouver une entrée `$phone{$_}` dans le vecteur associatif. Pour former la réponse, on utilise la consctuction: *test* ?*succes* :*echec*. Si on a trouvé, on affiche la réponse `$phone{$_}`, sinon, la chaîne `"unknown"`.

4.4 Le serveur Internet

Le programme précédent fonctionne bien, mais présente plusieurs inconvénients:

- tout d'abord, il relit toute la base chaque fois qu'on le lance. Ce n'est pas trop grave si elle est de petite taille, mais serait inutilisable avec une "vraie" application (ces exemples proviennent d'une application pour l'astronomie ou la base comportait des dizaines de milliers d'entrées, chacune avec une dizaine de champs: la lecture demandait plusieurs minutes).

– il ne sert qu'à une personne à la fois.

La solution est d'écrire un serveur Internet multitâche. On lance l'application une seule fois, elle attend sur un port tcp convenu (ici, on prendra 2345). Quand une connexion s'établit, on détache un nouveau processus qui la gérera jusqu'à ce qu'elle soit close, et le processus initial attend de nouvelles demandes. De cette manière, le processus initial est toujours prêt à accepter une connexion et il lance autant de processus que nécessaire pour traiter des demandes simultanées. Pour se connecter sur ce serveur, il suffit d'utiliser **telnet** en précisant le port du serveur (et bien sûr son adresse), et on se trouve dans les mêmes conditions qu'avec un terminal dans le cas précédent.

```
# prepare socket stuff (p.344)
($port) = @ARGV;
$port = 2345 unless $port;
$AF_INET = 2;
$SOCK_STREAM = 1;
$sockaddr = 'S n a4 x8';
($name, $aliases, $proto) = getprotobyname('tcp');
$this = pack($sockaddr, $AF_INET, $port, "\0\0\0\0");
socket(S, $AF_INET, $SOCK_STREAM, $proto) || die "socket: $!";
bind(S,$this) || die "bind: $!";
listen(S,5) || die "listen : $!";
# all is ready, loop on connexions
for(;;) {
   accept(NS,S) || die "accept: $!";
   if( fork() == 0) {   # in the child
      select(NS); $| = 1;   # set "hot pipes"
      $query = 0;   # queries number
      print
         "Welcome to this small perl program. Enter q to quit\n> ";
      while (<NS>) {   # try to answer queries
                                          # cleanup and lowercase
         $_ =~ tr/[ \t\r\n]//d ;$_ =~ tr/A-Z/a-z/;
         last if ( $_ eq "q" );         # quit ?
         $query++;
         print $_ ,
             " : " , $phone{$_} ? $phone{$_} : "unknown" , "\n>";
      }
      print $query , " queries, Goodbye...\n";   # game over...
      exit;
   }
   close(NS);      # in the parent: goto 1
}
```

L'initialisation de l'environnement "socket" n'est pas des plus simples[1]: il repose sur trois appels systèmes:

- `socket` qui crée un descripteur de fichier de type `socket` dans la famille Internet (`$AF_INET`), de type "stream" (`SOCK_STREAM`) et suivant le protocole `tcp`. Le numéro de protocole pour `tcp` est récupéré par l'appel `getprotobyname` qui fournit une liste dans laquelle il figure.

- `bind` qui enregistre le socket sur un numéro de port. L'argument de `bind` est une structure de type `sockaddr` qui a été remplie par un `pack`. La structure comprend un entier court non signé (c'est la famille de protocoles), un entier court avec les octets dans l'ordre réseau (c'est le numéro de port), quatre caractères (c'est l'adresse IP) et huit octets à zéro. On la décrit donc à `pack` par `'S n a4 x8'`. L'adresse ip est mise à zéro pour indiquer que l'on accepte des connexions depuis n'importe quelle machine.

- `listen` qui signale au noyau que l'on est prêt à recevoir des connexions sur le socket. Le deuxième paramètre de cet appel est le nombre de connexions qui peuvent être en queue en attendant d'être prises en charge par un `accept`.

Une fois cette machinerie mise en place, les choses deviennent plus simples: on crée une boucle infinie (plus exactement très longue...) dans laquelle on attend une connexion par `accept`. Quand on sort de `accept`, on a l'ancien socket `S` qui peut servir à un nouvel `accept`, et un nouveau socket `NS` qui pointe sur la connexion nouvellement arrivée. L'appel `fork` qui suit crée un nouveau processus (le fils) qui hérite de l'ensemble de l'environnement de l'ancien (le père). En particulier, les deux processus continuent à exécuter le même code, mais:

- dans le père, `fork` donne le numéro de processus du fils. On saute donc le `if`, on ferme le nouveau socket `NS` qui ne nous sert plus à rien ici et on recommence un `accept` pour une nouvelle connexion.

- dans le fils, `fork` donne `0` et on exécute le bloc du `if`. Ce bloc est pratiquement le même que celui du programme précédent en lisant et écrivant sur le socket au lieu du terminal. Il faut simplement prendre la précaution de positionner la variable spéciale `$|` pour le socket de manière à forcer un `flush` à chaque écriture (sinon, les données écrites s'accumulent dans le tampon d'où elles ne sortent que par packets, ce qui donne un dialogue... disons... saccadé).

Tout au long du programme, toutes les fonctions qui peuvent échouer sont bien sûr "gardées" par des `||` die...-.

1. Pour suivre, il faut un minimum de connaissances des fichiers d'en-tête `<sys/socket.h>` et du fonctionnement des sockets

4.5 La version complète

La version précédente fonctionne correctement, mais on peut facilement lui apporter quelques améliorations cosmétiques:

- si on interrompt le programme et qu'on le relance, on obtient le message "bind: socket already in use". Cela est dû au fait que le socket reste dans un état `"TIME_WAIT"` pendant environ une minute et que le noyau refuse que l'on se resserve de cette adresse pendant ce temps. Pour éviter cela, il faut utiliser une option du socket: `SO_REUSEADDR` qui peut être positionnée par la fonction `setsockopt`;

- il serait intéressant de garder une trace des connexions qui arrivent sur le serveur. L'adresse de la machine cliente est fournie par `accept` sous la forme d'une structure `sockaddr`. Il suffit donc d'en extraire l'adresse ip et le numéro de port par `unpack`. On peut donc afficher l'établissement des connexions avec l'adresse du client ainsi que l'heure, ce qui est l'occasion d'écrire un sous-programme donnant l'heure dans le format voulu (`asctime`). De même, on peut afficher l'heure de fin de connexion avec le nombre de requêtes.

- quand le programme est interrompu par `^C`, il serait utile d'avoir un résumé du nombre de connexions et par exemple le temps cpu consommé. Rien de plus simple: en modifiant la variable spéciale `%SIG`, on peut définir un sous-programme qui sera appelé à la réception du signal SIGINT (celui que génère `^C`). Ce sous-programme peut afficher l'heure d'arrêt, le nombre de connexions reçues et le temps cpu consommé. Ce dernier est obtenu par la fonction `times`. Ceci ne doit s'exécuter que dans le processus père, ce que l'on peut tester à l'aide du numéro de processus obtenu par la variable spéciale `$$`. A noter que sur certains systèmes, l'appel système `read` est automatiquement relancé après une interruption: dans ce cas, si on interrompt le père pendant qu'une connexion est établie, il s'arrête, mais le fils continue à répondre!

Le programme suivant inclus toutes ces petites améliorations et peut servir de base à des applications plus complexes.

```
#!/usr/local/bin/perl
# read database
open(DATA, "data") || die "Could't open data file";
while(<DATA>){
   /^\*name:/ && do { split ; $name = $_[1] ; next };
   /^\*phone:/ && do { split ; $phone{$name} = $_[1] ; next };
};
close DATA;
```

```
# report the result
@name = sort keys(%phone);   # build sorted keys list
print $#name+1, " entries read:\n";
foreach $name (@name) {
   printf "%-18s, phone number: %4s\n" , $name , $phone{$name};
};
# return a formated date
sub asctime {
   ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                                   localtime(time);
   $asctime = sprintf("%3s %3s %2d %2d:%2d:%2d %4d",
         (Sun,Mon,Tue,Wed,Thu,Fri,Sat)[$wday],
         (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec)[$mon],
         $mday, $hour, $min, $sec, 1900+$year);
}
# etablish an interrupt handler for ^C
$ppid = $$;   # parrent pid
$SIG{'INT'} = 'ended';   # interrupt handler
sub ended {   # handle ^C at keyboad
   if($$ == $ppid){   # only in the parrent
      print "\n", &asctime,
            " : Stopping after ", $connexion-1, " connexions\n";
      ($usr, $sys) = times;
      printf "Cpu: (%6.2fusr, %6.2fsys)\n", $usr, $sys;
      exit;
   }
}
# prepare socket stuff (p.344)
($port) = @ARGV;
$port = 2345 unless $port;
$AF_INET = 2;
$SOCK_STREAM = 1;
$sockaddr = 'S n a4 x8';
($name, $aliases, $proto) = getprotobyname('tcp');
$this = pack($sockaddr, $AF_INET, $port, "\0\0\0\0");
socket(S, $AF_INET, $SOCK_STREAM, $proto) || die "socket: $!";
$on = pack('I', 1);   # set socket to reuse address
$SOL_SOCKET = 0xffff;
$SO_REUSEADDR = 0x0004;
setsockopt(S, $SOL_SOCKET, $SO_REUSEADDR, $on);
bind(S,$this) || die "bind: $!";
listen(S,5) || die "listen : $!";
$connexion = 0;      # connexions number
```

```
# all is ready, loop on connexions
for(;;) {
   $connexion++;
   ($addr = accept(NS,S)) || die "accept: $!";
   if( fork() == 0) {   # in the child
      select(NS); $| = 1; select(stdout);   # set "hot pipes"
      ($af, $port, $inetaddr) = unpack($sockaddr, $addr);
      @inetaddr = unpack('C4', $inetaddr);
      print &asctime ,
       " : Connexion $connexion: ",
       "From @inetaddr, port $port starting\n";
      $query = 0;   # queries number
      print NS
        "Welcome to this small perl program. Enter q to quit\n> ";
      while (<NS>) {   # try to answer queries
                                          # cleanup and lowercase
         $_ =~ tr/[ \t\r\n]//d ;$_ =~ tr/A-Z/a-z/;
         last if ( $_ eq "q" );         # quit ?
         $query++;
         print NS $_ ,
                " : " , $phone{$_} ? $phone{$_} : "unknown" , "\n>";
      }
      print NS $query , " queries, Goodbye...\n";   # game over...
      print &asctime ,
           " : Connexion $connexion: ended after $query queries\n";
      exit;
   }
   close(NS);      # in the parent: goto 1
}
```

Chapitre 5

Juste pour le fun!

Vous avez déjà remarqué qu'un programme **Perl** pouvait être difficile à comprendre. Alors essayez de prédire ce que fait celui-ci!.

```
#!/usr/local/bin/perl
$_=(32*257*263*3461).(8*9*91372201).(2*3*267*435850691);
$_.=(5*7*811*2689).(2*3*5*31*7285697);s/../printf('%c',$&)/ge;
```

Annexe A

Perl Version 5

La version 5 de **Perl** est une refonte complète de la précédente. Elle reste toutefois compatible avec la version 4 dont tous les programmes devraient continuer à fonctionner.

Les principales nouveautés sont présentées ici:

- de nombreuses améliorations ont été apportées pour rendre les programmes plus lisibles, en particulier pour les variables spéciales. On peut par exemple utiliser `$INPUT_LINE_NUMBER` à la place de `$.` . Les messages d'erreurs sont aussi plus explicites.

- une grande nouveautée est la possibilité d'avoir des références sur une variable (ou un sous-programme). C'est un peu l'analogue des pointeurs en **C** . Comme ces références sont des scalaires, on peut maintenant avoir des listes ou des vecteurs associatifs dont les éléments sont des références sur d'autres listes, etc... On peut ainsi construire des structures de complexité arbitraire.

- dans un **package** , deux sous-programmes particuliers: `BEGIN` et `END` sont invoqués automatiquement respectivement au début et à la fin de l'exécution. Ils sont appelés `constructeur` et `destructeur`.

- le concept de **package** a été étendu à celui de module. Ces derniers exportent en plus leur table de symboles. Au passage, on peut utiliser une nouvelle représentation de ces symboles: `$module::var` au lieu de `$module'var`. Ces notations sont bien sûr cascadables. Les modules sont dénommés avec une extension `.pm`.

- encore une nouvelle notation: pour appeler une fonction:

  ```
  function variable @list;
  ```

on peut maintenant utiliser la notation:

```
variable -> function( @list );
```

- une interface de programmation existe pour appeler du code **C** depuis **Perl** et réciproquement. On peut ainsi construire des modules chargeables dynamiquement en **C** .
- parmi ces modules, on a une interface conforme a POSIX (IEEE 1003.1), une interface avec le gestionnaire d'écrans X11 Tk, et des outils pour mesurer les temps d'exécution de portions de code.
- **Perl** est devenu un langage orienté objet: les objets sont réalisés par des références, les classes par des packages et les méthodes par des sous-programmes. Avec les quelques changements de notations que l'on a vu, cela devient tout naturel...

Utilisé dans l'esprit "programmation objet", **Perl** devient pratiquement un autre langage, avec une apparence de code qui se rapproche du **C++**.

Bibliographie

[1] /usr/man/man1 *man perl*

[2] Larry Wall and Randall L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., 1991.

[3] Randall L. Schwartz, *Learning Perl*, O'Reilly & Associates, Inc., 1993.

Index

Initiation au langage Perl[1]

Noël Giraud
Centre de Calcul In2p3[2]

2 avril 1996

1. Ce document a été composé avec $\LaTeX$
2. 43 Bd. du 11 Novembre 1918, 69622 Villeurbanne Cedex