Table Des Matières

INTRODUCTION 3

DIS, C'EST QUOI PERL? 3
T'AS PAS ENCORE PERL SUR TA MACHINE? 4

1. LES PRINCIPES ESSENTIELS DE PERL 5

1.1. POUR COMMENCER... 7
1.2. LES LISTES 14
1.3. L'INTERPOLATION DE VARIABLE 19
1.4. LA SELECTION PAR LES PATTERNS : LE MATCHING 20
1.5. LES TABLEAUX ASSOCIATIFS 22

2 . PERL EN DETAIL 23

2.1. LES VARIABLES SCALAIRES 25
2.2. LES LISTES ET LES TABLEAUX 31
2.3. STRUCTURES DE CONTROLE 37
2.4. LES TABLEAUX ASSOCIATIFS 41
2.5. LES ENTREES/SORTIES 45
2.6. LES EXPRESSIONS REGULIERES 49
2.7. LES FONCTIONS 59
2.8. DIVERSES STRUCTURES DE CONTROLE 63
2.9. LES MANIPULATEURS DE FICHIERS ET LES TESTS SUR LES FICHIERS 67
2.10. LES FORMATS 73
2.11. LES REPERTOIRES 81
2.12. MANIPULATION DE FICHIERS ET DE REPERTOIRES 85
2.13. LA GESTION DE PROCESSUS 91
2.14. AUTRES TRANSFORMATIONS SUR LES DONNEES 97
2.15. ACCES AUX DONNEES SYSTEME 103
2.16. MANIPULATION DE BASES DE DONNEES UTILISATEUR 109
2.17. CONVERTIR D'AUTRES LANGAGES EN PERL 115
2.18. COMPLEMENTS 117

3. EXEMPLE D'APPLICATION 123

3.1. UN SYSTEME DE RESERVATION DE VOLS 125
3.2. LE PROGRAMME PERL 127
3.3. RESULTATS 137

4. EXERCICES 143

4.1. ENONCES 145
4.2. SOLUTIONS 149

ANNEXE : LE GUIDE DE REFERENCE DE PERL 157

INTRODUCTION

DIS, C'EST QUOI PERL?

Créé tout d'abord pour traiter efficacement du texte grâce au «matching», Perl s'est développé tant et si bien qu'il est aujourd'hui également un puissant langage de manipulation de fichiers: renommage, déplacement, modification des droits; et de processus: création, destruction, communication, contrôle de flux.
C'est même un langage réseau, puisqu'il offre la possibilité de communiquer avec des processus se trouvant sur d'autres machines, par l'intermédiaire de sockets.

Perl permet donc d'exécuter d'une façon plus concise et lisible des tâches ordinairement effectuées par un programme en C ou par l'un des shells. C ne permet pas de faire tout ce que l'on peut faire avec un shell, et vice versa. Perl comble cette lacune: c'est une passerelle entre le C et les shells.

Perl fonctionne sous divers systèmes d'exploitation; on s'intéressera en particulier à son fonctionnement sous Unix.

C'est un langage simple. Les structures et les types qu'il emploie sont faciles à comprendre et à utiliser. Pas besoin de connaître Perl en détail pour commencer à écrire de petits programmes sympas. Pas de formule magique, non plus, pour compiler un programme Perl: il s'exécute comme un script shell.
Cependant, Perl est également un langage riche, puisqu'il emprunte ses fonctionnalités au C, aux shells mais aussi à *sed* et à *awk*. Il permet de développer des programmes rapidement. En tant que langage script interprété, il fournit un « feedback » immédiat si une erreur est détectée. De plus, un debugger symbolique est intégré, qui comprend n'importe quelle expression Perl puisqu'il est écrit en Perl.

Dans ce document, les principes essentiels de Perl dans sa **version 4.0** seront tout d'abord rapidement présentés à travers quelques exemples simples. Nous verrons ensuite plus en détail les structures et les types utilisés par ce langage. Puis nous verrons un exemple de programmation en Perl.
Enfin, nous compléterons cette approche avec quelques exercices et un guide de référence, bien pratique pour vérifier rapidement certains points en cas de doute.

Parallèlement , il existe une page man pour Perl: perl(1). Le newsgroup Usenet sur Perl est une source d'informations supplémentaire. Il n'y a pas de question trop bête pour être posée... il n'y a que des questions trop bêtes pour qu'on y réponde!

T'AS PAS ENCORE PERL SUR TA MACHINE?

Pas de panique! La gratuité étant la seule monnaie de l'art, Perl est gratuit.

Il est disponible par ftp anonyme sur les machines suivantes:

uunet.uu.net	192.48.96.2
tut.cis.ohio-state.edu	128.146.8.60
jpl-devvax.jpl.nasa.gov	128.149.1.143

Pour ceux qui ne sont pas sur Internet, Perl est disponible par uucp anonyme à partir de uunet et de osu-cis.

Sur uunet.uu.net, Perl se trouve dans un unique fichier archive: gnu/perl-3.0.41.tar.Z. Sur jpl-devvax.jpl.nasa.gov, il est dans le répertoire pub/perl.3.0; sur tut.cic.ohio-state.edu, il est dans perl/3.0 (bientôt ce sera 4.0 pour les deux).

1. LES PRINCIPES ESSENTIELS DE PERL

1.1. POUR COMMENCER...

Perl est facile à apprendre et simple d'utilisation. Il n'y a pas grand chose à dire avant de dire effectivement ce que l'on veut dire. Par exemple, il n'y a aucune déclaration à effectuer avant d'écrire du code.

1.1.1. Les manipulateurs de fichiers (filehandle)

Voici un exemple de script Perl simple:

```
print "Donnez un nombre: \n";
$nombre = <STDIN>;
print "Nombre lu : $nombre \n";
```

Ce script permet de lire une entrée de l'utilisateur avec un retour d'information. La variable `$nombre` reçoit ce que l'utilisateur vient de taper; en effet, le symbole `<STDIN>` indique simplement qu'il faut lire la ligne courante de l'entrée standard.
La commande **print** possède un paramètre optionnel indiquant la redirection de la sortie; `STDOUT` est la sortie par défaut. On aurait aussi bien pu écrire:

```
print STDOUT "Nombre lu : $nombre\n";
```

Ainsi `STDIN` est le nom de l'entrée, `STDOUT` celui de la sortie et `STDERR` celui du fichier erreur.
`STDIN`, `STDOUT`, `STDERR` sont des manipulateurs de fichier, ou *filehandles*, qui existent en permanence dans une application Perl. Un filehandle est un nom que l'on donne à un fichier, un pipe ou une socket afin de pouvoir y réaliser facilement des entrées/sorties. Pour créer un filehandle et l'associer à un fichier, on utilise la fonction **open**. Exemples:

```
open( MANIPULE, "NomDeMonFichier");
open( MANIPULE, "> nomfic");
open( MANIPULE, "| commande de sortie du pipe");
```

`MANIPULE` donne alors accès au fichier ou au pipe auquel il a été associé jusqu'à ce que cette association soit fermée ou que l'on effectue un nouvel **open** sur ce filehandle. `STDIN`, `STDOUT`, `STDERR` peuvent être fermés et rouverts.

1.1.2. Les variables

Dans notre premier exemple, nous n'avons donc pas eu à déclarer la variable `$nombre`. En effet, le symbole **$** indique à Perl que `$nombre` peut contenir une variable: un nombre ou une chaîne de caractères. Nous appellerons cela une variable scalaire.
Il y a d'autres variables:

VARIABLE	un filehandle
$VARIABLE	une variable scalaire
@VARIABLE	un tableau indexé par des nombres
%VARIABLE	un tableau indexé par des chaînes de caractères
&VARIABLE	une subroutine
*VARIABLE	tout ce qui s'appelle VARIABLE

On peut affecter des variables, modifier leur valeur; exemples:

```
$nombre = 4278;                  # un entier
$chaine = "coucou";              # une chaîne de caractères
$commande = `who | grep toto`;   # une commande
```

En Perl, un commentaire est précédé par **#**.
Une ligne d'instruction Perl se termine toujours par **;** .

Si une variable n'est pas initialisée par l'utilisateur, elle est automatiquement initialisée à la valeur nulle (0). Une variable est évaluée comme un nombre, une chaîne, un booléen ou un élément de tableau en fonction du contexte. Perl effectue la conversion automatiquement. Ainsi:

```
$magic = "45";
print $magic+1, "\n";
```

Le résultat de l'exécution de ce script est "46".

1.1.3. Comment lancer un script Perl?

Il y a plusieurs manières pour lancer votre premier script!

- En ligne :

```
> perl  -e  'print "coucou \n" ; '
```

- Si on utilise un fichier pour stocker le script:

```
> perl nomfic
```

- Si votre système supporte **#!**, vous pouvez spécifier le nom de l'interpréteur en mettant en tête du script l'incantation:

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

  Le fichier contenant le script doit être exécutable (`chmod u+x nomfic`); pour exécuter le script, on peut ensuite taper:

```
> nomfic
```

- En dernier recours, on peut remplacer l'incantation ci-dessus par la ligne:

```
eval  '/usr/bin/perl  -S  $0  ${ 1 + "$@" }'  if  0 ;
```

1.1.4. Une variable sympa : $_

La variable `$_` est affectée par un grand nombre d'opérations.
Elle permet d'accéder à la ligne courante d'un fichier que l'on est en train de consulter (cf. **$0** en *awk*).

1.1.5. Comment exprimer des conditions?

Perl dispose d'une boucle **while** dont le corps s'exécute tant que la condition est vraie. Exemple:

```
!#/usr/bin/perl
$credit = 10;
while ($credit != 0)
{ print  "J'offre un café à Rouv ! \n  ";
  $credit - = 2.5; }
print " J'ai plus de sous! \n ";
```

La condition `$credit!=0` est vraie tant que `$credit` ne vaut pas 0. L'opérateur **!=** (différent) compare deux valeurs numériques et retourne *vrai* si la première est différente de la seconde; sinon il retourne *faux*.

Perl n'a pas de type booléen; une valeur est vraie si elle est différente de 0 ou "0" ou "" (chaîne vide). Les opérateurs relationnels tels que ==, !=, <, ... retournent 1 pour *vrai*, 0 pour *faux*. D'autres opérateurs peuvent retourner d'autres valeurs non nulles pour *vrai* et la chaîne vide pour *faux*.
Donc tout ce qui a une valeur peut être utilisé comme condition d'un *while*; par exemple, une affectation, puisqu'elle retourne la valeur finale de la variable affectée:

```
!#/usr/bin/perl
while ( $_ = <ARGV>)
{ print $_; }
```

Ce script permet de faire afficher les unes à la suite des autres toutes les lignes des fichiers passés en argument dans la ligne de commande. C'est l'équivalent de la commande shell *cat*.
`ARGV` est un filehandle; `<ARGV>` est l'entrée.
Si ce script est enregistré sous le nom *equivcat* et que *toto*, *tutu* et *titi* sont des fichiers, la ligne de commande:

```
> equivcat  toto  tutu  titi
```

a pour résultat le stockage de la première ligne de *toto* dans `$_` et son impression sur la sortie standard. Puis la deuxième ligne de *toto* est rangée dans `$_` et affichée. Et ainsi de suite jusqu'à épuisement de *toto*. L'opération est alors réitérée avec *tutu*, puis avec *titi*.

Le script précédent peut encore être simplifié:

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

On obtient le même résultat que précédemment. En effet, `<ARGV>` et `<>` représentent la même chose; de plus, un symbole désignant une entrée, utilisé seul dans une condition de boucle `while`, affecte automatiquement la variable `$_`; enfin, si l'on ne précise pas de paramètre à `print`, le contenu de `$_` est affiché par défaut.

L'autre façon d'exprimer une condition en Perl est d'utiliser **if**. L'exemple suivant est l'équivalent du *grep* du shell:

```
!#/usr/bin/perl
$pattern = shift( @ARGV );
while (<>)
{ if ( /$pattern/)
   { print ; }
}
```

Le premier argument passé à la commande est le pattern à rechercher; il est stocké dans la variable `$pattern`. Le reste des arguments est considéré comme des noms de fichiers; c'est dans leur contenu que le pattern va être recherché. Les lignes dans lesquelles `$pattern` a été reconnu sont ensuite affichées.

Que se passe-t-il si la recherche est effectuée dans une ligne vide?
Une ligne vide contient en fait le caractère "\n", dont la valeur est *vrai*; elle n'est donc pas vraiment vide! On continue alors la recherche dans la ligne suivante. Toute ligne se termine d'ailleurs par "\n".
La fonction `print` n'effectue pas de retour à la ligne automatique; on est donc obligé de spécifier si on le désire.

1.1.6. La vérité sur les tests

Les opérateurs de test ne sont pas les mêmes suivant que l'on compare des valeurs numériques ou des chaînes de caractères. Ces opérateurs retournent 1 si la comparaison est vraie, 0 sinon.

Opérateur de test sur les nombres	**Opérateur de test sur les chaînes**	**Signification**
==	eq	égal
!=	ne	différent
>	gt	supérieur
>=	ge	supérieur ou égal
<	lt	inférieur
<=	le	inférieur ou égal
<=>	cmp	différent, avec retour du signe

Il faut donc faire attention quand on écrit un test; exemple:

```
$a = "5";
$b = "10";
if ( $a < $b )    { print " $a < $b \n";}
if ( $a gt $b )   { print " $a gt $b \n";}
```

Le résultat de ce script est:

```
5 < 10          # comparaison numérique
5 gt 10         # car la valeur ASCII de  1 est plus petite que celle de 5
```

1.1.7. Les opérateurs scalaires

Voici une liste d'opérateurs, parmi les plus importants:

- Sélection par les patterns

`$a =~ /pat/`	matching; retourne vrai si `$a` contient le pattern `pat`
`$a =~ s/p/m/`	substitution des `p` dans `$a` par des m
`$a =~ tr/a-z/A-Z/`	conversion des minuscules en majuscules

- Opérateurs logiques

`$a && $b`	et logique
`$a || $b`	ou logique
`! $a`	négation

- Opérateurs arithmétiques

`$a + $b`	addition
`$a - $b`	soustraction
`$a * $b`	multiplication
`$a / $b`	division
`$a % $b`	modulo
`$a ** $b`	`$a` exposant `$b`
`++ $a, $a ++`	incrémentation
`-- $a, $a --`	décrémentation
`rand($a)`	tire aléatoirement une valeur entre 0 et `$a`

- Opérateurs sur les chaînes de caractères

`$a . $b`	concaténation
`$a x $b`	`$a` répétée `$b` fois
`substr( $a, $i, $l)`	extraction d'une sous chaîne de `$a` commençant à la position `$i` et ayant pour longueur `$l`
`index( $a, $b)`	position de la chaîne `$b` dans `$a`

- Opérateurs pour l'affectation

`$a = $b`	affectation
`$a +=$b`	addition et affectation
`$a -=$b`	soustraction et affectation
`$a .=$b`	concaténation et affectation

- Opérations sur les fichiers

`-r $a`	retourne vrai si `$a` est autorisé en lecture
`-w $a`	retourne vrai si `$a` est autorisé en écriture
`-d $a`	retourne vrai si `$a` est un répertoire
`-f $a`	retourne vrai si `$a` est un fichier régulier
`-T $a`	retourne vrai si `$a` est un fichier texte

Les opérateurs **&&** et || sont des opérateurs court-circuit. Cela signifie que si l'évaluation de la partie gauche est suffisante pour décider de la valeur finale de l'expression, la partie droite n'est pas évaluée.

Il y a bien sûr d'autres opérateurs, mais ceux-ci suffisent pour commencer!

1.1.8. Compléments

Voici un exemple qui reprend ce que nous avons vu jusqu'ici, plus quelques petites choses nouvelles:

```
#!/usr/bin/perl
open(FIND," find . -print |")||die"find n'a pas démarré:$!\ n";
FILE:
while ( $nomfic = <FIND>)
{  chop $nomfic;
   next FILE unless -T $nomfic;
   if ( ! open( FICTEXT, $nomfic ) )
   {  print STDERR  "Ouverture de $nomfic impossible ";
       next FILE;
    }
    while ( <FICTEXT> )
   { foreach $mot ( @ARGV )
      { if ( index($_, $mot) >= 0)
         {  print $nomfic, " \n ";
            next FILE;
         }
      }
    }
}
```

Le but de ce programme est de retrouver, dans les fichiers texte du répertoire courant, les arguments passés lors de l'appel du script.

La variable **$!** contient le message d'erreur produit par le système lors de l'échec d'ouverture du filehandle `FIND`.
Il est possible d'étiqueter une boucle; le label `FILE` permet à **next** de faire appel de nouveau à la boucle. L'opérateur `next` indique la boucle à effectuer en spécifiant son étiquette.
foreach est une structure de boucle puisqu'elle permet à `$mot` de prendre pour valeur les éléments de `@ARGV`, successivement.
La fonction **index** retourne la position de `$mot` dans la ligne courante s'il a été trouvé, -1 sinon.

1.2. LES LISTES

En plus des opérateurs scalaires, Perl dispose de nombreux opérateurs travaillant sur les listes. La force de Perl, c'est de pouvoir agir sur plusieurs objets en une seule ligne de commande.

1.2.1. Qu'est-ce qu'une liste?

Une liste est tout simplement un ensemble **ordonné** de scalaires. On peut donc avoir des listes de nombres, de chaînes de caractères, ou un mélange des deux. Une liste portant un nom est appelée tableau.
Comme une liste est ordonnée, on peut donc parler de son premier élément, de son deuxième élément ..., et de son dernier élément. Les éléments d'une liste sont donc accessibles séparément. Le premier élément porte le numéro 0, le second le numéro 1, etc.

Lorsqu'on tape une commande Perl, ses arguments sont rangés dans le tableau `@ARGV`. Donc

```
@ARGV[0]
```

est le premier argument et

```
@ARGV[$#ARGV]
```

est le dernier, sachant que `$#ARGV` donne accès à la position du dernier élément.

L'opérateur **range**, noté **.. ,** produit une liste de valeurs comprises entre les valeurs des opérandes gauche et droit. Par exemple, le script suivant permet de faire afficher tous les arguments de la commande faisant appel à lui :

```
#!/usr/bin/perl
foreach $i (0..$#ARGV)
{ print $ARGV[$i];
   if ($i == $#ARGV)
   { print "\n"; }
   else
   { print " " ; }
}
```

L'opérateur de construction d'une liste est la virgule, séparateur des différents éléments. Perl s'attend en général à trouver une liste entre parenthèses; par exemple :

```
@speinf =
      ('seb','cricri','pascal','xav','gilles','hugo','sissou');
@chiffre = ( 1, 2, 3, 4, 5, 6, 7, 8, 9 );
# même chose que ( 1..9 )
@etudiant = ( $no_etudiant, $nom, $prenom, $adresse );
```

Les parenthèses sont nécessaires ici; en effet, la virgule est moins prioritaire que l'affectation.

Une liste parenthésée de scalaires peut être affectée directement par une liste de valeurs, comme le montrent les exemples suivants :

```
( $no_compte, $debit, $credit ) = ( 14578963, 2000, 0 );
($nom,$passwd,$uid,$gid,$gcos,$home,$shell)=split(/:/,<PASSWD>);
( $tab[0], $tab[1], $tab[2], $tab[3] ) = @speinf;
```

split crée une liste de chaînes de caractères en scindant la ligne courante du fichier que `PASSWD` permet de manipuler, à chaque occurrence des **:** . Le troisième exemple montre que l'on peut affecter une liste à partir d'un tableau, quelque soit la taille de celui-ci. Si le tableau est trop grand, les dernières valeurs ne sont pas prises en compte. S'il est trop petit, les valeurs non affectées de la liste demeurent indéfinies.

La taille d'un tableau peut être modifiée: des éléments peuvent être ajoutés, supprimés. La fonction **splice** permet de manipuler ainsi des tableaux. Sa syntaxe est :

```
splice ( ARRAY, INDEX, LONGUEUR [, LIST] )
```

`ARRAY` est le tableau auquel s'applique **splice**. `INDEX` est le premier élément de `ARRAY` qui va disparaître. `LONGUEUR` est le nombre d'éléments de `ARRAY` qui vont disparaître. `LIST` est la liste éventuelle dont les éléments seront insérés dans `ARRAY` à partir de la position `INDEX`. Un exemple d'utilisation :

```
@arbre = ( 'sapin', 'épicéa', 'chêne', 'peuplier', 'hêtre' );
@modif = splice ( @arbre, 1, 3, 'boulot', 'palmier' );
```

Le résultat de cette fonction est une modification du tableau `@arbre`; on a maintenant :

```
@arbre = ( 'sapin', 'boulot', 'palmier', 'hêtre' );
```

`splice` retourne la liste des éléments de `@arbre` qu'elle a modifiés :

```
@modif = ( 'épicéa', 'chêne', 'peuplier' );
```

Les fonctions suivantes sont utiles et simples pour travailler avec des listes:

```
$a = shift(@tab);       # $a contient le premier élément de @tab;
                        @tab a perdu cet élément
$b = pop(@tab);         # $a contient le dernier élément de @tab;
                        @tab a perdu cet élément
unshift(@tab, $a );     # insertion de $a en tête de @tab
push(@tab, $b );        # insertion de $b en fin de @tab
```

1.2.2. S'il vous plaît! On ne s'endort pas!

Il y a deux choses importantes à retenir sur les listes.

Premièrement, il est important pour certains opérateurs de savoir s'ils sont évalués comme un élément d'une liste. Ils retournent alors à l'intérieur d'une liste des valeurs différentes de celles qu'ils auraient retournées ailleurs. Prenons l'exemple de l'opérateur d'entrée <> :

```
$a = <STDIN>;           # $a contient la ligne courante de
                        l'entrée standard
@tab = <STDIN>;         # @tab contient toutes les lignes qu'il
                        reste dans l'entrée standard
```

Deuxièmement, une liste (la structure syntaxique) concatène toujours ses valeurs scalaires et ses tableaux, comme s'il s'agissait d'une seule et longue liste. Chaque élément de la liste est évalué comme une expression dans un contexte de tableau, et s'il produit une liste, les valeurs de cette liste sont ajoutées à la liste de départ, comme s'elles avaient été spécifiées séparément. Un exemple, pour éclaircir tout çà :

```
@a = ( 1 .. 3);
@b = (78,45,@a,92);     # équivaut @b = (78,45,1,2,3,92)

@a = ();                # liste vide
@b = ( 0, @a, 4 );      # équivaut à @b = ( 0, 4 )
```

Comme `@a` est évalué dans un contexte de tableau, il renvoie la liste qu'il contient. S'il était évalué dans un contexte scalaire, `@a` renverrait le nombre d'éléments qu'il comporte. Exemple:

```
 while (@a)         # la boucle est effectuée autant de fois que @a
                    contient d'éléments.
{ #code }
```

1.2.3. Quelques opérateurs usuels sur les listes

On utilise le tableau suivant comme argument :

```
@speinf =
       ('seb','cricri','pascal','xav','gilles','hugo','sissou');
```

Opérations	**Résultats**
`$speinf[2]`	`('pascal')`
`@speinf[3, 6]`	`('xav','sissou')`
`sort` `@speinf`	`('cricri','gilles','hugo','pascal', 'seb','sissou' ,'xav')`
`reverse` `@speinf`	`('sissou','hugo','gilles','xav', 'pascal', 'cricri', 'seb')`
`reverse sort` `@speinf`	`('xav','sissou','seb','pascal','hugo', 'gilles','cricri')`
`grep``(/ill/, @speinf)`	`('gilles')`
`3..7`	`( 3, 4, 5, 6, 7 )`
`reverse` `3..7`	`( 7, 6, 5, 4, 3 )`
`@speinf[0..2]`	`('seb', 'cricri', 'pascal')`

1.2.4. Et on fait quoi avec les listes?

Forts de tout ce que nous venons d'apprendre, nous allons passer à un exemple plus pratique d'utilisation des listes. Imaginons qu'on veuille récupérer les noms de tous les fichiers texte de notre compte.
Il suffit de faire appel au script suivant, grâce à la ligne de commande:

```
> nom_du_script *
```

* donne les noms de tous les fichiers et répertoires du compte.

```
#!/usr/bin/perl
while (@ARGV)
{ $fichier = shift @ARGV;
   push(@fichier_texte, $fichier) if -T $fichier; }
print join(' ', @fichier_texte), "\n";
```

Dans le `while`, `@ARGV` est évalué dans un contexte scalaire. Il retourne donc le nombre d'arguments passés au script (ici, le nombre de fichiers et répertoires du compte). La variable `$fichier` reçoit le premier élément de la liste `@ARGV`. `@ARGV` perd cet élément. Si `$fichier` est effectivement un fichier texte (test avec **-T**), alors il est rangé à la fin de la liste `@fichier_texte`.

La fonction **join** permet ensuite de constituer une chaîne de caractères où les éléments de `@fichier_texte` sont séparés par un espace.

Autre possibilité :

```
#!/usr/bin/perl
print join(' ', grep(-T, @ARGV)), "\n";
```

La fonction **grep** de Perl est plus puissante que la fonction d'Unix. En effet, elle accepte non seulement les expressions régulières comme critère, mais aussi les expressions Perl.

Si on avait voulu la liste des fichiers texte classés par ordre alphabétique, on aurait écrit :

```
#!/usr/bin/perl
print join(' ', sort grep(-T, @ARGV)), "\n";
```

1.3. L'INTERPOLATION DE VARIABLE

Considérons les lignes de code suivantes :

```
print "$var \n";
print "@ARGV \n";
```

Ces commandes interpolent respectivement le scalaire `$var` et le tableau `@ARGV` en une chaîne de caractères (par défaut, les éléments de `@ARGV` seront séparés par un espace).
Attention, l'interpolation de variable n'a pas lieu entre quotes simples. Pour mieux comprendre, prenons un exemple; soit le script *affícharg*, auquel on passe les arguments suivants:

```
> afficharg pascal gilles xav
```

Si dans *affícharg* on utilise les commandes suivantes, on obtient les résultats suivants:

```
print "@ARGV";              pascal gilles xav
print @ARGV;                pascalgillesxav
print '@ARGV\n';            @ARGV\n
```

L'interpolation de variable est également exécutée à l'intérieur de backquotes, mais elle s'accompagne de l'exécution de la chaîne contenue dans les backquotes comme d'une commande dont le résultat peut être affecté:

```
$lignes = `wc -l $nomfic`;
```

Si les caractères suivant le nom d'une variable à interpoler peuvent prêter à confusion, il faut délimiter ce nom avec des accolades; exemple:

```
$fred = "pay"; $fredday = "wrong!";
$barney = "It's $fredday";                  # "It's wrong!"
$barney = "It's ${fred}day";                # "It's payday"
```

1.4. LA SELECTION PAR LES PATTERNS : LE MATCHING

Perl dispose de deux opérateurs de sélection par les patterns:

- **//**, opérateur de sélection; il recherche le pattern entre // dans une chaîne de caractères et retourne *vrai* ou *faux* suivant que le pattern a été trouvé ou non.

- **s///**, opérateur de substitution; il fait la même chose, mais, en plus, remplace la partie matchée de la chaîne par ce qui se trouve entre les deux derniers délimiteurs.

Les patterns sont spécifiés en utilisant les expressions régulières. En voici quelques exemples:

.	matche n'importe quel caractère sauf newline
[a-z0-9]	matche un caractère de cet ensemble
[^a-z0-9]	matche tout caractère ne figurant pas dans cet ensemble
\d | \D	matche un chiffre | matche tout sauf un chiffre
\w | \W	matche un caractère alphanumérique | contraire
\s | \S	matche un caractère d'espacement | contraire
\n | \r | \t | \f | \b	matchent respectivement newline, return, tab, formfeed, backspace (à l'intérieur de crochets seulement)
\0 | \000	matchent le caractère nul
nnn	matche le caractère ASCII ayant cette valeur octale
metachar	matche le caractère lui-même (| , \. , *, ...)
(abc)	enregistre la sélection
\1	matche le premier pattern contenu dans la sélection
\2 | \3	matche le deuxième | troisième pattern contenu dans la sélection
x?	matche 0 ou 1 *x* où *x* est un des motifs ci-dessus
*x**	matche 0 ou plusieurs *x*
x+	matche 1 ou plusieurs *x*
x{m, n}	matche au moins m *x* et au plus n
abc	matche tous les a, b, c
ok | coucou | he	matche ok ou coucou ou he
\b | \B	matche un délimiteur de mots | contraire
^	donne accès au début de ligne ou de chaîne
$	donne accès à la fin de ligne ou de chaîne

Lorsque l'on fait de la sélection par pattern, il y a également des variables magiques qui sont affectées lorsque le matching réussit:

- **$&**, contient le pattern recherché
- **$`**, contient tout ce qu'il y a avant la sélection
- **$'**, contient tout ce qu'il y a après la sélection
- **$1**, **$2**, ..., **$9** contiennent les patterns matchés

Un exemple de matching en Perl; imaginons que l'on dispose d'un fichier dont la ligne courante (contenue dans $_) est :

```
Date : 3 juin 1995 13:26:00 GMT
```

Cette ligne peut être décomposée comme suit:

```
/^Date : (\d+)(\w+)(\d+)(\d+):(\d+):(\d+)(.*)$/;
$jour = $1;                 # récupère "3"
$mois = $2;                 # récupère "juin"
$annee = $3;                # récupère "1995"
$heure = $4;                # récupère "13"
$minute = $5;               # récupère "26"
$seconde = $6;              # récupère "00"
$reference = $7;            # récupère "GMT"
```

Si on avait fait les regroupements suivants:

```
/^Date : ((\d+)(\w+)(\d+))((\d+):(\d+):(\d+))(.*)$/
```

on aurait la décomposition:

```
$date = $1;                 # récupère "3 juin 1995"
$jour = $2;
$mois = $3;
$annee = $4;
$temps = $5;                # récupère "13:26:00"
$heure = $6;
$minute = $7;
$seconde = $8;
$reference = $9;
```

1.5. LES TABLEAUX ASSOCIATIFS

On peut dire que c'est la partie la plus importante de cette présentation de Perl, puisque c'est la propriété principale de ce langage. Les tableaux associatifs permettent de mettre en place des structures récursives telles que les arbres.

Un tableau associatif est une liste constituée de paires de valeurs indexées par la première valeur de chaque paire, appelée clé. Les clés sont rangées dans une table de hachage, ce qui permet de consulter rapidement le tableau auquel elles sont associées, quelque soit sa taille.

Un tableau associatif est déclaré par le symbole **%**. Exemple:

```
%caracteristiques= (   'xav', 'discret',
                       'cricri', 'marrant',
                       'seb', 'râleur',
                       'gilles', 'emporté',
                       'pascal', 'cultivé',
                       'hugo', 'délire',
                       'sissou', 'bavarde' );
```

On utilise donc la même notation que pour les tableaux simples.
Lors de l'évaluation de `%caracteristiques` dans un contexte de tableau, on n'obtiendra pas forcément les paires dans cet ordre (cela dépend de la fonction de hachage).

Pour accéder à un élément d'un tableau associatif, on utilise les accolades et la clé:

```
$caract = $caracteristiques{'xav'};    # $caract prend la valeur
                                       'discret'
$home = $ENV{'HOME'};
$SIG{'UP'} = 'IGNORE';
```

Les deux derniers tableaux associatifs cités font référence à deux tableaux propres à Perl:
- **%ENV** permet de consulter et fixer les variables d'environnement de l'utilisateur;
- **%SIG** permet de fixer les valeurs des signaux système.

Les tableaux associatifs permettent de lier un ensemble de chaînes à un autre ensemble de chaînes. Ainsi, ils se rapprochent des pointeurs disponibles dans les autres langages.

La fonction **keys** permet d'accéder à la liste des clés d'un tableau associatif. Cette fonction est souvent utilisée comme suit:

```
foreach $key ( sort keys( %tab_nom ) )
{...   }
```

Les tableaux associatifs sont des structures puissantes qui permettent des consultations rapides et qui nécessitent peu de code.

2 . PERL EN DETAIL

2.1. LES VARIABLES SCALAIRES

Nous avons vu plus haut que pour Perl les variables scalaires sont des nombres ou des chaînes de caractères. Des opérateurs permettent de les manipuler: elles peuvent être affectées, lues et écrites dans des fichiers.

2.1.1. Les nombres

Une variable scalaire peut contenir un entier ou un décimal. De toute façon, Perl utilise la même représentation pour les deux: flottant double précision (l'équivalent de *double* en C).

Un *littéral* est la façon dont une valeur est représentée dans le texte d'un programme Perl. Perl accepte tous les littéraux flottants disponibles en C. Exemples:

```
1.435
+3.78
-23e-4
45.7E7
```

Les littéraux entiers sont représentés simplement par:

```
45
-67
```

Ils ne doivent pas commencer par 0; en effet, Perl reconnaît les littéraux octaux et hexadécimaux, dont les représentations commencent respectivement par 0 et 0x ou 0X. Les chiffres hexadécimaux A à F représentent les valeurs 10 à 15. Exemples:

```
0377        # 377 octal, équivaut à 255 en décimal
-0xff       # -FF hexadécimal, équivaut à 255 en décimal
```

2.1.2. Les chaînes de caractères

Ce sont des suites de caractères pris dans l'ensemble des 256 caractères codés sur 8 bits. La plus petite chaîne est la chaîne nulle; la plus longue remplirait toute la mémoire disponible. Comme les nombres, les chaînes ont une représentation littérale; il y en a deux en fait: entre simples quotes et entre doubles quotes.

- Les simples quotes ne font pas partie de la chaîne de caractères qu'elles encadrent; elles sont là pour indiquer à Perl le début et la fin de la chaîne. Une chaîne entre simples quotes peut contenir n'importe quel caractère; il y a cependant deux exceptions: pour avoir une simple quote ou un antislash dans une chaîne entre simples quotes, ceux-ci doivent être précédés par un antislash. Exemples:

```
'coucou'              # 6 caractères: c, o, u, c, o, u
'c\'est pas gagné! '  # c'est pas gagné
'A bientôt \n'        # A bientôt \n
```

  \n entre simples quotes n'est pas interprété comme un retour à la ligne mais comme deux caractères, \ et n.

- Une chaîne entre doubles quotes est l'équivalent d'une chaîne de caractères en C. L'antislash permet donc de spécifier certains caractères de contrôle ou tout caractère par ses représentations octale et hexadécimale. Exemples:

```
"coucou \n"      # coucou, un espace et un retour à la ligne
"nouveau \177"   # nouveau , un espace et le caractère delete
```

  Un antislash peut précéder de nombreux caractères pour signifier différentes choses, comme le montre la table suivante:

\n	newline
\r	return
\t	tab
\f	formfeed
\b	backspace
\v	vertical tab
\a	bell
\e	escape
\007	n'importe quelle valeur octale ASCII (ici, 007=bell)
\x7f	n'importe quelle valeur hexadécimale ASCII (ici, 7f=delete)
\cC	n'importe quel caractère de contrôle (ici, control C)
\\	antislash
\l	mettre la lettre suivante en minuscule
\L	mettre toutes les lettres suivantes en minuscule, jusqu'à \E
\u	mettre la lettre suivante en majuscule
\U	mettre toutes les lettre suivantes en majuscule, jusqu'à \E

\E	permet d'achever \L et \U
\"	double quote

A l'intérieur des doubles quotes, l'interpolation de variable peut avoir lieu, comme nous l'avons annoncé au paragraphe 1.3. Pour empêcher la substitution, on peut utiliser les simples quotes ou faire précéder le caractère $ par un antislash; exemple:

```
$a = "Seb";
$b = "Salut " . '$a';      # $b contient "Salut $a"
$b = "Salut \$a";          # idem
```

2.1.3. Les opérateurs

Une liste d'opérateurs sur les variables scalaires est disponible au paragraphe 1.1.7.
Perl dispose des opérateurs arithmétiques que l'on trouve généralement dans des langages tels que le C. Mais il possède également l'opérateur d'exponentiation ** :

```
3 ** 7      # 3 à la puissance 7
```

On trouve aussi des opérateurs sur les chaînes de caractères:

- l'opérateur **.** , qui permet la concaténation de deux chaînes; exemple:

```
"hello " . "world";    # équivaut à "hello word"
```

- les opérateurs de comparaison de chaînes de caractères; ils sont énumérés dans le chapitre 1.1.6.

- l'opérateur **x**, qui permet de répéter une chaîne autant de fois que l'indique la valeur de l'opérande droit; exemple:

```
"coucou" x 4           # équivaut à "coucoucoucoucoucoucoucou"
"eh" x (3+1)           # équivaut à "eh" x 4, soit "eheheheh"
```

Les parenthèses permettent l'évaluation de l'expression qu'elles renferment, avant l'exécution de la répétition. La partie entière de l'opérande droit est calculée et utilisée pour la répétition.

La multiplication et la division sont prioritaires par rapport à l'addition et la soustraction. L'usage de parenthèses permet de modifier les priorités.

Si une chaîne est utilisée comme opérande d'un opérateur numérique, Perl effectue automatiquement la conversion, de même si une valeur numérique a été utilisée là où une chaîne était attendue.

Pour donner une valeur à une variable, on utilise l'affectation:

```
$a = 3;
$b = $b * 2;
```

Une affectation scalaire a une valeur; cette valeur est le nombre affecté: `$a = 3` a pour valeur 3. C'est pratique pour copier la même valeur dans plusieurs variables:

```
$b = 4 + ($a = 3);          # $a contient 3 et $b contient 7
$c = $d = 5;                # $c et $d contiennent 5
```

Perl dispose d'opérateurs d'affectation binaires:

```
$a += 3;          équivaut à        $a = $a + 3 ;
$b *= $a;         équivaut à        $b = $b * $a ;
$str .= " ";      équivaut à        $str = $str . " ";
```

Perl dispose aussi de l'auto-incrémentation et de l'auto-décrémentation, qui peuvent être utilisées sous forme de préfixe ou de suffixe:

```
++$a;             équivaut à        $a += 1;
$d = $c = 17;
$e = ++ $d;      # $d et $e valent tous les deux 18
$e = $c --;      # $e vaut 17 et $c vaut 16
```

2.1.4. Les noms de variables scalaires

Le nom d'une variable est constant tout au long d'un programme alors que la valeur qu'elle contient varie.
Les noms de variables scalaires commencent par **$**, peuvent contenir des lettres, des chiffres ou des underscores et être aussi longs qu'on veut. Une différence est faite entre minuscule et majuscule: `$A` et `$a` ne font pas référence à la même variable.

2.1.5. L'opérateur chop()

Cet opérateur prend comme unique paramètre le nom d'une variable, retire le dernier caractère de la chaîne à laquelle cette variable est égale ou équivalente et retourne le caractère éliminé. Exemple:

```
$var = "bonjour a tous";
$lettre = chop($var);
# $var contient maintenant "bonjour a tou"; $lettre contient "s"
```

2.1.6. La variable scalaire <STDIN>

Lorsque `<STDIN>` est utilisée là où une valeur scalaire est attendue, Perl lit la ligne courante de l'entrée standard (jusqu'au prochain *newline*) et utilise cette chaîne pour donner une valeur à `<STDIN>`. S'il n'y a rien à lire, le programme Perl s'arrête et attend qu'une valeur soit saisie. La valeur chaîne de `<STDIN>` se termine donc par un `\n`. L'opérateur `chop()` permet de l'éliminer. Un exemple d'utilisation:

```
print "Donnez un nombre :";
$a = <STDIN>;
chop($a);
print "Votre nombre : $a";
```

2.1.7. L'opérateur de sortie print()

L'opérateur `print` permet d'obtenir sur la sortie standard (par défaut) la valeur scalaire du paramètre qu'on lui passe; exemple:

```
print "coucou";   # coucou s'affiche à l'écran
```

2.1.8. La valeur undef

Si une variable est utilisée avant d'avoir été initialisée, ce n'est pas grave; les variables non initialisées ont pour valeur *undef*, qui est un 0 quand on parle de nombres ou la chaîne vide quand on parle de chaînes de caractères. De nombreux opérateurs retournent la valeur *undef* lorsque les paramètres qu'on leur a passé n'ont pas de sens.
`<STDIN>` retourne normalement la ligne courante de l'entrée; s'il n'y a rien à lire, il retourne *undef*.

2.2. LES LISTES ET LES TABLEAUX

Un tableau est une liste ordonnée de scalaires. Le nombre d'éléments d'un tableau est illimité. Le plus petit tableau n'a pas d'élément; le plus grand remplirait toute la mémoire disponible.

2.2.1. Représentation littérale

Un tableau est donc un ensemble de scalaires séparés par des virgules et encadré par des parenthèses. Ces éléments ne sont pas forcément des constantes: ils peuvent être des expressions qui seront évaluées à chaque utilisation. Exemples de tableaux:

```
( 1, 2, 3 )
("coucou", 4.5 )
()                # tableau vide
```

Il existe également un opérateur de construction de liste: **range**, noté **..** . Il permet de créer une liste dont les valeurs sont comprises entre l'opérande gauche et l'opérande droit et sont obtenues à partir de l'opérande gauche par incrémentations successives de pas 1; exemples:

```
( 1..3 )          # équivaut à la liste (1, 2, 3 )
( 3.3 .. 6.1 )    # équivaut à la liste ( 3.3, 4.3, 5.3 )
```

2.2.2. Les variables tableaux

Un nom de tableau vérifie les mêmes propriétés qu'un nom de scalaire, si ce n'est qu'il commence par le caractère **@** . La valeur d'un tableau qui n'a pas encore été initialisé est la liste vide (). Exemples:

```
@tableau
@Tableau          # différent de @tableau
@un_tableau
```

2.2.3. L'affectation

Elle permet de donner une valeur à une variable tableau; exemples:

```
@tab = ( 1, 2, 3 );
@chiffres = 6;
```

Dans ce deuxième exemple, le scalaire 6 devient l'unique élément du tableau `@chiffres`.
Le nom d'un tableau peut apparaître à l'intérieur d'une liste. Perl le remplace alors automatiquement par ses éléments; exemple:

```
@chiffres = ( @tab, 6 );     # équivaut à @chiffres = (1,2,3,6)
```

Si un tableau contient uniquement des références à des variables, il peut être traité comme une variable. En particulier, il peut constituer la partie gauche d'une affectation; exemples:

```
( $a, $b, $c ) = ( 1, 2, 3 );     # $a contient 1, $b 2, $c 3
( $d, @tab ) = ( $a, $b, $c );    # $d a la même valeur que $a,
                                  @tab contient( $b, $c )
```

Si le nombre d'éléments affectés ne correspond pas au nombre de variables, les valeurs en trop (à droite de l'affectation) sont éliminées et les variables non affectées (à gauche de l'affectation) reçoivent la valeur *undef*.

Si une variable scalaire est affectée par un tableau, elle reçoit en fait le nombre d'éléments de ce tableau; exemples:

```
@tab = ( 1, 2, 3 );
$a = @tab;        # $a contient la valeur 3
($a) = @tab;      # $a contient le premier élément de @tab : 1
```

Le troisième exemple est une affectation entre tableaux, alors que le deuxième est une affectation entre scalaires. La valeur d'une affectation entre tableaux est un tableau, ce qui permet d'affecter des tableaux en cascade; exemples:

```
@tab = ( @chiffres = ( 1, 2, 3 ) );
# @tab et @chiffres contiennent tous les deux ( 1, 2, 3 )
@tab = @chiffres = ( 1, 2, 3 );        # même chose
```

2.2.4. L'accès aux éléments

Les éléments d'un tableau sont repérés par des entiers; le premier élément est indexé par 0. Ils sont accédés par des variables scalaires ou des tableaux; exemples:

```
@tab = ( 1, 2, 3 );
$a = $tab[0];          # $a contient le premier élément de @tab,
                       soit 1
$b = $tab[ $a ];       # $b contient le deuxième élément de
                       @tab, soit 2
@tab2 = @tab[0, 2];    # @tab2 contient le premier et le dernier
                       élément de @tab, soit( 1, 3 )
```

Dans le dernier exemple, on accède à une liste d'éléments du même tableau et on crée donc un tableau, d'où le signe @.

Si on tente d'accéder à un élément dont l'indice est supérieur au nombre d'éléments du tableau, la valeur *undef* est retournée; exemple:

```
@tab = ( 1, 2, 3 );
$val = $tab[7];        # $val contient undef
```

Si on affecte un élément dont l'indice est supérieur au nombre d'éléments du tableau, celui-ci est étendu automatiquement, et les valeurs non affectées reçoivent la valeur *undef*; exemple:

```
@tab = ( 1, 2, 3 );
$tab[5] = 6;
# @tab contient maintenant ( 1, 2, 3, undef, undef, 6 )
```

Enfin, la variable `$#tab` permet d'accéder à l'indice du dernier élément du tableau `@tab`.

2.2.5. Les opérateurs push() et pop()

push permet d'ajouter un élément ou une liste d'éléments à la fin d'un tableau.
pop permet d'éliminer le dernier élément d'un tableau; exemples:

```
push ( @tab, $a );          # @tab = (@tab, $a)
push ( @tab, 4, 5, 6 );     # @tab = (@tab, 4, 5, 6)
$b = pop ( @tab );          # @tab = (1, 2, 3, 4, 5) et $b = 6
```

`pop()` renvoie la valeur *undef* si la liste qu'on lui a passée en paramètre est vide.

2.2.6. Les opérateurs shift() et unshift()

shift permet d'éliminer le premier élément d'un tableau. **unshift** permet d'ajouter un élément ou une liste d'éléments en début de tableau; exemples:

```
@tab = ( 1, 2, 3 );
unshift ( @tab, $a );        # @tab = ( $a, @tab )
unshift ( @tab, 4, 5, $c ); # @tab = ( 4, 5, $c, @tab )
$b = shift ( @tab );         # @tab = (5,$c,1,2,3) et $b = 4
```

`shift()` renvoie la valeur *undef* si la liste qu'on lui a passée en paramètre est vide.

2.2.7. L'opérateur reverse()

Cet opérateur permet d'inverser l'ordre des éléments d'un tableau. Il retourne une liste; exemples:

```
@tab = ( 1, 2, 3 );
@inv_tab = reverse ( @tab );           # @inv_tab = ( 3, 2, 1 )
@b = reverse ( @b );         # inverse @b et le range dans @b
```

2.2.8. L'opérateur sort()

Cet opérateur permet de classer dans l'ordre croissant des caractères ASCII les éléments du tableau qu'on lui passe en argument; exemples:

```
@x = sort ("petit", "moyen", "grand" );
# @x=("grand","moyen","petit")
@y = ( 1, 2, 4, 8, 16, 32, 64 );
@y = sort (@y);              # @y = ( 1, 16, 2, 32, 4, 64, 8 )
```

2.2.9. L'opérateur chop()

Il fonctionne comme sur les variables scalaires: chaque élément du tableau qu'on lui passe en paramètre est amputé de son dernier caractère; exemple:

```
@machin = ("Coucou \n", "Ca va ? \n" );
chop ( @machin );      # @machin = ("Coucou", "Ca va ?" )
```

2.2.10. Le tableau <STDIN>

Dans un contexte de tableau, cet opérateur renvoie une valeur différente que dans un contexte scalaire. En effet, il retourne une liste dont les éléments sont toutes les lignes restantes de l'entrée standard à partir de la ligne courante, prises séparément. Les *newline* sont pris en compte.

2.2.11. Interpolation de tableaux

Les tableaux et leurs éléments peuvent eux aussi être interpolés à l'intérieur des doubles quotes; exemples:

```
@speinf = ("seb", "cricri", "hugo");
$phrase1 = "$speinf[0] joue bien au baby."
# $phrase1 = "seb joue bien au baby"
$phrase11 = "$speinf[0, 1] jouent bien au baby."
# $phrase1 = "seb cricri jouent bien au baby"
$phrase2 = "@speinf s'occupent du 486."
# $phrase2 = "seb cricri hugo s'occupent du 386"

$speinf = "coucou";
$phrase3 = "voici $speinf[1]";        # $phrase3 = "voici cricri"
$phrase4 = "voici $speinf \[1]";      # $phrase4 = "voici coucou [1]"
$phrase5 = "voici $speinf" . "[1]";   # $phrase5 = "voici coucou [1]"
$phrase6 = "voici ${speinf }[1]";     # $phrase6 = "voici coucou [1]"
```

Les exemples des phrases 11 et 2 montrent que les éléments d'une liste sont interpolés dans l'ordre et séparés par un espace. Les exemples des phrases 4, 5 et 6 montrent comment faire suivre une variable scalaire d'un crochet ouvrant, sans que Perl interprète cela comme l'interpolation d'un élément de tableau.

2.3. STRUCTURES DE CONTROLE

2.3.1. Les blocs d'instructions

Un bloc d'instructions est de la forme:

```
{  instruction 1;
    instruction 2;
 ...
    dernière instruction;
}
```

C'est donc une séquence d'instructions séparées par des points virgules, et encadrée par des accolades. Perl exécute les instructions dans l'ordre.

2.3.2. L'instruction if / unless

Une instruction `if` est de la forme:

```
if (expression)
{  instructions à exécuter si l'expression est vraie;
}
else
{ instructions à exécuter si l'expression est fausse;
}
```

Les accolades sont obligatoires!
L'expression est évaluée en tant que **chaîne** à l'exécution. Si elle est égale à la chaîne vide (de longueur 0) ou au caractère "0", elle est évaluée à *faux*. Sinon elle est évaluée à *vrai*.
Le bloc *else* est optionnel.

Un exemple d'instruction if:

```
print "Donnez votre age : \n";
$age = <STDIN>;
chop ( $age );
if ($age >= 18)
{ print "Vous pouvez aller voter! "; }
else
{ print "Vous êtes trop jeune pour voter! "; }
```

`if` peut être remplacé par `unless`. Cela revient à dire: « Si l'expression est fausse, alors... ». `unless` peut avoir lui aussi un bloc `else`.
S'il y a plus de deux choix possibles, on peut ajouter des blocs **`elsif`** à une instruction `if`; exemple:

```
if (expression1)
{ bloc1; }
elsif (expression2)
{ bloc2; }
elsif (expression3)
{ bloc3; }
else
{ bloc4; }
```

Le nombre de blocs `elsif` n'est pas limité. Perl évalue les expressions les unes après les autres; lorsqu'il en rencontre une ayant pour valeur *vrai*, le bloc d'instructions correspondant est exécuté et les autres sont ignorés. Si toutes les expressions sont fausses, le `else` est exécuté (s'il y en a un).

2.3.3. L'instruction while / until

L'instruction `while` permet de répéter l'exécution d'un bloc d'instructions tant que l'expression de contrôle est vraie; cette instruction est de la forme:

```
while (expression)
{ instruction_1;
   instruction_2;
  ...
   instruction_n;
}
```

Pour exécuter le `while`, Perl évalue l'expression. Si elle est vraie, le bloc d'instructions est exécuté. L'expression est alors réévaluée. Ceci se répète jusqu'à ce que l'expression soit fausse; le bloc d'instructions est alors ignoré.

`while` peut être remplacé par `until`; cela revient à dire: « Tant que cette expression est fausse, exécuter ...»; exemple:

```
until (expression)
{ bloc d'instructions; }
```

2.3.4. L'instruction for

Elle est très semblable à celle dont on dispose en C; elle est en effet de la forme:

```
for (initial_exp; test_exp; increment_exp)
{ instruction_1;
   instruction_2;
 ...
   instruction_n;
}
```

Cela se traduit avec les instructions étudiées jusqu'ici par:

```
initial_exp;
while ( test_exp )
{ instruction_1;
   instruction_2;
 ...
   instruction_n;
   increment_exp;
}
```

L'*initial_exp* est évaluée en premier; en général elle permet d'initialiser un compteur de boucle (mais ce n'est pas obligatoire). Ensuite la *test_exp* est évaluée; si elle est vraie, le corps de boucle et l'*increment_exp* sont exécutés. Perl réévalue alors *test_exp* , tant que cela est nécessaire.

L'exemple suivant permet de faire afficher les nombres de 1 à 10, séparés par un espace:

```
for ( $i = 1; $i <= 10 ; $i ++ )
{ print "$i"; }
```

2.3.5. L'instruction foreach

Cette instruction affecte une variable scalaire avec les éléments d'une liste pris les uns après les autres et exécute le bloc d'instructions; elle est de la forme:

```
foreach $i ( @liste )
{ instruction_1;
 ...
   instruction_n;
}
```

La variable scalaire est **locale** à la boucle; c'est-à-dire qu'elle reprend la valeur qu'elle avait avant l'exécution de la boucle lorsqu'on sort de celle-ci.
On peut omettre le nom de la variable scalaire; dans ce cas `$_` est utilisée par défaut.

Si la liste qu'on utilise est en fait simplement un tableau, alors la variable scalaire devient une référence à chaque élément de ce tableau, plutôt que d'être une copie de chacun d'eux. Cela signifie que si la variable scalaire est modifiée, alors l'élément du tableau qu'elle représente alors est aussi modifié; un exemple pour y voir plus clair:

```
@val = ( 2, 5, 9 );
foreach $chiffre (@val)
{ $chiffre *= 3;}
# on a maintenant @val = ( 6, 15, 27 )
```

2.4. LES TABLEAUX ASSOCIATIFS

2.4.1. Les variables tableaux associatifs

Leur définition est donnée au paragraphe 1.5.
Leur nom commence par le symbole **%**. Il suit ensuite les mêmes règles que les noms de variables scalaires ou de tableaux.

En général, on ne fait pas référence à un tableau associatif dans son entier, mais à ses éléments. Chaque élément est accédé par sa clé; ainsi, les éléments du tableau associatif `%tab` sont accédés par `$tab{$cle}`, où `$cle` est une expression scalaire. Vouloir accéder à un élément du tableau qui n'existe pas a pour résultat *undef*.

De nouveaux éléments sont créés par affectation et peuvent être manipulés comme des variables scalaires; exemples:

```
$tab{"coucou"} = "ca va? ";
# création de la clé "coucou", à laquelle est associée la valeur
"ca va? "
$tab{123.5} = 4568;
# création de la clé 123.5, à laquelle est associée la valeur
4568
print "$tab{"coucou"}";          # affichage de "ca va? "
$tab{123.5} +=3;
# la valeur associée à la clé 123.5 est maintenant 4571
```

La représentation littérale d'un tableau associatif est une liste de paires de clés et des valeurs qui leur sont associées. L'ordre des paires dans la liste ne peut pas être contrôlé; il dépend de la fonction de hachage utilisée par Perl pour accéder rapidement aux éléments du tableau; exemples:

```
%autre_tab = %tab;
# %autre_tab a les mêmes éléments que %tab
@liste = %tab;
# @liste = ("coucou", "ca va? ", 123.5, 4571)
```

2.4.2. Les opérateurs sur les tableaux associatifs

- L'opérateur **keys()**

  Cet opérateur retourne la liste des clés du tableau associatif qu'on lui passe en paramètre. L'ordre des éléments de cette liste dépend de la fonction de hachage utilisée par Perl pour accéder rapidement aux éléments du tableau.
  Les parenthèses sont optionnelles.
  Exemples:

```
$tab{"coucou"} = "ca va? ";
$tab{123.5} = 4568;
@liste = keys(%tab);
# @liste = ("coucou ", 123.5) ou bien (123.5, "coucou")
@liste = keys %tab;       # idem
```

  Les éléments des tableaux associatifs sont interpolés à l'intérieur des doubles quotes; exemple:

```
foreach $key (keys %tab)
{ print "la valeur associée à la clé $key est $tab{$key}."; }
```

  Dans un contexte scalaire, `keys()` retourne le nombre de paires (clé-valeur) du tableau passé en paramètre; exemple:

```
while ($i < keys %tab)
{ ... }
```

- L'opérateur **values()**

  Cet opérateur retourne la liste des valeurs du tableau associatif qu'on lui passe en paramètre, dans le même ordre que les clés retournées par `keys()`. Les parenthèses sont optionnelles.
  Exemple:

```
@liste_valeurs = values(%tab);
# @liste_valeurs contient("ca va?",4568)ou bien(4568,"ca va?")
```

- L'opérateur **each()**

  Pour examiner tous les éléments d'un tableau associatif, on peut donc faire appel à `keys()` puis récupérer les valeurs correspondant aux clés. En utilisant `each()`, on obtient directement une paire (clé-valeur) du tableau passé en paramètre.

A chaque évaluation de cet opérateur pour un même tableau, la paire suivante est retournée, jusqu'à ce qu'il n'y ait plus de paire à accéder; `each()` retourne alors la chaîne vide. L'exemple précédent peut alors s'écrire:

```
while ( ($cle, $valeur) = each(%tab) )
{ print "la valeur associée à la clé $cle est $valeur . "; }
```

- L'opérateur **delete**

Il permet d'éliminer la paire (clé-valeur) du tableau associatif dont on a passé la clé en paramètre; exemple:

```
%tab = ("coucou ", "ca va? ", 123.5, 4571);
@element = delete $tab{"coucou"};
# @element = ("coucou", "ca va?") et %tab = (123.5, 4571)
```

2.5. LES ENTREES/SORTIES

2.5.1. Les entrées avec STDIN

Lire l'entrée standard avec le manipulateur de fichier `STDIN` est très simple. Nous l'avons déjà fait avec l'opérateur `<STDIN>` qui peut être évalué:

- dans un contexte scalaire, à la ligne courante de l'entrée standard ou à *undef* si il n'y a plus de ligne;
- dans un contexte de tableau, au reste des lignes contenues dans l'entrée standard, sous forme de liste, chaque ligne étant un élément de la liste.

Pour lire les lignes contenues dans l'entrée standard et les manipuler séparément, on utilise `$_` comme suit:

```
while ($_ = <STDIN>)
{ # code }
```

Tant qu'il y a quelque chose à lire `<STDIN>` est évalué à *vrai* et la boucle s'exécute. Quand il n'y a plus rien à lire, `<STDIN>` retourne *undef*, qui est évalué à *faux*, ce qui termine la boucle. Quand un test de boucle consiste uniquement en un opérateur d'entrée (comme < ... >), Perl copie automatiquement la ligne lue dans `$_`; donc, l'exemple précédent peut s'écrire:

```
while (<STDIN>)
{ # code }
```

2.5.2. Les entrées avec l'opérateur diamant <>

Cet opérateur se comporte comme `<STDIN>`:

- dans un contexte scalaire, il retourne une ligne ou *undef*;

dans un contexte de tableau, il retourne les lignes restantes.

Cependant, il lit les données dans les fichiers spécifiés dans la ligne de commande faisant appel au programme Perl.
Par exemple, le script suivant:

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

a le même résultat que la commande *cat*; s'il est stocké dans le fichier *pcat* et qu'on fait appel à lui avec la ligne de commande:

```
> pcat fic1 fic2 fic3
```

les contenus des trois fichiers seront affichés à la suite les uns des autres.

En fait, <> fonctionne grâce au tableau `@ARGV` dont nous avons déjà fait la connaissance au chapitre 1. Pour ceux qui n'ont pas suivi, ce tableau est la liste des arguments de la ligne de commande.

2.5.3. La sortie avec STDOUT

Perl utilise `print` et `printf` pour écrire sur la sortie standard.

- **print**

  Cet opérateur prend une liste de chaînes et les envoie les unes après les autres sur la sortie standard. `print` retourne une valeur, comme tout opérateur sur les listes: *vrai* si la sortie a été effectuée avec succès, *faux* sinon. On a donc le droit d'écrire:

```
$a = print ("hello ", "word", "\n");
# affichage de "hello word" avec retour à la ligne et $a = 1
```

  On est parfois obligé de mettre des parenthèses, en particulier si la première chose à afficher commence par une parenthèse ouvrante; exemples:

```
print (2+3),"coucou";
# faux; n'affiche que 5, ignore "coucou"
print ( (2+3), "coucou");      # affichage de 5coucou
print 2+3, "coucou";           # idem
```

- **printf**

  Cet opérateur prend une liste de paramètres (encadrés ou non par des parenthèses). Le premier paramètre est une chaîne de contrôle des formats, indiquant comment afficher les paramètres restant. C'est comme le *printf* du C; exemple:

```
printf "%15s %5d %10.2f \n", $s, $n, $r ;
```

2.6. LES EXPRESSIONS REGULIERES

Nous en avons parlé rapidement au paragraphe 1.4. Nous allons maintenant les étudier plus en détail.
Une expression régulière est un pattern qui est recherché dans une chaîne de caractères. Le résultat de la recherche peut être positif ou négatif. Parfois, c'est simplement ce résultat qui nous intéresse; parfois, on souhaite pouvoir remplacer le pattern par une autre chaîne.
Les expressions régulières sont utilisées par de nombreux programmes Unix comme *grep*, *sed*, *awk*, *ed*, *vi*, *emacs* et les différents shells. N'importe quelle expression régulière qui peut être décrite avec l'un des outils Unix peut également être écrite en Perl, sans nécessairement utiliser les mêmes caractères.

2.6.1. Utilisations simples des expressions régulières

Si l'on désire récupérer les lignes du fichier *nomfic* comportant la chaîne `abc`, on utilise la commande:

```
> grep abc nomfic
```

Ici, `abc` est l'expression régulière utilisée par `grep` pour tester toutes les lignes de *nomfic*. Les lignes matchées sont envoyées sur la sortie standard.

En Perl, `abc` est considérée comme une expression régulière si elle est encadrée par des slashes; le script Perl équivalent à la commande `grep` précédente est donc:

```
while (<>)
{if (/abc/)
   { print; }  }
```

Le matching est effectué sur la variable `$_`, qui contient la ligne courante du fichier passé en paramètre au script. La sortie se fait sur la sortie standard.

Si maintenant on veut trouver les lignes de *nomfic* contenant un `a` suivi de zéro ou plusieurs `b` et d'un `c`, on peut utiliser la commande:

```
> grep "ab*c" nomfic
```

On est obligé d'utiliser des doubles quotes pour la chaîne contenant l'astérisque, si l'on ne veut pas que le shell l'interprète comme un nom de fichier.

En Perl, cela est réalisé de la façon suivante:

```
while (<>)
{if (/ab*c/)
   { print; }  }
```

Un autre opérateur sur les expressions régulières est l'opérateur de substitution `s///`, qui remplace la partie de la chaîne matchée par l'expression régulière par une autre chaîne. Exemple:

```
s/ab*c/def/;
```

La chaîne (ici `$_`) est parcourue. On y recherche un `a` suivi de zéro ou plusieurs `b` et d'un `c`. Si la recherche est fructueuse, la partie de `$_` matchée est remplacée par la chaîne `def`. Sinon, il ne se passe rien.

2.6.2. Les patterns

Il y a différentes sortes de patterns:

- Les patterns à un seul caractère

  Le plus simple est un unique caractère qui se matche lui-même. Ainsi, `/a/` permet de rechercher la lettre `a` dans une chaîne.

  Le point, "**.**", permet de matcher n'importe quel caractère sauf *newline* (\n). Par exemple, `/a./` permet de matcher toute séquence de deux caractères commençant par `a` et n'étant pas "a\n".

  Une classe de caractères pour le matching est représentée par deux crochets encadrant une liste de caractères. Un seul de ces caractères doit être présent dans la ligne parcourue pour que le matching fonctionne. Exemple:

  ```
  /[abcde]/
  ```

  matche l'un des cinq caractères, en minuscule.
  Si l'on veut inclure le crochet fermant dans la liste des caractères à matcher, il doit être précédé par un antislash ou se trouver en début de liste. On peut figurer un intervalle de valeurs en utilisant le tiret (-). Si celui-ci doit figurer dans la liste de caractères à matcher, il suffit de le faire précéder d'un antislash.

Exemple:

```
/[a-e]/         # même chose que l'exemple précédent
```

Il existe un caractère de négation: **^**; placé juste après le crochet ouvrant, il permet de nier une classe de caractères: tout caractère ne se trouvant pas dans la liste sera matché. Exemple:

```
/[^0-9]/  # matche tout ce qui n'est pas un chiffre
```

Si on veut faire figurer ^ dans la liste des caractères à matcher, il suffit de le faire précéder d'un antislash.

Certaines classes sont prédéfinies:

Construction	Classe équivalente	Construction négative	Classe équivalente
/d (digits)	[0-9]	/D (digits, not!)	[^0-9]
/w (words)	[a-zA-Z0-9]	/W (words, not!)	[^a-zA-Z0-9]
/s (space)	[\r \n \f \t]	/S (space, not!)	[^ \r \n \f \t]

- Les patterns groupants

1. Les séquences

   Comme nous l'avons vu plus haut, `abc` matche un `a` suivi d'un `b` suivi d'un `c`. On appelle cela une séquence.

2. Les multiplicateurs

   Nous avons déjà vu l'astérisque (*****); elle signifie « 0 ou plusieurs » pour le caractère immédiatement précédent.
   On dispose également des caractères **+** et **?**, signifiant respectivement « 1 ou plusieurs » et « 0 ou 1 » pour le caractère immédiatement précédent. Exemple:

   ```
   /fo+ba?r/
   ```

   matche un `f` suivi d'un ou plusieurs `o`, suivi d'un `b`, suivi de zéro ou un `a`, suivi d'un `r`.
   Ces trois patterns groupants sont dits *gloutons*; en effet, s'ils peuvent matcher plusieurs fois le même caractère, ils en matcheront toujours le maximum. Exemple:

   ```
   $_ = "Salut XXXXXXXXXX ! ";
   s/X*/pascal/;        # on obtient "Salut pascal ! ";
   ```

Si on a besoin d'être plus précis quant au nombre de répétitions du même caractère à matcher, il faut utiliser le **multiplicateur général**. Il se présente sous la forme de deux accolades renfermant un ou deux nombres séparés par une virgule.
Exemple:

```
/X{5,10}/
```

Comme pour les trois multiplicateurs déjà rencontrés, le caractère immédiatement précédent (ici, "X") doit être trouvé avec un nombre de répétitions compris entre les deux nombres contenus dans les accolades (ici, entre 5 et 10). On peut aussi avoir les patterns:

```
/X{5,}/   # matche au moins 5 répétitions de X
/X{5}/    # matche exactement 5 répétitions de X
/X{0,5}/  # matche au plus 5 répétitions de X
/a.{5}b/  # matche un a suivi de 5 caractères différents de
          newline, suivis d'un b
```

S'il y a deux multiplicateurs dans une même expression, c'est le plus à gauche qui est le plus glouton; Exemple:

```
$_ = "a xxx c xxxxxxx c xxx d";
/a.*c.*d/
```

Le premier ".*" matche tous les caractères jusqu'au deuxième `c`. Ici, ca ne fait pas grande différence, mais nous verrons tout à l'heure que ça peut en avoir.

3. Les parenthèses pour la mémorisation

Un autre pattern groupant est la paire de parenthèses encadrant un pattern. Elle permet de mémoriser la chaîne matchée par le pattern qu'elle renferme et de la référencer par la suite. Donc `(a)` matche toujours un `a`, `([a-z])` matche toujours n'importe quelle lettre minuscule.
Pour référencer la partie de la chaîne qui a été mémorisée, on utilise un antislash suivi d'un entier qui indique de quelle paire de parenthèses il s'agit (on compte à partir de 1).
Exemple:

```
/fred(.)barney\1/;
```

matche une chaîne comportant les caractères `fred`, suivis d'un caractère qui n'est pas un *newline*, suivi de `barney`, suivi du même caractère que précédemment. Donc la chaîne `fredxbarneyx` est matchée; par contre, la chaîne `fredxbarneyy` ne l'est pas.
Autre exemple:

```
/a(.)b(.)c\2d\1/;
```

matche un `a`, suivi d'un caractère (noté #1), suivi d'un `b`, suivi d'un caractère (noté #2), suivi d'un `c`, suivi du caractère #2, suivi d'un `d`, suivi du caractère #1.
On peut enfermer plus d'un caractère dans les parenthèses; exemple:

```
/a(.*)b\1c/;
```

matche un `a`, suivi une séquence de caractères, suivie d'un `b`, suivi de la même séquence de caractères, suivie d'un `c`.

Avec l'opérateur de substitution, la structure avec les \1, \2, etc permet d'utiliser les morceaux de chaîne mémorisés pour la construction de la chaîne de remplacement; exemple:

```
$_ = "a xxx b yyy c zzz d";
s/b(.*)c/d\1e/;                 # $_ = "a xxx d yyy e zzz d"
```

4. Les alternatives

Une construction alternative, telle que `a|b|c`, est une autre forme de pattern groupant. Elle signifie qu'il faut matcher l'une des alternatives qu'elle comporte (`a` ou `b` ou `c`). Les alternatives peuvent être constituées de plusieurs caractères; exemple:

```
/coucou|salut/
```

Remarquez que `/a|b|c/` est équivalent à `/[a-c]/`.

- Les patterns d'ancrage

Il y en a quatre. Ils permettent préciser à quel endroit de la chaîne la recherche du pattern doit commencer.
La première paire d'ancres vérifie qu'une certaine partie de la chaîne matchée se trouve ou non à l'une des extrémités d'un mot. Une extrémité de mot est l'endroit entre des caractères matchés par \w et \W, ou entre des caractères matchés par \w et le début ou la fin de la chaîne. **\b** exige une extrémité de mot à l'endroit où il est placé pour que le matching s'effectue. **\B** exige le contraire. Exemples:

```
/fred\b/        # matche fred, Alfred, mais pas frederic
/\bwiz/         # matche wizard mais pas qwiz
/\bfred\b/      # matche uniquement fred
/\b+\b/         # matche x+y, mais pas ++ ou +
/\bfred\B/      # matche frederic, mais pas fred flintstone
```

La deuxième paire d'ancres vérifie qu'une certaine partie de la chaîne matchée est au début ou à la fin de la chaîne. ^ exige que le pattern qui le suit soit en début de chaîne. $ exige que le pattern qui le précède soit en fin de chaîne.

Exemples:

```
/^a/      # matche les chaînes commençant par a
/a^/      # matche les chaînes comportant un a, suivi de ^
/\^/      # matche les chaînes contenant le caractère ^
/c$/      # matche les chaînes dont le dernier caractère est
          un c
/\$/      # matche les chaînes contenant le caractère $
```

- Priorité

Comme pour les opérateurs, il existe des règles de priorité pour les patterns groupants et d'ancrage. Elles sont explicitées dans le tableau suivant, de la priorité la plus grande à la plus petite:

Nom	Représentation
Parenthèses	( )
Les multiplicateurs	* + ? {m,n}
Séquences et ancrages	abc ^ $ \b \B
Alternatives	|

Par exemple, `/a|b*/` matche les chaînes contenant un `a` ou plusieurs `b`. Par contre `/(a|b)*/` matche les chaînes contenant plusieurs `a` ou plusieurs `b`.
Quand on utilise les parenthèses pour la priorité, elles déclenchent également la mémorisation. Attention donc lors de l'utilisation des \1, \2, etc.

2.6.3. Compléments sur l'opérateur de matching

Nous avons déjà vu l'utilisation la plus simple de cet opérateur: une expression régulière entre deux slashes. Nous allons à présent faire la connaissance des autres possibilités qu'offre cet opérateur.

- Les opérateurs =~ et !~

Ces opérateurs sont utilisés pour effectuer la recherche d'un pattern dans une autre chaîne que celle contenue dans `$_`. Exemple:

```
$a = "bonjour tout le monde! ";
$a =~ /^bon/;             # $a contient vrai
$chaine !~ /pattern/;
# équivaut à !($chaine=~/pattern/);
```

La cible des opérateurs =~ et !~ peut être n'importe quelle expression dont la valeur est une chaîne. On peut donc les utiliser avec <STDIN>; exemple:

```
print "Bonjour, professeur Falken! Vous voulez bien jouer avec
moi? "
if (<STDIN> =~ /^[oO]/)
{ print "Je vous propose une guerre thermonucléaire globale."
}
```

- L'option **i**

Cette option permet d'ignorer la différence entre minuscules et majuscules. On la place après le deuxième slash. Les lettres du pattern matchent alors les lettres de la chaîne, qu'elles soient minuscules ou majuscules. L'exemple précédent devient:

```
print "Bonjour, professeur Falken! Vous voulez bien jouer avec
moi? "
if ( <STDIN> =~ /^o/i )
{ print "Je vous propose une guerre thermonucléaire globale."
}
```

- Utilisation d'un autre délimiteur

Lorsqu'on doit inclure un slash dans le pattern, celui-ci doit être précédé par un antislash. Exemple:

```
$path =~ /^\/usr\/etc/;
```

Ce n'est donc pas marrant s'il y a plusieurs slashes dans le pattern. Perl permet donc, ici encore, de se simplifier la vie en autorisant tout autre caractère non alphanumérique à servir de délimiteur, à condition qu'il soit précédé d'un **m** (matching!). Exemple:

```
$path =~ m#^/usr/etc#;    # # est utilisé comme délimiteur
$path =~ m@^/usr/etc@;    # @ est utilisé comme délimiteur
$path =~ m/^\/usr\/etc/;
# / est utilisé comme délimiteur; le m est alors optionnel
```

- L'interpolation de variable

L'interpolation de variable est effectuée avant l'évaluation de l'expression régulière. Exemple:

```
$assertion = "Un chasseur sachant chasser";
$mot = "chasseur";
if ($assertion =~ /$mot/ )
{ print "L'assertion contient le mot $mot. "; }
```

- Des variables spéciales

  Après une reconnaissance de pattern réussie, les variables $1, $2, etc prennent les mêmes valeurs que \1, \2, etc. Exemple:

```
$_ = "ceci est un test";
/(\w+)\W+(\w+)/;     # matche les deux premiers mots de $_
# $1 = "ceci", $2 = "est"
```

  On peut aussi récupérer ces valeurs en effectuant le matching dans un contexte de tableau. L'exemple précédent peut s'écrire:

```
$_ = "ceci est un test";
($premier, $deuxième) = /(\w+)\W+(\w+)/;
# $premier = "ceci", $deuxième = "est"
```

  $1 et $2 demeurent inchangées: $1 = "ceci" et $2 = "est".

  Il existe trois autres variables, accessibles en lecture uniquement, comme $1, $2, etc:

  1. $& contient le bout de chaîne matché
  2. $´ contient le bout de chaîne situé après la partie matchée
  3. $` contient le bout de chaîne situé avant la partie matchée

  Exemple:

```
$_ = "ceci est un test simple";
/te.*t/;        # matche test
#$& = "test", $´ = "simple", $` = "ceci est un"
```

2.6.4. Les substitutions

Nous avons déjà vu la plus simple utilisation de cet opérateur; complétons maintenant nos connaissances.
Si l'on souhaite que la substitution soit effectuée à chaque occurrence du pattern, il suffit d'ajouter un **g** après le dernier slash; exemple:

```
$_ = "coucou! ca va? ";
s/ou/ri/g;        # $_ = "cricri! ca va? "
```

L'interpolation de variable a lieu dans la chaîne de remplacement; exemple:

```
$_ = "bonjour, pascal! ";
$autre = "seb";
s/pascal/$autre/;# $_ = "bonjour, seb! "
```

Les caractères de la chaîne de remplacement ne sont donc pas forcément fixés à l'avance et peuvent même être extraits des caractères matchés; exemple:

```
$_ = "ceci est un test";
s/(\w+)/<$1>/;          $_ = "<ceci> <est> <un> <test>"
```

Si on ajoute un **i** après le dernier slash, les majuscules ou les minuscules sont ignorées dans l'expression régulière (même chose qu'avec le matching).
On peut également utiliser des délimiteurs différents du slash; exemple:

```
s#fred#barney#;
# il suffit d'utiliser trois fois le même caractère
```

L'opérateur **=~** permet d'effectuer la substitution sur une autre chaîne que celle contenue par `$_`. Exemple:

```
$lequel = "ceci est un test";
$lequel =~ s/test/quizz/;   # $lequel = "ceci est un quizz"
```

2.6.5. Les opérateurs split() et join()

- L'opérateur *split()*

  Cet opérateur prend comme paramètres une expression régulière et une chaîne, et cherche toutes les occurrences de l'expression régulière dans la chaîne. Les parties de la chaîne qui n'ont pas été matchées sont retournées sous forme d'une liste. Exemple:

```
$ligne = "merlyn::118:10:Randal:/home/merlyn:/usr/bin/perl";
@champs = split(/:/, $ligne);
# décompose $ligne en prenant : comme délimiteur
# @champs = ("merlyn","","118","10","Randal","/home/merlyn",
 "/usr/bin/perl")
```

La chaîne par défaut pour cet opérateur est la variable `$_`; le pattern par défaut est `/\s+/`. Exemple:

```
@mots = split;       # équivalent à @mots = split(/\s+/, $_)
```

- L'opérateur *join()*

Cet opérateur prend une liste de valeurs et les recolle en plaçant zéro, un ou plusieurs caractères *collants* entre chaque élément de la liste; exemple:

```
$grosse_chaine = join($colle, @liste);
# $colle contient les caractères collants
```

Pour recoller la ligne de */etc/passwd* découpée précédemment, on peut utiliser l'instruction:

```
$resultat = join(":", @champs);
```

2.7. LES FONCTIONS

L'utilisateur peut définir ses propres fonctions en Perl. On parle alors plutôt de *subroutine* ou *sub*.

2.7.1. Définition d'une subroutine

Une subroutine dans un programme Perl est définie comme suit:

```
sub nom_subroutine
{ instruction_1;
   instruction_2;
 ...
   instruction_n;
}
```

Un nom de subroutine suit les mêmes règles que les noms de variables, de tableaux et de tableaux associatifs. Le bloc d'instructions suivant le nom de la subroutine devient sa définition. Lorsqu'on fait appel à la subroutine, c'est le bloc d'instructions qui est exécuté et tout résultat est renvoyé à l'utilisateur.
Les subroutines peuvent être définies n'importe où dans le programme. Les définitions des subroutines sont globales.

A l'intérieur d'une subroutine, on peut manipuler des variables partagées avec le reste du programme (globales donc!).

2.7.2. L'appel à une subroutine

Pour faire appel à une subroutine, n'importe où dans le programme, on fait précéder son nom par le symbole **&**; exemple:

```
for ( $i=&val-init; $i<&val_test; $i+=&val_increment)
{...   }
```

Une subroutine peut faire appel à d'autres subroutines faisant elles-mêmes appel à d'autres subroutines...

2.7.3. Résultats d'une fonction

Le résultat d'une subroutine est la valeur de la **dernière expression évaluée** dans le corps de la subroutine, à chaque appel. Une subroutine peut également retourner une liste, dans un contexte de tableau.
Exemples:

```
sub somme_a+b
{ $a + $b; }
$a = 3; $b = 5;
$c = &somme_a+b;       # $c = 8

sub choix_a_ou_b
{ if ($a > 0)
   {  print "je choisis a ($a) \n";
       $a;  }
   else
   { print "je choisis b ($b) \n";
       $b;  }
}
```

Dans ce deuxième exemple, la dernière expression évaluée est soit `$a` soit $b. Si on avait inversé les lignes avec `$a` et le `print`, on aurait 1 comme valeur de retour (`print` a marché) plutôt que `$a`.

2.7.4. Les paramètres

Si l'appel à une subroutine est suivi d'une liste parenthésée, la liste est automatiquement rangée dans le tableau **`@_`** pendant la durée d'utilisation de la subroutine. Celle-ci peut accéder ce tableau pour savoir combien il y a d'arguments et quelle est leur valeur. Exemple:

```
sub bonjour_ a_qui
{ print "bonjour, $_[0] ! \n"; }
```

`$_[0]` est le premier élément de la liste d'arguments. **Ne pas le confondre avec la variable $_!**

Un appel possible de cette subroutine:

```
&bonjour_a_qui ("cricri");
# affichage de "bonjour, cricri !"
```

Un exemple de subroutine utilisant plus d'un paramètre:

```
sub parler
{ print "$_[0], $_[1] ! \n"; }

&parler ("bonjour", "cricri");
# affichage de "bonjour, cricri !"
```

Les paramètres en trop sont ignorés. Les paramètres manquant se voient attribuer la valeur *undef*.
`@_` est local à la subroutine. S'il existait une valeur globale pour `@_`, elle est sauvegardée avant l'appel à la subroutine et restaurée lorsque l'exécution de la subroutine est achevée. Une subroutine peut donc passer des arguments à une autre subroutine sans pour autant perdre sa propre liste d'arguments.

Reprenons l'exemple de `somme_a+b` en le modifiant de façon à pouvoir effectuer la somme de deux nombres, quels qu'ils soient:

```
sub somme2
{ $_[0] + $_[1] ; }

$c = &somme2( 3, 5);          # $c = 8
```

Si maintenant on veut faire la somme de plusieurs nombres:

```
sub somme
{ $somme = 0 ;                # initialisation
   foreach ( @_ )
   { $somme += $_ ; }         # somme de chaque élément
   $somme ;
   # dernière expression évaluée: le résultat
}

$c = &somme ( 3, 4, 6, 7);         # $c = 20
```

2.7.5. Les variables locales

L'opérateur **local()** permet de créer des variables locales à une subroutine. Il reçoit une liste de noms de variables et en crée des instanciations. Si ces variables existaient déjà en tant que variables globales ou locales à d'autres fonctions, elles sont sauvegardées avant l'exécution de la subroutine qui les utilise et restaurées lorsque cette exécution est terminée. Reprenons l'exemple ci-dessus:

```
sub somme
{ local ($somme) ;          # fait de $somme une variable locale
   $somme = 0 ;             # initialisation
   foreach ( @_ )
   { $somme += $_ ; }       # somme de chaque élément
   $somme ;
   # dernière expression évaluée: le résultat
}
```

Le résultat de `local()` est une liste de variables pouvant être affectées. On peut donner des valeurs initiales à toutes ces nouvelles variables; sinon, elles sont initialisées avec *undef*. On peut donc modifier l'exemple précédent en remplaçant les deux premières lignes par:

```
local ($somme) = 0 ;
```

2.8. DIVERSES STRUCTURES DE CONTROLE

2.8.1. L'opérateur last

Cet opérateur permet de sortir plus tôt d'une boucle (c'est l'équivalent du *break* en C). L'exécution se poursuit avec l'instruction située immédiatement après la boucle. Exemple:

```
while (expression_1)
{ instruction_11;
   instruction_12;
 ...
   if (expression_2)
   { instruction_21;
      instruction_22;
      last;       # permet de sortir du while
    }
   instruction_1n;
 ...
}
# on reprend ici avec last
```

`last` n'a cet effet qu'avec les blocs de boucle `for`, `foreach`, `while` et les blocs indépendants (ne se rapportant pas à un `if` ou une subroutine ou un bloc plus important).

2.8.2. L'opérateur next

Comme `last`, `next` modifie l'ordre séquentiel d'exécution. Cet opérateur permet de sauter l'exécution d'un corps de boucle, sans terminer le bloc. Il est utilisé comme suit:

```
while (expression_1)
{ instruction_11;
   instruction_12;
 ...
   if (expression_2)
   { instruction_21;
      instruction_22;
      next;
```

```
     }
   instruction_1n;
 ...
   # on reprend ici avec next
}
```

2.8.3. L'opérateur redo

Cet opérateur permet de revenir au début du bloc d'itérations et donc de l'exécuter de nouveau, sans réévaluation de l'expression de contrôle; exemple:

```
while (expression_1)
{ # on revient ici avec redo
   instruction_11;
   instruction_12;
 ...
   if (expression_2)
   { instruction_21;
      instruction_22;
      redo;
    }
   instruction_1n;
 ...
}
```

2.8.4. Les blocs étiquetés

Pour sortir de plusieurs blocs à la fois, il suffit d'utiliser les opérateurs `next`, `redo` et `last` et d'étiqueter les blocs.
Une étiquette est un nom suivant les mêmes règles que les noms de variables scalaires, de tableaux ou de subroutines. Cependant ce nom n'est précédé par aucun symbole spécial; donc une étiquette telle que `print` est interdite, car elle serait confondue avec l'opérateur `print`. C'est pourquoi il est conseillé que le nom d'une étiquette soit composé uniquement de lettres majuscules et de chiffres.

L'étiquette est placée en tête du bloc à étiqueter; passée en paramètre à `next`, `last` ou `redo`, elle indique le bloc concerné par ces opérateurs.

Exemple:

```
EXTERIEUR:
for ( $i = 1; $i <= 10; $i++)
{ INTERIEUR:
   for ( $j = 1; $j <= 10; $j++)
   {  if ( $i * $j == 63 )
       {  print "$i fois $j égal 63 !\n";
          last EXTERIEUR;
       }
       if ( $j >= $i )
       {  next EXTERIEUR; }
    }
}
```

Cet exemple teste des valeurs successives de `$i` et `$j` jusqu'à ce que leur produit soit égal à 63. Lorsque la paire est trouvée, on s'arrête. On vérifie toujours que `$i` est plus grand que `$j`, sinon on passe à la valeur suivante de `$i`; on testera donc uniquement les paires (1, 1), (2, 1), (2, 2), (3, 1), ...

Si on avait utilisé `last` et `next` sans étiquette et même si on n'avait pas étiqueté le bloc le plus interne (`INTERIEUR`), ces opérateurs se seraient pourtant rapportés à ce bloc.
On ne peut pas utiliser les étiquettes pour entrer dans un bloc, mais seulement pour en sortir; les opérateurs `next`, `last` et `redo` doivent être dans ce bloc.

2.8.5. Les modificateurs d'expression

Une autre façon de dire: « Si ceci, alors cela », est: « cela si ceci ». Perl permet ce genre de raccourci, comme ceci:

```
une_expression if expression_de_contrôle;
```

L'expression de contrôle est évaluée en premier; si elle est vraie, alors *`une_expression`* est exécutée. Cela revient à:

```
if (expression_de_contrôle)
{ une_expression ; }
```

mais il y a moins de chose à taper. Cependant *`une_expression`* doit être une unique expression (et non un bloc d'instructions).

Un exemple d'utilisation: sortir d'une boucle quand une certaine condition est vérifiée:

```
LIGNE:
while (<STDIN>)
{last LIGNE if  /^Date/ ;}
```

De la même façon, il existe:

```
exp2 while exp1; # équivaut à while (exp1) { exp2;}
exp2 unless exp1;# équivaut à unless (exp1) { exp2;}
exp2 until exp1; # équivaut à until (exp1) { exp2;}
```

2.8.6. &&, || et ?: comme structures de contrôle

Une autre façon d'écrire: « si ceci, alors cela », est d'utiliser le **et-logique**:

```
ceci && cela ;
```

Comment ça peut marcher? Et bien:

- si `ceci` est vraie, alors l'expression ci-dessus n'est pas encore totalement évaluée puisqu'elle dépend de `cela`. Donc `cela` doit être évaluée.
- si `ceci` est fausse, alors l'expression ci-dessus est fausse aussi; pas besoin d'évaluer `cela`.

Donc `cela` est évaluée uniquement si `ceci` est vraie; c'est bien équivalent aux deux constructions que nous avons vues précédemment.

De la même façon, le **ou-logique** peut remplacer la structure `unless`; exemple:

```
ceci || cela;
```

Si `ceci` est vraie, alors inutile d'évaluer `cela`; sinon, `cela` est évaluée.

Enfin, il existe l'opérateur ternaire:

```
exp1 ? exp2 : exp3 ;
```

qui évalue `exp2` si `exp1` est vraie, et `exp3` sinon. Cela revient à:

```
if (exp1) { exp2 ; } else { exp3 ; }
```

mais il n'y a pas toute cette ponctuation!

2.9. LES MANIPULATEURS DE FICHIERS ET LES TESTS SUR LES FICHIERS

2.9.1. Les manipulateurs de fichiers ou filehandles

Ils ont été présentés au paragraphe 1.1.1., mais ça ne fait jamais de mal de répéter un peu!

Un manipulateur de fichier est le nom, dans un programme Perl, d'une connexion entrée/sortie entre le processus Perl et le monde extérieur.
Nous en avons déjà utilisé; par exemple `STDIN` est le filehandle désignant la connexion entre le processus Perl et l'entrée standard d'Unix. Perl dispose aussi de `STDOUT` et `STDERR`.

Un nom de filehandle suit les mêmes règles que les noms d'étiquettes, de tableaux, de subroutines ... Cependant, comme les étiquettes, il n'est précédé par aucun symbole particulier. Il est donc conseillé d'utiliser des majuscules et de ne pas interférer avec les mots réservés.

2.9.2. Ouverture et fermeture de filehandle

`STDIN`, `STDOUT` et `STDERR` sont ouverts automatiquement par le père du processus Perl. Pour ouvrir d'autres filehandles, on utilise la fonction **open()**; exemple:

```
open ( MANIP, "nomfic");          # ouverture en lecture
```

`MANIP` est le nouveau manipulateur de fichier et *nomfic* est le nom du fichier Unix associé à ce nouveau filehandle. Cette instruction ouvre le filehandle en lecture; pour l'ouvrir en écriture, on utilise encore `open` mais on fait précéder le nom du fichier par le signe supérieur:

```
open ( MANIP, "> nomfic");        # ouverture en écriture
```

Pour ouvrir un fichier et écrire à la suite de ce qu'il contient déjà, il suffit d'écrire:

```
open ( MANIP, ">> nomfic");
```

Ces trois formes de `open` retournent *vrai* si l'ouverture s'est déroulée avec succès, *faux* sinon.

Pour fermer un filehandle, on utilise l'opérateur **close()** comme suit:

```
close ( MANIP);
```

Rouvrir un filehandle a pour effet de refermer automatiquement le fichier associé par le précédent `open`.
La fin d'exécution du programme Perl a pour effet de refermer automatiquement tous les fichiers. `close()` n'est donc pas forcément nécessaire.

2.9.3. L'opérateur die()

Un filehandle qui n'a pas été ouvert avec succès peut continuer d'être utilisé sans qu'il y ait aucune indication de la part du programme. Si une lecture y est effectuée, le caractère de fin de fichier est retourné; toute donnée qu'on essaiera d'y écrire sera perdue.
Bien sûr, on peut utiliser une structure de contrôle avec un message d'erreur s'affichant si l'ouverture n'a pas marché.

Mais Perl permet, ici encore, de se simplifier la vie: l'opérateur `die()` prend comme argument une chaîne de caractères qu'il va écrire sur l'erreur standard, et tue le processus Perl (celui du programme) avec un *exit* Unix de statu non nul. Combiné avec le *ou-logique*, notre test est maintenant très simple:

```
open(MANIP, "> nomfic") || die ("Impossible d'ouvrir nomfic!");
```

Cela se lit: « Ouvre ce fichier ou meurt! ». `die()` n'est exécuté que si `open` retourne *faux*.
Le message renvoyé à la mort du processus comporte le nom du programme Perl et le numéro de ligne où ça n'a pas marché. Si on ne désire pas avoir ces informations, il suffit de terminer la chaîne de caractères de `die()` par le caractère "\n".

2.9.4. Utilisation des filehandles

Un filehandle ouvert en lecture peut être lu tout comme `STDIN`; exemple:

```
open(EP, "/etc/passwd");
while (<EP>)
{ chop;
   print "j'ai vu $_ dans le fichier des mots de passe! ";
}
```

Un filehandle ouvert en écriture doit être passé en argument à `print` comme suit:

```
print MANIP "fini de rigoler!";
```

Par ailleurs, nous avons vu que, par défaut, la sortie se fait sur `STDOUT`:

```
print STDOUT "coucou!";          # équivaut à print "coucou! ";
```

2.9.5. Les tests sur les fichiers

Ils sont regroupés dans le tableau suivant:

-r	fichier ou répertoire accessible en lecture
-w	fichier ou répertoire accessible en écriture
-x	fichier ou répertoire exécutable
-o	fichier ou répertoire propriété de l'utilisateur
-R	fichier ou répertoire pouvant être lu par l'utilisateur réel, et non par l'utilisateur effectif (diffère de -r pour les programmes setuid)
-W	fichier ou répertoire pouvant être écrit par l'utilisateur réel, et non par l'utilisateur effectif (diffère de -w pour les programmes setuid)
-X	fichier ou répertoire exécutable par l'utilisateur réel, et non par l'utilisateur effectif (diffère de -x pour les programmes setuid)
-O	fichier ou répertoire propriété du l'utilisateur réel, et non de l'utilisateur effectif (diffère de -o pour les programmes setuid)
-e	fichier ou répertoire existant
-z	fichier existant, de taille nulle
-s	fichier ou répertoire existant et comptant un nombre non nul de caractères
-f	l'entrée est un fichier
-d	l'entrée est un répertoire
-l	l'entrée est un lien symbolique

-S	l'entrée est une socket
-p	l'entrée est un pipe portant un nom (fifo)
-b	l'entrée est un fichier avec bloc spécial
-c	l'entrée est un fichier de caractères spéciaux
-u	fichier ou répertoire ayant le bit 11 (setuid) positionné à 1
-g	fichier ou répertoire ayant le bit 10 (setgid) positionné à 1
-k	fichier ou répertoire ayant le bit s positionné
-t	isatty() sur le filehandle retourne vrai
-T	fichier texte
-B	fichier binaire
-M	age de la dernière modification du fichier en jours
-A	age du dernier accès au fichier en jours
-C	age de la dernière modification d'inode du fichier en jours

La plupart de ces tests retournent *vrai* ou *faux*.

Cependant, l'opérateur `-s` retourne *vrai* si le fichier est non vide, mais c'est un *vrai* particulier: c'est le nombre d'octets du fichier, qui est évalué à *vrai* si ce n'est pas 0.

Les opérateurs d'age `-M`, `-A`, `-C` retournent le nombre de jours qui se sont écoulés depuis la dernière modification, le dernier accès au fichier et la dernière modification d'inode de ce fichier. Cet age a une résolution de 1 seconde et peut avoir une valeur décimale: 36 heures sont notées 1,5 jours.

Tous ces opérateurs peuvent prendre comme paramètre aussi bien un filehandle qu'un nom de fichier. Si on ne leur passe pas de paramètre, ils prennent comme opérande par défaut le fichier dont le nom est contenu dans la variable `$_`; par exemple, pour tester en lecture une liste de nom de fichiers il suffit d'écrire:

```
foreach ( @liste_nom_fichiers )
{ print "$_ est accessible en lecture \n" if -r ; }
```

2.9.6. Les opérateurs stat() et lstat()

Pour obtenir toute autre information sur un fichier, on utilise l'opérateur `stat()` qui retourne pratiquement tout ce que l'appel système *stat()* retourne. Cet opérateur prend un filehandle ou une expression s'évaluant à un nom de fichier comme paramètre. Il retourne soit la valeur *undef* si l'instruction a échoué, soit un tableau de 13 éléments, décrit en détail dans la page man de *stat(2)*.

L'appel à `stat()` avec un nom de lien symbolique renvoie ce sur quoi ce lien pointe, mais pas d'information sur le lien lui-même. Pour obtenir ces informations, on dispose de l'opérateur `lstat()`.

Comme pour les tests sur les fichiers, `stat()` et `lstat()` prennent `$_` comme paramètre par défaut.

2.9.7. Le filehandle _

A chaque utilisation de `-r`, `-w`, `stat()` etc, Perl doit demander au système un buffer *stat* au sujet du fichier (le buffer résultat de l'appel système *stat*). Cela signifie que si l'on demande si un fichier est accessible en écriture et en lecture, Perl ira chercher deux fois la même information.

Pour éviter ça, on dispose du filehandle _. Passé comme paramètre à un test de fichier, à `stat()` ou à `lstat()`, il indique à Perl de réutiliser le buffer *stat* obtenu lors du dernier test de fichier ou du dernier appel à `stat()` ou `lstat()`. Exemple:

```
if ( -r $nomfic && -w _ )
{ print "$nomfic est accessible en écriture et en lecture \n"; }
```

Pour le premier test, on utilise `$nomfic` pour obtenir du système d'exploitation les données concernant ce fichier; pour le deuxième test, par contre, on utilise _: les données obtenues par le premier test sont réutilisées.

2.10. LES FORMATS

2.10.1. Qu'est-ce qu'un format?

Perl dispose d'un outil de mise en page, appelé **format**. Un format définit une partie constante (entête, labels, texte fixe, etc.) et une partie variable (les données auxquelles va s'appliquer le format). L'utilisation d'un format consiste en trois points:

1. définir le format
2. charger les données à afficher dans les parties variables du format
3. faire appel au format

Le premier point est effectué une fois, n'importe où dans le texte du programme, les deux autres sont effectués de façon répétitives.

2.10.2. Définition d'un format

Un format est défini par une définition de format. Celle-ci peut figurer n'importe où dans le texte du programme, comme les subroutines. Elle est de la forme:

```
format nomdeformat =
ligne de champ
valeur_1, valeur_2, valeur_3
ligne de champ
valeur_1, valeur_2, valeur_3
ligne de champ
valeur_1, valeur_2, valeur_3
.
```

La première ligne contient le mot réservé `format`, suivi du nom du format et du signe =. Le nom du format doit vérifier les mêmes règles que les autres noms de variables. Il peut cependant être un mot réservé, puisqu'il n'est pas utilisé dans le corps du programme (si ce n'est en tant que chaîne). Vient ensuite le **gabarit**, dont la fin est indiquée par une ligne contenant un seul et unique point. Attention, les gabarits sont sensibles aux espaces.

La définition du gabarit contient un certain nombre de lignes de champ. Chacune d'elles peut contenir du texte fixe, texte qui sera affiché tel quel à chaque appel du format. Les lignes de champ peuvent comporter des **fieldholders** pour le texte variable; exemple:

```
Bonjour! Mon nom est @<<<<<<<<<<
$nom
```

Le *fieldholder* est `@<<<<<<<<<<`; il spécifie un champ de 11 caractères avec alignement gauche.

Si une ligne comporte plusieurs *fieldholders*, on doit spécifier autant de valeurs; celles-ci sont alors séparées par une virgule; exemple:

```
Bonjour! Mon nom est @<<<<<<<<<<; j'ai @<< ans
$nom, $age
```

Une ligne de champ ne comportant pas de *fieldholders* n'est donc pas suivie par une ligne de valeurs.

Les espaces sont ignorés dans la ligne de valeurs. Les valeurs sont prises en compte en tant que scalaires; donc toute expression est évaluée dans un contexte scalaire.

2.10.3. L'appel à un format

On fait appel à un format avec l'opérateur **write**. Celui-ci prend le nom d'un filehandle comme paramètre et génère du texte pour ce filehandle, avec le format courant de ce filehandle. Le format par défaut est le format du même nom (pour `STDOUT`, le format `STDOUT` est utilisé par défaut). Voyons un exemple concret:

```
format ETIQUETTEADRESSE=
================================
| @<<<<<<<<<<<<<< @<<<<<<<<<<<<<|
$nom,                   $prenom
| @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<|
$adresse
| @<<<< @<<<<<<<<<<<<<<<<<<<<<<<|
$code_postal, $ville
================================
.

open(ETIQUETTEADRESSE, ">etiquette-a-imprimer") ||
                 die ("Création impossible! ");
open (ADRESSES, "adresses") ||
                 die ("Impossible d'ouvrir adresses! ");
while (<ADRESSES>)
```

```
{ chop;
   ($nom, $prénom, $adresse, $code_postal, $ville) =
                                                  split (/:/);
   write ETIQUETTEADRESSE;
}
```

On commence par définir un format pour les étiquettes. Notez qu'on lui donne le même nom qu'au filehandle utilisé pour la sortie. On ouvre celui-ci et le filehandle contenant les adresses, qui sont censées avoir le format suivant:

```
Poiraud:Cecile:19 avenue du docteur Donat:06800:Cagnes-sur-Mer
```

Chaque ligne de ce filehandle est lue, découpée, et les 5 morceaux sont affectés aux 5 champs de l'étiquette. Notez que les noms des variables recueillant les morceaux sont les mêmes que ceux utilisés dans la définition du format; ça aussi, c'est important.
Enfin, l'opérateur `write` fait appel au format. Notez cependant que le paramètre de `write` est le filehandle sur lequel est effectuée la sortie et que le format du même nom est utilisé par défaut.
Chaque champ du format est alors remplacé par la valeur correspondante; exemple:

```
================================
| Poiraud         Cecile       |
| 19 avenue du docteur Donat   |
| 06800 Cagnes-sur-Mer         |
================================
```

2.10.4. Compléments sur les fieldholders

- Les champs de texte

  La plupart des *fieldholders* commencent par @. Les caractères suivant @ indiquent le type du champ; le nombre de caractères (en incluant @) définit la taille du champ.
  Si @ est suivi de signes *inférieur* (<), alors le champ est à alignement gauche. Si la valeur ne remplit pas totalement le champ, celui-ci est complété à droite avec des espaces. Si la valeur est trop longue, elle est automatiquement tronquée.
  Si @ est suivi de signes *supérieur* (>), alors le champ est à alignement droit (si la valeur est trop courte, le champ est complété à gauche par des espaces).
  Si @ est suivi de barres *verticales* (|), alors le champ est centré (si la valeur est trop courte, des espaces sont rajoutés de part et d'autre pour qu'elle soit à peu près centrée).

- Les champs numériques

  Il existe également des champs numériques de précision fixe. Ces champs commencent aussi par @, et sont suivis d'un ou plusieurs # avec, en option, un point (la virgule décimale). Ici encore, @ compte comme un caractère du champ; exemple:

```
format COMPTE_CL =
Credit: @#####.## Debit: @#####.## Total: @#####.##
$credit, $debit, $credit-$debit
.
```

  Notez l'utilisation d'une expression dans le format.

- Les champs multilignes

  Perl s'arrête normalement au premier *newline* rencontré lorsqu'il charge les données dans le format. Cependant, il existe un *fieldholder* permettant d'inclure autant de lignes d'informations que l'on veut; on le note @*, seul sur une ligne. La ligne suivante définit la valeur qui remplira le champ. Exemple:

```
format STDOUT =
++++++++++++++
@*
$chaine_des_noms
++++++++++++++
.

$chaine_des_noms =
       "cricri\nseb\ngilles\nxav\npascal\nhugo\nsissou\n";
write;
```

  a pour résultat:

```
++++++++++++++
cricri
seb
gilles
xav
pascal
hugo
sissou
++++++++++++++
```

- Les champs paragraphes

  Ils permettent de créer des paragraphes, les lignes étant coupées à la **fin d'un mot**. Le *fieldholder* est le même que pour les champs de texte, si ce n'est qu'il commence par ^ (par exemple, ^<<<<<).
  La valeur correspondante doit être une variable scalaire contenant du texte.

Lorsque Perl remplit le champ paragraphe, il prend autant de mots dans la variable qu'il est possible d'en mettre dans le champ. Ces mots sont éliminés de la variable. Exemple:

```
format LES_GENS =
Nom: @<<<<<<<<< Commentaires: ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
     $nom,                    $commentaire
                               ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
                               ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
                               ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
.
```

La même variable, `$commentaire`, apparaît 4 fois. La première ligne affiche le nom de la personne et les premiers mots de `$commentaire`. Ces mots sont éliminés de la variable et, avec la deuxième ligne, l'affichage des commentaires se poursuit, de même qu'aux lignes 3 et 4.

Si *`$commentaire`* ne contient pas assez de mots pour remplir les 4 lignes, elles sont complétées par des espaces. Pour éviter l'impression de lignes blanches, il suffit d'utiliser l'indicateur de suppression tilde (~), qui est transparent à l'impression et peut figurer n'importe où dans la ligne sur laquelle il porte; exemple:

```
format LES_GENS =
Nom: @<<<<<<<<< Commentaires: ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
     $nom,                    $commentaire
     ~                        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
     ~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
     ~                         ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
.
```

Si `$commentaire` est trop longue, inutile de copier 20 fois les deux dernières lignes en espérant que ça suffira. Perl permet de faire ça tranquillement. Toute ligne contenant deux tildes successifs (~~) sera répétée automatiquement jusqu'à ce qu'elle affiche une ligne entièrement vide. Notre exemple devient alors:

```
format LES_GENS =
Nom: @<<<<<<<<< Commentaires: ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
     $nom,                    $commentaire
~~                             ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                              $commentaire
.
```

2.10.5. Le format haut-de-page

Perl permet de définir un format *haut-de-page*, qui enclenche un mode de pagination. Perl compte le nombre de lignes générées par n'importe quel appel à un format pour un même filehandle. Lorsque le format courant ne loge pas sur ce qu'il reste de la page, Perl place un *formfeed* , effectue un appel automatique au format haut-de-page et continue l'affichage sur la page suivante.

Le format haut-de-page est défini comme n'importe quel format. Son nom par défaut est le nom du filehandle suivi de **_TOP**.
La variable **$%** contient le nombre d'appels au format haut-de-page pour un même filehandle; elle peut donc être utilisée pour numéroter les pages. L'exemple suivant de format haut-de-page permet d'éviter que les étiquettes de l'exemple précédent soient coupées au changement de page et de numéroter les pages d'étiquettes:

```
format ETIQUETTEADRESSE_TOP =
Mes Adresses -- Page @<
                     $%
.
```

La longueur par défaut d'une page est de 60 lignes.

2.10.6. Remplacer les valeurs par défaut par des formats

- L'opérateur **select()**

  Nous avons vu que le filehandle par défaut de `print` est `STDOUT`. En fait, pas vraiment. Le véritable filehandle par défaut de `print`, de `write`, est le **filehandle courant**.
  `STDOUT` est le premier filehandle courant; cependant, on peut modifier ce filehandle avec l'opérateur `select()`, qui prend comme unique paramètre un nom de filehandle. Une fois que le filehandle courant est modifié, cela affecte toutes les opérations à venir faisant appel à lui. Exemple:

```
print "Bonjour tout le monde! ";     # affichage sur STDOUT
select (LOGFILE);
# le filehandle courant est maintenant LOGFILE
print "Coucou! ";              # affichage dans LOGFILE
select (STDOUT);
# STDOUT est de nouveau le filehandle courant
print "Ca va? ";               # affichage sur STDOUT
```

Il faut être prudent! Des subroutines peuvent modifier le filehandle courant; on peut alors avoir des surprises. Heureusement, `select()` retourne le nom de l'ancien filehandle courant, ce qui permet de le rétablir en fin d'exécution de la subroutine. Exemple:

```
$vieux_nom = select(LOGFILE);
# sauvegarde du nom de l'ancien filehandle courant
print "Coucou! ";
select($vieux_nom);       # restaure le filehandle précédent
```

- Pour changer le nom d'un format

Le format par défaut pour un certain filehandle est le format du même nom. Cependant, on peut associer au filehandle courant le format dont le nom figure dans la variable **$~**. Cette variable contient le format courant du filehandle courant.
Si l'on veut utiliser le format `ETIQUETTEADRESSE` pour `STDOUT`, il suffit d'écrire:

```
$~ = ETIQUETTEADRESSE;
```

Si l'on veut associer le format `RESUME` au filehandle `RAPPORT`:

```
$vieux_nom = select(RAPPORT);
# changement de filehandle courant
$~ = "RESUME";
# format pour le filehandle courant
    ...                          # code
write RAPPORT;
# impression dans RAPPORT avec le format RESUME
select($vieux_nom);
# retour à l'ancien filehandle courant
```

- Pour changer le nom du format haut-de-page

Il suffit de donner à la variable **$^** le nom du format haut-de-page à appliquer au filehandle courant. On peut consulter cette variable pour connaître le format haut-de-page du filehandle courant.

- Pour changer la longueur d'une page

Nous avons vu que, par défaut, une page contient 60 lignes. Cependant, on dispose de la variable **$=**; elle contient le nombre courant de lignes par page pour le filehandle courant et peut être modifiée.

- Pour changer de position dans la page

  Si on `print` du texte dans un filehandle, le compteur de lignes du filehandle devient faux puisque Perl ne compte que les lignes issues d'un `write`. Si l'on veut que le compteur de Perl reste à jour, il suffit de modifier la variable **`$-`**. Cette variable contient le nombre de lignes restant dans la page courante du filehandle courant. Chaque `write` décrémente cette variable du nombre de lignes effectivement imprimées; lorsque `$- = 0`, un appel au format haut-de-page est effectué et `$-` prend pour valeur `$=`.
  En début de programme, `$-` vaut zéro pour tous les filehandles.

2.11. LES REPERTOIRES

2.11.1. Se déplacer dans l'arborescence

L'opérateur Perl **chdir()** permet de modifier le répertoire courant d'un processus. Il prend comme paramètre une expression s'évaluant à un nom de répertoire et attribue cette valeur au répertoire courant. Il retourne *vrai* si cette modification a bien pu être effectuée, *faux* sinon. Exemple:

```
chdir ( "/etc" ) || die ( "cd /etc impossible!" ) ;
```

Les parenthèses ne sont pas obligatoires.

Chaque processus Unix possède son propre répertoire courant. Lorsqu'un nouveau processus est créé, il hérite ce répertoire de son père. Si un programme Perl change son répertoire, cela n'affecte pas le shell qui a lancé le processus Perl. De même, les processus créés par un programme Perl ne peuvent pas modifier le répertoire courant de ce programme.

2.11.2. Le globbing

En shell, une simple `*` permet d'accéder à la liste des fichiers du répertoire courant. De même, `[a-m]*.c` donne accès à la liste des fichiers dont le nom commence par une lettre de la première moitié de l'alphabet et se termine par `.c`.
Ce genre d'action s'appelle le *globbing*. Perl l'accepte à condition que le *globbing pattern* soit encadré par les signes < et >; exemple:

```
@liste = </etc/host*>;
# @liste contient la liste des noms de fichiers du répertoire
/etc commençant par host
```

Cela ressemble beaucoup à la lecture d'un filehandle:

dans un contexte scalaire, le glob retourne le nom de fichier matché ou *undef* s'il n'y en a pas;

- dans un contexte de tableau, le glob retourne la liste des noms de fichier matchés restants, ou la liste vide s'il n'y en a pas.

Exemple d'utilisation:

```
while ($nomfic = </etc/host*>)
{ print "Un des fichiers est $nomfic \n " ; }
```

On peut avoir plusieurs patterns à l'intérieur du glob: les listes sont construites séparément puis concaténées; exemple:

```
@liste_ceci_cela = <ceci* cela*>;
```

Bien que globbing et matching fonctionnent de façon assez similaire, il ne faut pas oublier que la signification des divers caractères spéciaux est différente: `<\.c$>` ne permet pas d'obtenir la liste des noms de fichier ayant pour extension `.c` !

L'interpolation de variable a lieu dans le glob avant exécution de celui-ci; exemple:

```
if (-d "/usr/etc" )
{ $ou = "/usr/etc" ; }
else
{ $ou = "/etc" ; }
@liste = < $ou/* > ;
```

Une exception cependant: le pattern `<$var>` (signifiant qu'il faut utiliser la variable `$var` comme globbing pattern) doit être écrit `<${var}>` pour des raisons que nous verrons plus tard.

2.11.3. Les manipulateurs de répertoire

Un manipulateur de répertoire est une connexion avec un répertoire particulier. Il est toujours ouvert en lecture uniquement: on ne peut pas l'utiliser pour changer un nom de fichier ou détruire un fichier.
Passé en paramètre à la fonction `readdir` et à ses copines, il permet de manipuler le répertoire auquel il est associé.

Un nom de manipulateur de répertoire suit les mêmes règles qu'un nom de filehandle: les majuscules sont conseillées et les mots réservés interdits. Le manipulateur de fichier `MANIP` et le manipulateur de répertoire `MANIP` sont indépendants.

2.11.4. Ouverture et fermeture d'un manipulateur de répertoire

La fonction **opendir()** prend pour paramètres le nouveau manipulateur de répertoire et le nom du répertoire à ouvrir, auquel le manipulateur sera associé. Cette fonction retourne *vrai* si le répertoire a pu être ouvert, *faux* sinon. Exemple:

```
opendir ( ETC, "/etc" ) || die ("opendir n'a pas marché! \n" ) ;
```

Pour fermer un manipulateur de fichier, on utilise la fonction **closedir()**; exemple:

```
closedir(ETC);
```

Comme *close()*, *closedir()* n'est pas souvent nécessaire puisque tous les manipulateurs de répertoires sont fermés automatiquement avant d'être rouverts ou en fin d'exécution du programme Perl.

2.11.5. Lecture avec un manipulateur de répertoire

Lorsqu'un manipulateur de répertoire a été ouvert, la liste des noms de fichiers contenus dans ce répertoire peut être lue grâce à la fonction **readdir()**. Cette fonction prend comme unique argument le manipulateur de répertoire et retourne:

- dans un contexte scalaire, le nom de fichier courant, dans un ordre aléatoire; *undef*, s'il ne reste plus de nom;
- dans un contexte de tableau, les noms de fichiers restant sous forme de liste dont chaque élément est un nom.

Exemples:

```
opendir (ETC, "/etc") || die ("opendir n'a pas marché!") ;
while ( $nom = readdir(ETC) )          # contexte scalaire
{ print "$nom \n" ; }
closedir(ETC) ;

opendir (ETC, "/etc") || die ("opendir n'a pas marché!") ;
foreach $nom ( sort readdir(ETC) )
# contexte de tableau, tri
{ print "$nom \n" ; }
closedir(ETC) ;
```

2.12. MANIPULATION DE FICHIERS ET DE REPERTOIRES

2.12.1. Destruction d'un fichier

L'opérateur Perl **unlink()** détruit un nom de fichier (qui pourrait avoir plusieurs noms). Lorsque le dernier nom d'un fichier est détruit, le fichier lui-même est détruit. C'est l'équivalent de la commande *rm*. Comme la plupart du temps un fichier a un nom unique, détruire son nom, c'est donc le détruire. Exemple:

```
unlink($nomfic);
# le fichier dont le nom est référencé par la variable $nomfic
est détruit
```

L'opérateur `unlink()` admet une liste de noms de fichier comme paramètre; tous les fichiers dont les noms figurent dans cette liste sont détruits. Exemples:

```
unlink("fic1", "fic2");          # fic1 et fic2 sont détruits
unlink(<*.o>);
# tous les fichiers du répertoire courant ayant pour extension
.o sont détruits
```

`unlink()` retourne le nombre de fichiers effectivement détruits et prend `$_` comme paramètre par défaut; exemple:

```
foreach (<*.o>)
{ unlink || print "Impossible de détruire $_ \n " ; }
```

Tous les fichiers objet du répertoire courant sont détruits successivement; un message permet de connaître les noms des fichiers ne pouvant être détruits.

2.12.2. Renommage d'un fichier

Cette opération est effectuée avec l'opérateur **rename()**, comme suit:

```
rename ($old, $new);
rename ("fic", "fic1");
# le fichier fic est renommé fic1.
```

Cet opérateur retourne la valeur *vrai* si le renommage a pu être effectué.

Si la commande shell *mv* permet le raccourci `mv fichier un_répertoire`, la commande `rename()` ne le permet pas; il faut écrire:

```
rename("fichier", "un_répertoire/fichier");
```

2.12.3. Création de liens sur un fichier

Comme si un nom pour un fichier ne suffisait pas, parfois on en veut deux ou trois ou une douzaine. Cette opération de création de plusieurs noms pour un même fichier s'appelle le **linking**. Il y a deux sortes de linking: établissement de lien physique et établissement de lien symbolique.

- Liens physique et symbolique

  Un lien physique sur un fichier n'est pas distinguable du nom du fichier.
  Le noyau Unix met à jour un compteur qui est le nombre de liens physiques se rapportant à un fichier. Si un lien est détruit, ce compteur est décrémenté; si un nouveau lien est créé, le compteur est incrémenté. Lorsque le fichier est créé, le compteur prend la valeur 1.

  Chaque lien physique sur un fichier doit se trouver sur la même *partition montée*.
  On ne peut créer qu'un seul lien physique sur un répertoire: celui qui est établi lors de la création du répertoire. Toute tentative d'en créer un deuxième est fatale. C'est pour protéger la hiérarchie Unix des fichiers.

  Un lien symbolique est une forme spéciale de fichier qui contient comme données un chemin d'accès. Lorsque ce fichier est ouvert, le noyau Unix considère son contenu comme un aiguillage supplémentaire pour le chemin d'accès; il continue alors son avance dans la hiérarchie.

  Un lien symbolique peut être créé sur un répertoire.

Le contenu d'un lien symbolique n'a pas besoin d'être un fichier ou un répertoire existant. Il peut même se trouver sur une autre partition montée. Un lien symbolique peut pointer sur un autre lien symbolique.

Un lien physique évite la perte du contenu d'un fichier (puisqu'il est en fait un nom du fichier). Un lien symbolique ne peut empêcher cette perte.

Il faut être autorisé en écriture dans le répertoire où l'on crée ces deux sortes de liens, sans pour autant être autorisé à ouvrir le fichier sur lequel sont créés les liens. Il faut pouvoir appliquer `stat()` au fichier sur lequel on veut créer un lien physique; ceci n'est pas nécessaire pour créer un lien symbolique.

- Création de liens physique et symbolique avec Perl

L'opérateur **link()** permet de créer des liens physiques. Il prend comme paramètres le vieux nom du fichier et le nouvel alias pour ce fichier. Il retourne *vrai* si le lien a pu être créé. Comme avec `rename()`, il faut préciser dans son entier le nouveau nom de fichier (et pas seulement le répertoire). Exemple:

```
link("fic1", "fichier_1") || die("Impossible de créer un lien
physique entre fic1 et fichier_1 \n");
```

Pour un lien physique, le vieux nom ne peut pas être celui d'un répertoire et le nouvel alias doit être sur la même partition montée.

Sur les systèmes admettant les liens symboliques, l'opérateur **symlink()** permet de créer cette sorte de lien. Il prend comme paramètres le vieux nom du fichier ou du répertoire et le nouvel alias pour ce fichier ou ce répertoire. Le fichier ou le répertoire n'a pas besoin d'exister pour qu'il soit possible de créer un lien symbolique sur lui. Exemple:

```
symlink("fic2", "fichier_2") || die("Impossible de créer un
lien symbolique entre fic2 et fichier_2 \n");
```

L'opérateur **readlink()** permet de savoir ce sur quoi le lien symbolique passé en paramètre pointe. Exemple:

```
if ($nom = readlink("fichier_2"))
{ print "fichier_2 pointe sur $nom \n" ; }
# $nom contient fic2
```

`readlink()` retourne *undef* si le lien testé n'existe pas, ne peut être lu ou n'est pas un lien symbolique.

2.12.4. Création et destruction d'un répertoire

L'opérateur **mkdir()** permet de créer un nouveau répertoire dont le nom et les droits lui sont passés en paramètres. Les droits sont spécifiés par un entier (le mode) interprété comme un format de droits internes. La page man *chmod(2)* précise les droits internes. Si vous êtes pressés, le mode 0777 permet de tout faire pratiquement. Exemple:

```
mkdir("rep", 0777);
# création du répertoire rep avec le mode 0777
```

L'opérateur **rmdir()** permet de supprimer un répertoire vide; exemple:

```
rmdir("rep");          # destruction du répertoire rep
```

2.12.5. Modification de droits

Les droits sur un fichier ou un répertoire indiquent qui a le droit de faire quoi avec ce fichier ou ce répertoire. L'opérateur Perl **chmod()** permet de changer ces droits. Il prend comme paramètres un entier, le mode, et une liste de noms de fichiers, et tente de modifier les droits de tous ces fichiers avec le mode indiqué. Exemple:

```
chmod(0666, "fic1", "fic2");
```

Avec le mode 0666, *fic1* et *fic2* deviennent accessibles en lecture et en écriture pour tout le monde (utilisateur, groupe et autres).

`chmod()` retourne le nombre de fichiers dont il a réussi à modifier les droits.

2.12.6. Modification de la propriété

Tout fichier ou répertoire a un propriétaire et un groupe dans le système de fichiers. Le propriétaire et le groupe déterminent à qui les droits de propriété et de groupe s'appliquent. Ils sont déterminés au moment de la création du fichier, mais on peut les changer.

L'opérateur **chown()** prend comme paramètres un UID, un GID et une liste de noms de fichiers, et tente de changer la propriété de chacun des fichiers spécifiés. Il retourne le nombre de fichiers dont il a réussi à modifier la propriété.

Remarquez qu'on change le propriétaire et le groupe en même temps.
UID et GID doivent être des valeurs numériques, et non les noms symboliques correspondants. Exemple:

```
chown(1234, 35, "fic1", "fic2");
```

fic1 et *fic2* appartiennent maintenant à l'utilisateur d'UID 1234 et au groupe de GID 35.

2.12.7. Modification des timestamps

Trois *timestamps* sont associés à un fichier. Nous en avons parlé brièvement au paragraphe 2.9.4. Il s'agit des âges de dernière modification, de dernier accès d'un fichier et de dernière modification de l'inode de ce fichier.

L'opérateur **utime()** permet de donner aux deux premiers timestamps des valeurs arbitraires. Modifier ces deux valeurs modifie automatiquement la troisième et lui donne pour valeur le temps courant. Donc, pas besoin d'opérateur spécial pour ça!

`utime()` prend comme paramètres les âges en secondes du dernier accès et de la dernière modification, et une liste de noms de fichiers; il retourne le nombre de fichiers pour lesquels l'opération a réussi. Exemple:

```
$atime = $mtime = 700 000 000;
utime($atime, $mtime, "fic1", "fic2");
```

Les temps sont mesurés en temps interne Unix; ce sont des valeurs entières de secondes, comptées à partir du 01/01/70 G.M.T.

2.13. LA GESTION DE PROCESSUS

2.13.1. L'utilisation de system()

Perl peut lancer de nouveaux processus. La façon la plus simple de le faire est d'utiliser l'opérateur `system()`. Cet opérateur donne en fait la chaîne qu'on lui passe en paramètre à un shell */bin/sh* qui l'exécute comme une commande. Quand l'exécution est terminée, `system()` retourne **0** si tout s'est bien passé. Exemple:

```
system("date");  # permet d'obtenir la date
```

L'opérateur `system()` hérite du processus Perl les fichiers d'entrée, de sortie et d'erreur standards. Le résultat de la commande shell sera donc dans `STDOUT`. Cependant il peut être redirigé; exemple:

```
system("date > ficdate") &&
                die "Impossible de créer ficdate \n";
# permet d'obtenir la date dans le fichier ficdate et de
vérifier la valeur de retour de system()
```

`system()` peut prendre comme paramètre tout ce que */bin/sh* accepte d'exécuter: plusieurs commandes séparées par des points virgules, des retours à la ligne, des lancements en arrière plan... Exemple:

```
system("(date ; who) > $fic & ") ;
# lancement de date et who en arrière plan avec sortie dans le
ficher référencé par la variable Perl $fic
```

Perl offre la possibilité de consulter et de modifier les variables d'environnement grâce au tableau associatif **`%ENV`**, dont chaque élément est une paire (nom de variable d'environnement, valeur de cette variable). Voici un exemple permettant d'afficher toutes les variables d'environnement et leur valeur:

```
foreach $key (sort keys %ENV)
{ print "$key = $ENV{$key} \n" ; }
```

L'opérateur `system()` admet également une liste de paramètres. Plutôt que de donner cette liste au shell, Perl interprète le premier élément de cette liste comme la commande à exécuter avec pour paramètres les éléments suivants de la liste. Exemple:

```
system "grep 'seb' fic_speinf";
system "grep", "seb", "fic_speinf";
# idem mais avec une liste
```

L'utilisation d'une liste plutôt que d'une chaîne économise un processus shell.

2.13.2. L'utilisation des backquotes

Une autre façon de lancer un processus est d'encadrer une ligne de commande */bin/sh* avec des backquotes. La commande est exécutée et le résultat est récupéré sur la sortie standard. Il devient la valeur de la chaîne entre backquotes; exemple:

```
$date = "on est aujourd'hui".`date`;
# $date contient maintenant la concaténation de la chaîne de
caractères et du résultat de la commande shell date
```

Si la commande entre backquotes est utilisée dans un contexte de tableau, le résultat est une liste de chaînes, chacune étant une ligne (terminée par un *newline*) de la sortie de la commande. Pour l'exemple ci-dessus, nous n'aurions eu qu'un élément, puisqu'une seule ligne est générée. Autre exemple:

```
foreach (`who`)
{ ($qui, $ou, $quand) = / (\S+) \s+ (\S+) \s+ (.*)/ ;
   print "$qui sur $ou à $quand \n" ; }
```

2.13.3. Utilisation de processus comme filehandles

Une autre méthode pour lancer un processus est de créer un processus qui ressemble à un manipulateur de fichier. Comme les filehandles sont ouverts soit en écriture soit en lecture, il est possible de créer un processus-filehandle qui peut capturer la sortie ou fournir l'entrée du processus; exemple:

```
open (PROC, "who | " ) ;
@who = <PROC>;
```

La barre verticale à droite de `who` indique à Perl qu'il n'est pas en train d'ouvrir un filehandle mais plutôt de lancer une commande. Comme la barre est à droite de la commande, le filehandle est ouvert en lecture, ce qui signifie que la sortie de `who` doit être capturée. La deuxième ligne permet de ranger le résultat de la commande dans un tableau dont chaque élément est une ligne du `who`.

De même, pour lancer une commande avec des paramètres, on peut ouvrir un processus-filehandle en écriture en mettant la barre verticale à gauche de la commande; exemple:

```
open (LPR, " | lpr -Pslatewriter" ) ;
print LPR @report ;
close(LPR) ;
```

Après avoir ouvert `LPR`, on y a écrit des données puis on l'a fermé. Ouvrir un processus-filehandle permet à la commande de s'exécuter en parallèle avec le programme Perl. `close()` force ce programme à attendre que le processus meurt. Si on ne ferme pas le processus-filehandle, le processus peut continuer de s'exécuter même après terminaison du programme Perl.

2.13.4. L'utilisation de fork

Une autre méthode pour lancer un nouveau processus est de cloner le processus Perl avec la primitive Unix *fork*.
L'opérateur Perl `fork` fait simplement ce que l'appel système *fork* exécute: il crée un processus fils, qui partage avec son père le même code exécutable, les mêmes variables et même les mêmes fichiers ouverts. Pour distinguer les deux processus, la valeur de retour de `fork` est 0 pour le fils et une valeur non nulle pour le père. Exemple:

```
if (fork)
{ # je suis le père }
else
{ # je suis le fils }
```

Perl dispose également des opérateurs **wait**, **exec** et **exit**. En fait, il se contente de passer les appels à ces opérateurs aux appels système d'Unix.

`exec` est comme l'opérateur `system`, si ce n'est qu'au lieu de créer un nouveau processus pour exécuter une commande shell, il remplace le processus courant par le shell. Pour obtenir quelque chose d'équivalent à un appel à `system`, il faut combiner `fork` et `exec`; exemple:

```
# méthode 1
system "date" ;
```

```
# méthode 2
unless(fork)
{  # fork a retourné 0,
   # je suis donc le processus fils et j'exécute:
   exec ("date") ; }
wait;       # le père attend que le fils meurt
```

`wait` permet de faire attendre le père jusqu'à ce que le fils meurt.

Enfin, l'opérateur `exit()` permet de sortir immédiatement du processus Perl courant. Cet opérateur peut prendre un paramètre optionnel qui est un entier indiquant la manière dont on est sorti. Par défaut, on sort avec la valeur 0, qui indique que tout s'est bien passé. Exemple:

```
unless(fork)
{ # je suis le processus fils
   unlink </tmp/bedrock.*> ;
   # destruction de tous ces fichiers
   exit();                  # le fils arrête de s'exécuter ici
}
# le père continue de s'exécuter ici
```

Sans le `exit`, le processus fils continue d'exécuter le programme Perl (à la ligne `'le père continue de s'exécuter ici'`).

2.13.5. Résumé des opérations sur les processus

Le tableau suivant résume les différentes façons de créer un processus avec Perl:

Opération	Entrée standard	Sortie standard	Erreur standard	Attendu?
system()	héritée du programme	héritée du programme	héritée du programme	oui
backquotes	héritée du programme	capturée en tant que chaîne	héritée du programme	oui
open() pour un filehandle en lecture	connectée au filehandle	héritée du programme	héritée du programme	au moment du close()
open() pour un filehandle en écriture	héritée du programme	connectée au filehandle	héritée du programme	au moment du close()
fork, exec, wait	choix de l'utilisateur	choix de l'utilisateur	choix de l'utilisateur	choix de l'utilisateur

La plus simple façon de créer un processus est d'appeler `system()`. `STDIN`, `STDOUT` et `STDERR` sont inchangées. Avec les backquotes, la sortie est capturée sous forme d'une chaîne de caractères; `STDIN` et `STDERR` sont inchangées. Avec ces deux méthodes, le programme Perl attend que le processus meurt pour continuer son exécution.

Pour une exécution en parallèle, l'ouverture d'une commande en tant que filehandle est une méthode simple.
La méthode la plus souple est d'utiliser les opérateurs `fork`, `wait` et `exec` qui sont directement reliés aux appels système du même nom. On peut alors choisir entre les exécutions séquentielle et parallèle.

2.13.6. Envoi et réception de signaux

Une méthode de communication interprocessus sous Unix est l'envoi et la réception de signaux. Un signal est un message de 1 bit envoyé à un processus par un autre processus ou par le noyau Unix.
La réponse à un signal s'appelle *action du signal*. Certains signaux ont pour action par défaut de tuer ou de suspendre le processus. D'autres sont complètement ignorés par défaut.

Quand un processus Perl capte un signal, une subroutine de votre choix est lancée automatiquement, interrompant temporairement ce qui était en train de s'exécuter. Lorsque la subroutine a été exécutée, le contexte est restitué, comme s'il ne s'était rien passé (sauf si la subroutine a exécuté des instructions).

En général, une telle subroutine fait l'une de ces deux choses:

- après avoir effectuer des instructions de nettoyage, elle arrête l'exécution du programme;
- elle positionne un flag (par exemple une variable globale) que le programme consulte régulièrement.

Les noms des signaux sont disponibles dans la page man de l'appel système *signal*. Pour associer une subroutine de traitement de signal à un signal, on utilise le tableau associatif `%SIG`. Ce tableau a pour clés les signaux. Pour associer la subroutine `&traite_sigint` au signal `SIGINT`, il suffit d'écrire:

```
$SIG{'INT'} = 'traite_sigint';

sub traite_sigint
{ $sigint_flag = 1 ;   # positionnement d'un flag
}
```

Cette subroutine positionne donc une variable globale et se termine tout de suite après. L'exécution du programme reprend là où elle s'était interrompue.

On peut également donner des valeurs spéciales aux signaux:

- 'DEFAULT': associe au signal son action par défaut;
- 'IGNORE': permet d'ignorer le signal.

Un signal peut être généré par l'utilisateur lorsqu'il frappe certains caractères. Un processus peut également générer un signal, grâce à l'opérateur **kill()**. Cet opérateur prend comme paramètres un nom ou un numéro de signal et les numéros des processus auxquels ce signal doit être envoyé. Exemple:

```
kill ( 2, 234, 237);
```

Le signal 2, `SIGINT`, est envoyé aux processus 234 et 237.

2.14. AUTRES TRANSFORMATIONS SUR LES DONNEES

2.14.1. Recherche d'une sous-chaîne

L'opérateur **index()** permet de retrouver une sous-chaîne dans une chaîne plus longue. Exemple:

```
$x = index($chaine, $ss_chaine);
```

Cet opérateur repère la première occurrence de la sous-chaîne dans la chaîne et renvoie un entier qui est la position du premier caractère de la sous-chaîne dans la chaîne; exemples:

```
$pos = index ("coucou", "co");              # $pos = 0
$pos = index ("coucou", "ouc");             # $pos = 1
$ss_chaine = "ba";
$pos = index ("coucou", $ss_chaine);
# $pos = -1 (on n'a pas trouvé "ba" dans "coucou")
```

Les opérandes d'`index()` peuvent être des chaînes, des variables scalaires contenant des chaînes ou des expressions ayant pour valeur une chaîne.
On peut donner un troisième paramètre à `index()` afin de trouver les autres occurrences de la sous-chaîne à l'intérieur de la chaîne. La recherche se fera à droite de la position indiquée par ce paramètre. Exemples:

```
$pos = index ("coucou", "co", 1);
 # $pos = 3, deuxième occurrence de "co"
$pos = index ("coucou","co",4);
# $pos=-1, "co" n'apparaît plus après la position 4
```

On peut effectuer la recherche en partant de la droite; pour cela il existe l'opérateur **rindex()**. La position retournée est toujours à compter à partir de la gauche, mais elle concerne la première occurrence de la sous-chaîne en partant de la droite. Exemples:

```
$pos = rindex ("coucou", "co");
# $pos=3, première occurrence de "co" en partant de la droite
$pos = rindex ("bateau", "ba");             # $pos = 0
$pos = rindex ("hello world", "o", 3);      # $pos = -1
$pos = rindex ("hello world", "o", 6);      # $pos = 4
```

Comme le montrent les deux derniers exemples, `rindex()` peut également prendre un troisième paramètre qui est la position à gauche de laquelle la recherche est effectuée.

2.14.2. Extraction et remplacement d'une sous-chaîne

Extraire une chaîne de caractères peut se faire grâce aux expressions régulières. On peut également utiliser l'opérateur **substr()**, qui a pour syntaxe:

```
$s = substr($chaine, $pos, $long);
```

`$s` reçoit la sous-chaîne extraite de `$chaine` à partir de la position `$pos` et de longueur `$long`; exemples:

```
$s = substr("hello world", 3, 4);          # $s =  "lo w"
$s = substr("hello world", 2, 100);        # $s =  "llo world"
$s = substr("hello world", -3, 3);         # $s =  "rld"
$s = substr("hello world", -3, 1);         # $s =  "r"
```

Une valeur négative de la position indique la position en partant de la fin du mot. **Attention**, le dernier caractère est accédé par la position -1, alors que le premier caractère est accédé par la position 0.
Si `$pos` est un nombre négatif plus grand en valeur absolue que la longueur de la chaîne, alors la recherche est effectuée à partir de la position 0. Si `$pos` est un nombre positif plus grand que la longueur de la chaîne, alors la chaîne vide "" est retournée. Si on n'indique pas de longueur, `substr()` retourne tout ce qui se trouve à droite de `$pos`, jusqu'à la fin de la chaîne.

Si le premier paramètre de `substr()` est une variable, alors `substr()` peut figurer à gauche d'une affectation; exemple:

```
$chaine = "hello cricri";
substr($chaine, 0, 5) = "howdy";
# $chaine = "howdy cricri"
```

Mais la sous-chaîne de remplacement n'a pas forcément la même longueur que celle qui disparaît. La chaîne finale est automatiquement allongée ou raccourcie; exemple:

```
$chaine = "hello cricri";
substr($chaine, 0, 5) = "hi";          # $chaine = "hi cricri"
substr($chaine, -6, 6) = "sebastien";
# $chaine = "hello sebastien"
```

2.14.3. Associer des formats aux données avec sprintf()

`printf()` permet d'afficher une liste de valeurs avec des formats. `sprintf()` fait la même chose, mais retourne le résultat dans une chaîne de caractères; exemple:

```
$var = sprintf("X%05d", $y);
```

2.14.4. Un tri performant

Nous avons déjà vu que l'opérateur `sort` appliqué à une liste permet de la trier suivant l'ordre ASCII croissant. Et si on veut réaliser d'autres tris, par exemple un tri numérique? Eh bien, en fait `sort` est un opérateur général; il suffit de définir une subroutine de comparaison de deux éléments.
Cette subroutine de comparaison est définie comme une subroutine ordinaire et est appelée de façon répétitive, jusqu'à ce que la liste soit triée.

Pour gagner en vitesse d'exécution, les deux éléments comparés ne sont pas passés à la subroutine sous forme de tableau, mais en tant que valeurs des variables globales `$a` et `$b`. La subroutine retourne:

- un nombre négatif si `$a` «est inférieure à» `$b`
- 0 si `$a` «égale» `$b`
- un nombre positif si `$a` «est supérieure à» `$b`.

inférieur, *supérieur* et *égal* ayant le sens qu'implique la comparaison effectuée. La subroutine suivante permet d'effectuer un tri numérique:

```
sub by_number
{ if ($a < $b)
   { -1; }
   elsif ($a > $b)
   { 1; }
   elsif ($a == $b)
   { 0; }
}
```

Comment l'utiliser? Prenons par exemple la liste suivante:

```
@liste = ( 1, 2, 4, 8, 16, 32, 48, 64, 128, 256);
```

Si on utilise sort comme on le fait d'habitude, les éléments de la liste `@liste` seront ordonnés comme des chaînes de caractères:

```
@mauvaise_liste = sort @liste;
# @mauvaise_liste = (1, 128, 2, 256, 32, 4, 48, 64, 8)
```

Pour un tri numérique, il faut utiliser notre subroutine de comparaison comme suit:

```
@bonne_liste = sort by_number @mauvaise_liste;
# @bonne_liste = (1, 2, 4, 8, 16, 32, 48, 64, 128, 256)
```

Remarquez que le & n'est pas obligatoire: Perl reconnaît que la chaîne entre `sort` et la liste est la subroutine de comparaison.

Ce genre de comparaison renvoyant -1, 0 ou 1 est si fréquent que Perl dispose de l'opérateur *vaisseau_spatial*, noté **<=>**. Si on l'utilise, notre subroutine s'écrit alors tout simplement:

```
sub by_number
{ $a <=> $b ; }
```

On peut aller encore plus loin en remplaçant le nom de la subroutine par l'instruction qu'elle exécute:

```
@bonne_liste = sort { $a <=> $b } @mauvaise_liste;
```

<=>, opérateur pour la comparaison numérique, a pour équivalent **cmp** pour la comparaison des chaînes de caractères. Une façon de trier une liste par ordre ASCII croissant (sort classique) est d'écrire:

```
@resultat = sort { $a cmp $b } @liste;
```

Ce nouvel opérateur peut sembler inutile à première vue. Pourtant il permet de garantir que la liste triée aura toujours les mêmes éléments à la même place. En effet, si des éléments de la liste sont égaux, alors on n'obtiendra pas forcément la même liste après chaque tri.
Supposons par exemple vouloir obtenir une liste des logins et des vrais noms des utilisateurs ordonnée par les noms d'utilisateur. Si ces valeurs se trouvent dans le tableau associatif `%nom` indexé par les logins, il suffit d'écrire:

```
@cles_triees = sort by_names keys(%nom);

sub by_names
{ ($nom{$a} cmp $nom{$b}) || ($a cmp $b) ; }

foreach @cles_triees
{ print "$_ a pour vrai nom $nom{$_} \n" ; }
```

Si deux noms d'utilisateurs sont identiques, on compare leurs logins, ce qui évite d'avoir à l'affichage un coup
```
"grossior a pour vrai nom grossiord"
"seb a pour vrai nom grossiord"
```
et une autre fois
```
"seb a pour vrai nom grossiord"
"grossior a pour vrai nom grossiord"
```

2.14.5. Translitération

Si l'on veut remplacer un caractère d'une chaîne par un autre ou supprimer toute occurrence d'un caractère, on peut utiliser l'opérateur `s///`. Mais si l'on veut remplacer tous les `a` par des `b` et tous les `b` par des `a`? Deux appels successifs à `s///` ne feront pas ce qu'on attend puisque le deuxième annulera l'effet du premier.

Perl dispose donc de l'opérateur **`tr///`** qui est très proche de la commande shell *tr*. Exemple:

```
tr /ab/ba/;
```

Cet opérateur prend deux arguments: la vieille chaîne et la nouvelle chaîne. Ils agissent comme les deux arguments de `s///`. `tr///` modifie le contenu de la variable `$_`; il cherche les caractères de la vieille chaîne et les remplace par ceux de la nouvelle chaîne. Exemples:

```
$_ = "coucou! ca va?" ;
tr /cau/fid/ ;          # $_ contient maintenant "fodfod! fi vi?"
tr /a-c/A-C/ ;          # $_ contient maintenant "CouCou! CA vA?"
```

Si la nouvelle chaîne est plus courte que la vieille, le dernier caractère de la nouvelle chaîne est répété autant de fois que nécessaire pour que la chaîne où s'effectue la recherche conserve la même longueur; exemple:

```
$_ = "coucou! ca va?" ;
tr /a-z/x/ ;            # $_ contient maintenant "xxxxxx! xx xx?"
```

Pour éviter cela, il suffit d'ajouter le caractère **d** à la fin de l'appel à `tr///`; exemple:

```
tr /a-z/A-C/d ;         # $_ contient maintenant "CC! CA A?"
```

Tout caractère de la vieille chaîne auquel ne correspond pas une valeur dans la nouvelle chaîne est supprimé.

Si la nouvelle liste est vide et qu'il n'y pas d'option `d`, la nouvelle liste est alors la même que la vieille liste. Cela peut paraître idiot. Cependant `tr///` retourne alors le nombre de caractères matchés par la vieille chaîne.

Exemples:

```
$_ = "coucou! ca va?" ;
$nb = tr /a-z// ;           # $_ n'a pas changé mais $nb = 10
$nb = tr /a-z/A-Z/ ;        # $_ est en majuscule et $nb = 10
```

Si on place un **c** en fin d'appel à `tr///`, on prend comme vieille chaîne le complémentaire de la vieille chaîne dans l'ensemble des 256 caractères alphanumériques; exemples:

```
$_ = "coucou! ca va?" ;
$nb = tr /a-z//c;           # $_ n'a pas changé mais $nb = 3
$nb = tr /a-z/_/c ;         # $_ = "coucou__ca_va_" et $nb = 3
tr /a-z//cd ;
# $_ contient maintenant "coucoucava"
```

Les options peuvent être combinées, comme le montre ce dernier exemple.

La dernière option de `tr///` est **s**, qui a pour effet de substituer les copies successives d'une même lettre de la nouvelle chaîne par une unique copie; un exemple est plus parlant!

```
$_ = "aaabbbcccdefghi";
tr /defghi/abcddd/s ;       #     $_     contient     maintenant
"aaabbbcccabcd"
$_ = "cricri et seb, hugo et xav";
tr /a-z/X/s ;               # $_ contient maintenant "X X X, X X
X"
$_ = "cricri et seb, hugo et xav";
tr /a-z/_/cs ;              #     $_     contient     maintenant
"cricri_seb_hugo_xav"
```

Comme `s///`, `tr///` peut être appliqué à une autre chaîne que `$_` en utilisant l'opérateur **=~**; exemple:

```
$chaine = "cricri et seb, hugo et xav";
$chaine =~ tr /a-z/_/cs ;
# $chaine contient maintenant "cricri_seb_hugo_xav"
```

2.15. ACCES AUX DONNEES SYSTEME

2.15.1. Pour récupérer des informations sur les mots de passe et les groupes

Les informations que possède le système sur votre nom d'utilisateur et votre UID sont publiques. En fait, pratiquement tout sauf votre mot de passe est disponible. Il suffit d'accéder le fichier */etc/passwd*. Ce fichier a un format particulier:

```
name:passwd:uid:gid:gcos:dir:shell
```

Les champs sont définis comme suit:

name	le login de l'utilisateur
passwd	le mot de passe crypté de l'utilisateur
uid	l'identificateur d'utilisateur (0 pour le root, un nombre non nul pour les autres utilisateurs)
gid	l'identificateur de groupe par défaut
gcos	le champ GCOS, qui contient en général le nom de l'utilisateur, une virgule et d'autres informations
dir	le répertoire racine de l'utilisateur
shell	le shell lancé au moment du login

Perl dispose de l'opérateur **getpwnam()**, qui prend comme unique argument un nom d'utilisateur et retourne la ligne du fichier */etc/passwd* correspondante, décomposée dans un tableau:

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

`$quota` est toujours vide; `$comment` et `$gcos` contiennent tous les deux le champ GCOS.

Si on utilise l'opérateur **getpwuid()** en lui passant le UID de l'utilisateur, on obtient le même tableau.

`getpwuid()` et `getpwnam()` sont des opérateurs permettant d'accéder à une ligne de */etc/passwd* grâce à une clé: le UID ou le nom de l'utilisateur. Si on désire maintenant un accès séquentiel à */etc/passwd*, Perl propose les opérateurs **setpwent()**, **getpwent()** et **endpwent()**. Ces opérateurs passent en revue toutes les lignes de */etc/passwd*. `setpwent()` initialise le parcours au début du fichier. Chaque appel à `getpwent()` retourne la ligne courante du

fichier; lorsqu'il n'y a plus de ligne à lire, il retourne la liste vide. Un appel à `endpwent()` permet de libérer les ressources utilisées pour scanner */etc/passwd*. Exemple:

```
setpwent;   # initialise le parcours
while (@liste = getpwent )
{ # récupère la ligne courante
   ($login, $home) = @liste[0, 7];     #  création  d'un  tableau
associatif dont la clé est le login
   print "Le répertoire racine de $login est $home";
}
endpwent;   # libération des ressources
```

En général, on utilise `getpwuid()` et `getpwnam()` pour accéder à un petit nombre d'informations. Si on souhaite accéder à plus d'informations, il vaut mieux utiliser `setpwent()`, `getpwent()` et `endpwent()`.

Pour accéder le fichier */etc/group*, il existe de même les opérateurs **setgrent()**, **getgrent()** et **endgrent()** pour un accès séquentiel, et **getgrgid()** et **getgrnam()** pour un accès par le GID ou le nom du groupe.
Un appel à `getgrent()` retourne le tableau:

```
($name, $passwd, $gid, $members)
```

Ces quatre valeurs correspondent exactement aux quatre champs du fichier */etc/group*.

2.15.2. Manipulation de données binaires

L'opérateur Perl **pack()** fonctionne un peu comme `sprintf()`. Il prend comme paramètre une chaîne de contrôle de formats et une liste de valeurs, ne faisant plus qu'une seule chaîne de cette liste. Cependant, il est orienté vers la création de structures de données binaires. Exemple:

```
pack ("CCCC", 140, 186, 65, 25);
```

Une chaîne est créée, qui contient quatre octets dont les valeurs sont celles des quatre entiers.
Le format **C** permet de transformer un petit entier (une valeur de caractère non signée) en un octet.
Le format **l** permet de générer une valeur *long* signée. Il dépend cependant de la machine; en général, c'est un nombre codé sur quatre octets. Exemple:

```
$buf1 = pack("l", 0x41424344);
$buf2 = pack("ll", 0x41424344, 0x45464748);
```

`$buf1` contient `ABCD` ou `DCBA` (cela dépend de la machine), et `$buf2` contient `ABCDEFGH` ou `DCBAHGFE`.

La liste des différents formats pour `pack()` est disponible dans la page man de Perl.

Pour réaliser l'opération contraire de l'opération précédente, on utilise l'opérateur **unpack()**, qui prend comme paramètres une chaîne de format et une chaîne de données, et retourne une liste de valeurs, images mémoire de la chaîne de données. Exemple:

```
($val1, $val2) = unpack("ll", "ABCDEFGH");
```

On obtient `$val1=0x61626364` ou `0x64636261`, `$val2=0x65666768` ou `0x68676665`. Les espaces ne sont pas pris en compte dans la chaîne de format. Un nombre dans cette chaîne permet en général de répéter le format précédent autant de fois; exemples:

```
pack ("C4", 140, 186, 65, 25);
# équivaut à pack ("CCCC", 140, 186, 65, 25);
pack ("C2C2", 140, 186, 65, 25); # idem
```

Un format peut également être suivi d'une *, ce qui permet de l'appliquer au reste de la liste ou au reste de l'image mémoire. Exemple:

```
pack ("C*", 140, 186, 65, 25);
# équivaut à pack ("CCCC", 140, 186, 65, 25);
@valeurs = unpack("C*", "bonjour tout le monde! \n");
```

Ce dernier exemple génère une liste de 14 éléments (nombres), chacun étant associé à un caractère de la chaîne.

2.15.3. Pour récupérer des informations sur le réseau

Perl permet la programmation réseau d'une façon très proche de celle du C. En fait, la plupart des routines Perl permettant d'accéder au réseau ont les mêmes noms et paramètres que leurs équivalents en C.

En général, on a besoin de trouver l'adresse d'une machine du réseau dont on connaît le nom, et vice versa. La routine Perl **gethostbyname()** permet de trouver l'adresse à partir du nom; exemple:

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

Le paramètre de cette routine est donc un *hostname*, par exemple *slate.bedrock.com*. Le résultat est une liste dont le nombre d'éléments dépend du nombre d'adresses associées au nom. Si le hostname n'est pas valide, la liste vide est retournée.

L'information importante est le tableau `@addrs`; chacun de ses éléments est une adresse IP, passée à Perl en tant que chaîne de 4 caractères. Si on souhaite afficher le résultat, il faudra convertir cette chaîne avec `unpack()`. Exemple:

```
($addr) = (gethostbyname("slate.bedrock.com"))[4];
print "L'adresse de Slate est ", join(".", unpack("C4", $addr)),
"\n";
```

`unpack()` prend 4 caractères et retourne 4 nombres, qui, recollés, constitue une adresse IP comme nous avons l'habitude de les voir.

2.15.4. Pour récupérer d'autres informations

On peut également récupérer des informations sur une machine hôte à partir de son adresse, et des informations sur le réseau à partir de son adresse et de son nom. Il existe aussi des routines permettant d'accéder aux protocoles et à la liste des services du réseau. Elles fonctionnent comme celles du C.
Voir le guide de référence de Perl en annexe.

2.15.5. Un exemple de programmation réseau avec Perl

- Le modèle socket

1. Le processus serveur crée une socket générique avec `socket()`.

2. Le serveur lie la socket à une adresse convenue avec `bind()`.
   Le serveur fait savoir au système que tout est prêt pour l'établissement de connexions avec `listen()`.

3. Le serveur attend la première connexion avec `accept()`.

4. Le processus client crée une socket générique avec `socket()`.

5. Le client lie la socket à une adresse sélectionnée par le système avec `bind()`.

6. Une fois lié, le client connecte sa socket à celle du serveur avec `connect()`, en utilisant l'adresse convenue. Cela établie la connexion.

7. Le serveur est averti de cette nouvelle connexion. En général, il fait alors appel à `fork` pour créer un processus fils qui sera en charge de cette nouvelle connexion, le père restant en attente de la prochaine connexion.

8. Le processus fils lit les données à partir de la socket envoyée par le client; il y écrit également des données qui peuvent ensuite être lues par le client. Ces E/S sont traitées comme s'il s'agissait de filehandles.

9. Lorsque le fils et le client ont terminé leur communication, ils ferment le filehandle, fermant ainsi le connexion.

- Un exemple de client

Ce client est connecté à une adresse précise (le standard «daytime») sur une machine hôte précise (local host) et affiche tout ce que le port génère en sortie:

```
require 'sys/socket.ph';
$sockaddr = 'S n a4 x8';
chop($hostname = `hostname`);
($name, $aliases, $proto) = getprotobyname('tcp');
($name, $aliases, $port) = getservbyname('daytime', 'tcp');
($name, $aliases, $type, $len, $thisaddr) =
                                  gethostbyname($hostname);
$thisport = pack($sockaddr, &AF_INET, 0, $thisaddr);
$thatport = pack($sockaddr, &AF_INET, $port, $thisaddr);

socket(S, &PF_INET, &SOCK_STREAM, $proto) ||
               die("Impossible de créer la socket. ");
bind(S, $thisport) || die("Impossible de lier la socket. ");
connect(S, $thatport) || die("Connexion impossible. ");
while (<S>)
{ print; }
exit 0;
```

- Un exemple de serveur

Ce serveur crée une socket à l'adresse 4242 sur la machine courante. Celui qui se connecte sur ce port reçoit un *gâteau de la chance*. Chaque gâteau n'est utilisé qu'une fois (de façon aléatoire) jusqu'à ce que tous les gâteaux soient mangés. La base de gâteaux est alors réamorcée.
Ce serveur peut être utilisé avec le client ci-dessus à condition de remplacer la ligne d'affectation de `$port` par l'instruction `$port = 4242`. On peut aussi écrire `telnet localhost 4242`, pour observer la sortie.

```
require 'sys/socket.ph';
$sockaddr = 'S n a4 x8';
chop($hostname = `hostname`);
($name, $aliases, $proto) = getprotobyname('tcp');
$port = 4242;
$thisport = pack($sockaddr, &AF_INET, $port, "\0\0\0\0");

socket(S, &PF_INET, &SOCK_STREAM, $proto) ||
                    die("Impossible de créer la socket. ");
bind(S, $thisport) || die("Impossible de lier la socket. ");
listen(S, 5) || die("Impossible d'ecouter. ");
for ( ; ; )
{ accept(NS, S) || die("Impossible d'accepter le socket. ");
   print NS &fortune;
   close NS;  }

sub fortune
{ @fortunes = split(/\n%%\n/, <<'END') unless @fortunes;
A fool and his money are soon parted.
%%
A penny save is a penny earned.
%%
Ask not what your country can do for you; ask what you can do
for your country.
%%
END
splice (@fortunes, int(rand(@fortunes)), 1)."\n";  }
```

2.16. MANIPULATION DE BASES DE DONNEES UTILISATEUR

2.16.1. Les données et tableaux DBM

La plupart des systèmes Unix ont une bibliothèque standard appelée **DBM**. Cette bibliothèque dispose d'un outil de gestion de base de données qui permet aux programmes de stocker des paires (clé, valeur) dans une paire de fichiers du disque. Ces fichiers contiennent les valeurs de la base de données entre deux appels aux programmes manipulant cette base. Les programmes peuvent créer de nouvelles données, détruire ou modifier des données déjà existantes.

Perl permet d'accéder à cet outil de DBM. Un tableau associatif est associé à une base de données de DBM de la même manière qu'on ouvre un fichier. Ce tableau associatif (tableau DBM) est ensuite utilisé pour accéder et modifier la base de données DBM.

La taille, le nombre et la nature des clés et valeurs de la base de données DBM sont limitées; on a donc les mêmes limitations pour le tableau DBM. En général, si on ne dépasse pas 1000 caractères binaires pour les clés et les valeurs, ca passe.

2.16.2. Ouverture et fermeture des tableaux DBM

Pour associer une base de données DBM à un tableau DBM, on utilise l'opérateur **dbmopen()**, qui a pour syntaxe:

```
dbmopen(%NOM_DE_TABLEAU, "nom_du_fichier_dbm", $mode);
```

`%NOM_DE_TABLEAU` est un tableau associatif Perl. S'il contient déjà des valeurs, elles sont écrasées. Ce tableau est connecté au fichier DBM *nom_du_fichier_dbm*, généralement stocké sur le disque en tant que paire de fichiers: *nom_du_fichier_dbm*.**dir** et *nom_du_fichier_dbm*.**pag** . N'importe quel nom est accepté pour le tableau, mais en général on utilise des majuscules pour marquer la ressemblance avec les filehandles.

Le paramètre `$mode` est un nombre contrôlant les droits de la paire de fichiers si ces fichiers doivent être créés. Ce nombre est spécifié en octal: la valeur 0644 donne l'accès à tout le

monde en lecture uniquement; seul le propriétaire est autorisé en lecture et en écriture. Si les fichiers existent déjà, ce paramètre n'a pas d'effet. Exemple:

```
dbmopen(%FRED, "mabasededonnees", 0644);
```

Le tableau `%FRED` est associé aux fichiers *mabasededonnees.dir* et *mabasededonnees.pag* dans le répertoire courant. Si ces fichiers n'existaient pas, ils sont créés avec le mode 0644. `dbmopen()` retourne *vrai* si la base de données a pu être ouverte ou créée, *faux* sinon. Si on ne veut pas que les fichiers soient créés, il suffit de donner au mode la valeur *undef*; exemple:

```
dbmopen(%A, "/etc/xx", undef) ||
                  die("Impossible d'ouvrir la DBM /etc/xx. \n");
```

Si les fichiers */etc/xx.dir* et */etc/xx.pag* n'ont pu être ouverts, ils ne sont pas créés et un message est renvoyé à l'utilisateur.

Le tableau DBM reste ouvert pendant toute la durée d'exécution du programme. Quand celui-ci s'achève, l'association se termine aussi. On peut également mettre fin à l'association avec **dbmclose()**; exemple:

```
dbmclose(%FRED);
```

Cet opérateur retourne *faux* si quelque chose s'est mal passé.

2.16.3. Utilisation d'un tableau DBM

Une fois que la base de données est ouverte, toute référence au tableau DBM est interprétée comme une référence à la base de données. Toute modification, destruction ou ajout d'une valeur du tableau engendre les entrées correspondantes dans les fichiers de la base de données. Exemple:

```
dbmopen(%FRED, "mabasededonnees", 0644);
%FRED{"fred"} = "bedrock";  # crée (ou modifie) un élément
delete $FRED{"barney"};          # détruit un élément de la base
de données
foreach $key (keys %FRED)
{ print "$key a pour valeur $FRED{$key} \n"; }
```

Avec l'opérateur `foreach`, on parcourt deux fois le fichier de la base de données: une fois pour les clés et une seconde fois pour les valeurs. Il vaut mieux ici utiliser l'opérateur `each` comme suit:

```
while ( ($key, $value) = each(%FRED) )
{ print "$key a pour valeur $value \n"; }
```

2.16.4. Bases de données de longueur fixe et d'accès aléatoire.

Une autre forme persistante de données est le fichier disque orienté enregistrements de longueur fixe. Chaque enregistrement est composé des mêmes champs de longueur fixe; tous les enregistrements ont donc la même longueur.

Par exemple, soit le fichier `SPEINF` contenant les enregistrements pour les années spéciales info 95; chaque enregistrement possède trois champs:

- 40 octets pour le nom
- 40 octets pour le prénom
- 2 octets pour l'age

Un enregistrement a donc une longueur fixe de 82 octets.

Perl permet de manipuler de tels fichiers disques. Cependant il faut savoir:

1. ouvrir un fichier disque en lecture et en écriture
2. se déplacer dans ce fichier
3. récupérer des données en connaissant la longueur d'une ligne plutôt qu'en cherchant le caractère de fin de ligne
4. écrire des données sous forme de blocs de taille fixe

L'opérateur `open()` prend un signe + devant ses spécifications d'E/S pour indiquer que le fichier est ouvert en lecture et en écriture; exemple:

```
open(A, " + < b");
# le fichier b est ouvert en lecture et en écriture
open(C, " + > d");
# le fichier d est créé avec les droits en lecture et en
écriture
open(E, " + >> f");
# le fichier f est ouvert ou créé avec les droits en lecture et
en écriture
```

Une fois qu'un fichier est ouvert, il faut pouvoir s'y déplacer. On utilise pour cela l'opérateur **seek()**, qui prend trois paramètres. Le premier est un filehandle; le second est un offset, interprété en conjonction avec le troisième paramètre. En général, on donnera la valeur 0 à celui-ci, de façon à ce le deuxième paramètre soit interprété comme une nouvelle position absolue pour la prochaine lecture ou écriture; exemple:

```
seek(SPEINF, 4*82, 0);
```

Le pointeur est positionné au début de la cinquième ligne du fichier `SPEINF` (la taille des enregistrements contenus dans ce fichier est 82 octets). C'est sur cette ligne que s'opéreront les prochaines entrées et sorties.

Pour écrire dans le fichier de la base de données, l'opérateur `print()` est utilisé, mais il faut être sûr de la longueur de la donnée écrite. Pour obtenir cette longueur, on fait appel à l'opérateur `pack()`. Exemple:

```
print SPEINF pack("A40A40s", $nom, $prenom, $age );
```

`pack()` permet de spécifier que, dans un enregistrement du fichier `SPEINF`, les premier et deuxième champs ont pour longueur 40 octets et le troisième est un *short* (2 octets). `$nom`, `$prenom` et `$age` doivent vérifier cela.

Pour aller chercher un enregistrement particulier dans le fichier, on utilise l'opérateur **read()**. Exemple:

```
$nb = read(SPEINF, $buf, 82);
```

Le premier paramètre est un filehandle, le deuxième est une variable scalaire qui contiendra les données qui seront lues, le troisième est le nombre d'octets à lire. `read()` retourne le nombre d'octets effectivement lus.
L'opérateur `unpack()` permet de découper la chaîne de 82 octets qui vient d'être lue; exemple:

```
($nom, $prenom, $age) = unpack("A40A40s", $buf);
```

Remarquez que les formats de `pack()` et `unpack()` sont les mêmes. En général cette chaîne est stockée en début de programme puisqu'elle est caractéristique du fichier étudié.

2.16.5. Bases de données de longueur variable

La plupart des bases de données système et utilisateur sont une série de lignes de texte compréhensible, avec un enregistrement par ligne. En général, elles sont modifiées directement avec un éditeur de texte.

Perl permet une telle mise à jour des bases de données constituées de lignes grâce à l'**édition en ligne**. Cette méthode consiste en une modification de la façon dont l'opérateur diamant <> lit les données des fichiers spécifiés dans la ligne de commande. En général, ce mode d'édition est accédé en positionnant l'option **-i** de la ligne de commande.

Avant de lancer l'*édition en ligne*, il faut donner une valeur à la variable scalaire **`$^I`**. Cette variable est importante et nous verrons plus bas en quoi elle consiste.

Lorsque l'opérateur <> est utilisé et que `$^I` a une valeur (différente de *undef*), les lignes marquées par `##INPLACE##` dans le code suivant sont ajoutées à la liste d'actions implicites de l'opérateur <>:

```
$ARGV = shift @ARGV;
open(ARGV, "<$ARGV");
rename($ARGV, "$ARGV$^I"); ## INPLACE##
unlink($ARGV); ## INPLACE##
open(ARGVOUT, "> $ARGV"); ## INPLACE##
select(ARGVOUT); ## INPLACE##
```

Les lectures effectuées par <> sont faites à partir du vieux fichier, les écritures sont faites sur une nouvelle copie du fichier. Le vieux fichier reste dans le fichier de backup, dont le nom est le nom du fichier suivi de l'extension contenue dans la variable `$^I`. Ceci est répété pour tous les fichiers de `@ARGV`.

En général, `$^I` a pour valeur **`.bak`** ou **`~`**. Si `$^I` est la chaîne vide "", le vieux fichier est éliminé. Cependant, il faut être prudent: si un incident a lieu pendant l'exécution du programme, vous perdez alors toutes vos vieilles données.

Voici une façon de changer le shell de lancement pour tous les utilisateurs du fichier */etc/passwd*:

```
@ARGV = ("/etc/passwd");    # affectation de l'opérateur diamant
$^I = ".bak";
# écrit /etc/passwd.bak pour plus de sécurité
while (<>)
{ # pour chaque ligne de /etc/passwd
   s#:[^:]*$#:/bin/sh#;
   # change  shell de lancement en /bin/sh
   print;
   # envoi la sortie sur ARGVOUT: le nouveau /etc/passwd
}
```

Ce programme est simple; pourtant on peut faire la même chose avec une unique ligne de commande:

```
> perl -p -i.bak -e 's#:[^:]*$#/bin/sh#' /etc/passwd
```

-p permet d'encadrer le programme avec une boucle `while` incluant une instruction `print`.

-i permet de fixer la valeur de `$^I`.

-e indique que l'argument qui suit est un morceau de code Perl à exécuter dans le corps de la boucle `while`.

Le dernier paramètre donne une valeur initiale à `@ARGV`.

2.17. CONVERTIR D'AUTRES LANGAGES EN PERL

2.17.1. Conversion de programmes awk en Perl

Si vous avez un programme *awk* que vous voulez faire tourner avec Perl, vous pouvez effectuer une traduction mécanique avec l'outil **a2p** fourni avec Perl. Cet outil convertit la syntaxe awk en syntaxe Perl et permet, pour la majorité des programmes *awk*, d'obtenir un script Perl prêt à l'emploi.

Pour utiliser `a2p`, mettre le programme *awk* dans un fichier séparé et appeler *a2p* avec le nom du fichier comme argument ou avec une redirection en entrée avec ce fichier. La sortie est un programme Perl exécutable. Exemple:

```
> a2p < mon_pg_awk   > mon_pg_perl
> perl mon_pg_perl <arguments>
```

Un programme *awk* converti en Perl aura généralement la même fonction, mais s'exécutera plus rapidement. Le code du programme Perl obtenu peut parfois être optimisé.

Certaines conversions ne sont pas mécaniques. Par exemple, *awk* utilise l'opérateur **>** pour comparer des chaînes ou des nombres, alors que Perl utilise **>** avec les nombres et **gt** avec les chaînes. `a2p` essaie de deviner quel opérateur Perl convient en fonction de ce qu'il sait des opérandes. S'il n'est pas sûr de son choix, `a2p` marque la ligne avec **#??** (commentaire Perl) et une explication. Il vaut donc mieux consulter le script Perl obtenu après conversion afin de corriger ce genre d'erreur.

Pour en savoir plus sur `a2p`, consulter sa page man.

2.17.2. Conversion de programmes sed en Perl

L'opérateur **s2p** convertit un programme *sed* en programme Perl. Il s'utilise de la même façon que `a2p`.
Cependant il n'y a en général pas d'erreur de conversion.
`s2p` est écrit en Perl!

2.17.3. Conversion de programmes shell en Perl

Il n'y a malheureusement pas de convertisseur shell/Perl. Le problème est que, dans un shell script, la plupart des commandes ne sont pas exécutées par le shell; des appels sont effectués à des programmes indépendants pour récupérer des morceaux de chaînes, comparer des nombres, concaténer des fichiers...

Donc, c'est à vous de jouer au convertisseur: étudiez le script shell, comprenez-le et écrivez un script Perl réalisant la même chose! L'opérateur `system()` peut vous aider à traduire rapidement.

2.18. COMPLEMENTS

2.18.1. L'opérateur require

C'est l'équivalent du *include* du C. Des programmes Perl peuvent être stockés dans un ou plusieurs fichiers indépendants et être ensuite inclus dans tout programme Perl qui a besoin d'eux; exemple:

```
require ' abbrev.pl';
```

permet d'inclure le texte du fichier `abbrev.pl` comme s'il faisait partie du fichier dans lequel figure cette instruction. Du code peut ainsi être partagé entre différents programmes Perl.

Cependant, les subroutines incluses doivent retourner une valeur **non nulle** pour que `require` fonctionne. Par exemple, toutes les routines de la bibliothèque se terminent par la ligne:

```
1;
```

2.18.2. La bibliothèque Perl

Elle contient des programmes Perl utiles, évitant de devoir les réécrire soi-même. Pour savoir dans quel répertoire elle se trouve, il suffit d'imprimer le premier élément du tableau `@INC`. La plupart des routines sont commentées. Pour les utiliser dans un programme, il suffit d'utiliser l'opérateur `require` ci-dessus en début de programme.

2.18.3. Les Here Strings

Après les chaînes entre simples ou doubles quotes, voici les *here strings* (même chose que les *here documents* en shell):

```
$a = <<"@HEAD"."\n".$body;
To: merlyn@ora.com
From: $username
Subject: What, do you think?
Date: $now
HEAD
```

Elles ne sont vraiment pas nécessaires.

2.18.4. Les alias

On peut faire de `b` un alias de `a` en écrivant `*b = *a`. Cela signifie que `$a` et `$b` référencent la même variable, tout comme `@a` et `@b`, ou même les filehandles et les formats `a` et `b`. `*b` peut également être utilisé localement avec l'instruction `local(*b)`.

2.18.5. L'opérateur eval() et s///e

C'est l'équivalent de l'opérateur du même nom en shell. Il permet d'évaluer du bout de code passé en paramètre lors de l'exécution du programme Perl. Un exemple, pour être plus clair:

```
print "Donnez la commande à exécuter : ";
chop($code = <STDIN>);
eval $code;
die("eval : $@") if $@;     # on teste si eval s'est bien passé
```

Ce script permet d'exécuter une commande passée par l'utilisateur pendant l'exécution comme si elle faisait partie du programme.

On peut mettre du code Perl dans la chaîne de remplacement de l'opérateur de substitution si celui-ci est suivi du flag **e**. Cela permet d'avoir des chaînes de remplacement compliquées, faisant par exemple appel à des subroutines retournant des résultats.

Exemple:

```
while(<>)
{  s/^(\S+)/$var+1/e;
    print;  }
```

Le premier mot de chaque ligne de l'entrée est remplacé par la valeur de `$var+1`.

2.18.6. Le debugger

Si l'on fait appel à Perl avec l'option **-d**, le script tourne sous le moniteur de deboggage. Pour connaître les commandes de deboggage, il suffit alors de taper **h**: cela donne accès à la page d'aide.
En tapant simplement:

```
> perl -de 0
```

le debugger démarre sans script.

2.18.7. L'opérateur , (virgule)

Cet opérateur évalue son opérande gauche, élimine le résultat de cette évaluation, puis évalue son opérande droit, retournant cette dernière évaluation comme résultat de l'expression (cela est pratique lorsque l'opérande gauche est à évaluer uniquement pour ses effets secondaires, comme l'incrémentation d'une variable ou l'appel à une subroutine). Exemple:

```
$a = ( 1, 3);
```

affecte `$a` avec la valeur `3`.

2.18.8. Les opérateurs sur les bits

Il existe les opérateurs de décalage de bit à gauche, **>>**, et à droite, **<<**. Le résultat dépend cependant des propriétés arithmétiques de la machine.

On dispose également des opérateurs logiques sur les bits:

- **&**, et logique
- |, ou logique
- ^, ou exclusif

Ces opérateurs travaillent sur des valeurs numériques entières.

2.18.9. Les paquetages

On désire parfois que les données d'un programme soient privées et inaccessibles à d'autres programmes. Perl dispose pour cela d'un mécanisme de paquetage, qui fournit un espace d'adressage indépendant. On peut alors avoir deux variables portant le même nom dans deux programmes différents: elles sont indépendantes. Le mot réservé **package** permet de déclaré des paquetages, n'importe où dans le programme. Une telle déclaration est effective de la ligne où elle est effectuée à la fin du bloc la contenant. Exemple:

```
package pecan;
sub tarte
{ $reponse = <STDIN>;  # une variable
  package canari;
  $reponse = <STDIN>;       # une autre variable
}
```

Il est possible de référencer les variables et les filehandles d'un paquetage à l'intérieur d'autres paquetages; il suffit de faire précéder leur nom du nom du paquetage et d'une quote simple. L'exemple précédent devient alors:

```
sub pecan'tarte
{ $pecan'reponse = <STDIN>;
  $canari'reponse = <STDIN>;      }
```

Si le nom du paquetage est nul, ce paquetage est considéré comme le paquetage principal ou **main package**. Alors `$'tarte` est la même chose que `$main'tarte`.

Seuls les identificateurs dont les noms commencent par une lettre sont stockés dans la table de symboles du paquetage. Les autres symboles (toutes les variables spéciales) et `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `AGVOUT`, `ENV`, `INC` et `SIG` sont conservés dans le paquetage principal.

La table de symboles de chaque paquetage est stockée dans un tableau associatif situé dans la table de symboles du paquetage principal. Le nom de la table de symboles du paquetage `bleu` n'est autre que `%_bleu`.

3. EXEMPLE D'APPLICATION

3.1. UN SYSTEME DE RESERVATION DE VOLS

Comme nous l'avons vu dans le chapitre 2, Perl a des domaines d'utilisation très variés:

- Manipulation de bases de données
- Administration du système
- Manipulation de fichiers, de textes, de processus
- Communication interprocessus ...

En voici un exemple concret: un système de réservation de vols.

Le fichier *place_disp* joue le rôle de base de données. Il contient des enregistrements de la forme:

```
code-vol:ville-départ:ville-arrivée:heure-départ:heure-
arrivée:date-départ:classe:nb-places-dispo:prix
```

Le fichier suivant est utilisé pour tester le programme:

```
AI132:Paris:Nice:10h00:11h20:22/06/95:eco:8:400
AF330:Lille:Toulouse:22h30:23h50:23/07/95:club:16:1030
BA120:Paris:Londres:15h40:17h00:21/08/95:prem:8:2500
AF562:Nice:Paris:21h00:22h20:21/07/95:eco:44:400
AF457:Nice:Paris:22h00:23h20:21/07/95:eco:20:400
IB120:Lisbonne:Lille:12h00:14h30:10/08/95:club:23:3000
KL498:Amsterdam:Nice:17h00:19h30:17/09/95:eco:60:1500
```

Une interface permet à l'utilisateur d'indiquer le vol qui l'intéresse. Le fichier *place_disp* est parcouru, à la recherche d'un vol qui correspond aux saisies de l'utilisateur.
Il est possible d'effectuer des réservations. Le fichier *reservation* reçoit alors les données.
Enfin, des factures peuvent être imprimées; pour cela, le fichier *reservation* est consulté.

3.2. LE PROGRAMME PERL

```
################## Formats ########################################

format ENTETE =
Vol @<<<<<<<<<< -> @<<<<<<<<<< le @<<<<<<<
$VD, $VA, $date


.


format FORMAT1 =
=================================================================
heure depart| heure arrivee| code vol| classe| nb places| prix
=================================================================
@||||     | @||||         | @||||   | @|||| | @||      | @<<<<
$HD, $HA, $codevol, $classe, $nb_place, $prix




.


format MENU =
*************************************************************
**      Bienvenue sur notre systeme de reservation!         **
*************************************************************

Nous vous offrons les services suivants :

     1.  renseignements/reservations
     2.  facture
     q.  quitter



.


format FORMAT3 =



Ville de depart: @<<<<<<<<<<
$VA
Ville d'arrivee: @<<<<<<<<<<
$VD
.
```

```
format FACTURE =
================================================================
                FACTURE
================================================================

     no reservation : @<<<<
$no_resa

     nom : @<<<<<<<<<<
$nom

@<<<<<<<   @<<<<<<<<<<   @<<<<
$date, $VD, $HD
           @<<<<<<<<<<   @<<<<
$VA, $HA

     code du vol : @<<<<
$codevol
     classe : @<<<<
$classe


                       somme a payer : @<<<<
$prix
                       somme versee  : @<<<<
$verse
                                  -----
                       somme due     : @<<<<
$prix-$verse


.



format ENREGISTRE =
@<<<<:@<<<<<<<<<<:@<<<<:@<<<<<<<<<<:@<<<<:@<<<<<<<<<<:@<<<<:@<<<<<<<:@<
<<<:@<<<<:@<<<<
$no_resa, $nom, $codevol, $VD, $HD, $VA, $HA, $date, $classe, $prix,
$verse
.



################## Programme principal ##########################

# initialisation des numeros de resa
$no_resa = 0;

#lancement
&menu_principal($no_resa);
```

```
##################  Subroutines ################################


sub menu_principal
{
  #### nettoyage de l'ecran
  system("clear");

  #### recuperation du numero de reservation
  $no_resa = $_[0];

  #### impression du menu
  $~ = "MENU";
  write;
  print "Votre choix: ";
  chop($choix=<STDIN>);
  print "\n";

  #### Traitement du choix de l'utilisateur
  if ($choix == 1)
  {
    #### nettoyage de l'ecran
    system("clear");

    #### obtenir des renseignements et eventuellement reserver

    #### Saisie des donnees et
    #### appel a la subroutine renseign
    &renseign(&saisie_cherch, $no_resa);
  }

  elsif ($choix == 2)
  {
    #### nettoyage de l'ecran
    system("clear");

    #### imprimer une facture
    &facture;
  }

  else
  {
    #### On sort du programme
    exit 0;
  }

}



sub saisie_cherch
{
    #### Saisie des villes de depart, d'arrivee et de la date de
depart
    print "Ville de depart: ";
    chop($VD = <STDIN>);
    print "\n";
```

```
    print "Ville d'arrivee: ";
    chop($VA = <STDIN>);
    print "\n";
    print "Date de depart: ";
    chop($date = <STDIN>);
    print "\n";

    #### valeur de retour
    @donnees = ($VD, $VA, $date);
}



sub reserv
{
  #### Saisie des donnees

  print "Nom du client: ";
  chop($nom = <STDIN>);
  print "\n";

  ($VD, $VA, $date, $o_resa, $codevol, $HD, $HA, $classe, $nb_place,
$prix) = @_;

  print "Accompte: ";
  chop($verse = <STDIN>);
  print "\n";

  #### mise a jour du numero de reservation
  $no_resa ++;


  # modification du nombre de places disponibles
  # ouverture en lecture du fichier place_disp
  open(A, "< place_disp");

  # ouverture en ecriture d'un fichier temporaire
  open(C, ">> temp_fic");

  # parcours du fichier place_disp
  while(<A>)
  {
    $sauv = $_;
    if ( $sauv =~ /^$codevol:$VD:$VA:$HD:$HA:$date:$classe:/i)
    {
      $sauv = $&;
      ($nb_place, $prix) = split(/:/, $');
      $nb_place --;
      $rempl = $sauv.$nb_place.":".$prix;
      s/^.*$/$rempl/;
    }
    print C;
  }
  close(A);
  close(C);
  unlink("place_disp");
  open(A, ">> place_disp");
  open(C, "< temp_fic");
```

```
  while(<C>)
  { print A unless (/^$/); }
  close(A);
  close(C);
  unlink("temp_fic");


  #### enregistrement dans le fichier reservation
  open(B, ">> reservation");
  select(B);
  $~ = "ENREGISTRE";
  write;
  select(STDOUT);

  print "\n\nReservation effectuee.\n\n\n";
  print "numero de resa = $no_resa\n\n\n";

  print "    Tapez q pour quitter\n          r pour menu
principal\n";
  chop($choix = <STDIN>);

  #### traitement du choix de l'utilisateur
  if ($choix eq 'r')
  {
    #### nettoyage de l'ecran
    system("clear");

    #### retour au menu pricipal
    &menu_principal($no_resa);
   }

  else
  {
    #### on sort du programme
    exit 0;
  }
}



sub renseign
{
  #### Initialisations
  $trouve = 0;
  @tab1 = ($VD, $VA, $date, $no_resa) = @_;

  #### ouverture en lecture du fichier place_disp
  open (A, "< place_disp");

  #### Consultation du fichier PLACE_DISP
  #### Recherche des renseignements
  BLOC:
  while (<A>)
  {
    #### Sauvegarde de $_
    $sauv = $_;
```

```
    #### Recherche des vols remplissant les conditions
    if (/$VD:$VA:.*:.*:$date/i)
    {
      $_ = $sauv;

      $~ = "ENTETE";
      write;

      #### Recuperation des informations
      @tab2 = ($codevol, $HD, $HA, $classe, $nb_place, $prix) =
/^(\S+):.*:.*:(\S+):(\S+):.*:(\S+):(\S+):(\S+)$/;

      #### un vol a ete trouve
      $trouve ++;

      #### Affichage des informations
      $~ = "FORMAT1";
      write;
      print "    Tapez q pour quitter \n          r pour reserver\n
s pour suivant\n          a pour A/R\n";
      chop($choix = <STDIN>);

      #### traitement du choix de l'utilisateur
      if ($choix eq 'q')
      {
     #### on sort du programme
        exit 0;
      }

      elsif ($choix eq 'r')
      {
        #### nettoyage de l'ecran
        system("clear");

     #### fermeture de place_disp
     close(A);

        #### on effectue une reservation
        &reserv(@tab1, @tab2);
      }

      elsif ($choix eq 'a')
      {
        #### nettoyage de l'ecran
        system("clear");

        #### on recherche un retour
        $~ = "FORMAT3";
     write;
     print "Date de retour: ";
     chop($date = <STDIN>);
     print "\n";

     #### Inversion des villes de depart et d'arrivee
     $sauv = $VA;
     $VA = $VD;
     $VD = $sauv;
```

```
      #### Reinitialisation de l'index
      $trouve = 0;

      #### Modification de @tab1
      @tab1 = ($VD, $VA, $date, $no_resa);

      #### Reinitialisation du pointeur sur la ligne courante de
           place_disp
      open(A,"< place_disp");

      #### on relance la recherche
      next BLOC;
       }
     }
  }

  if ($trouve == 0)
      {
      #### aucun vol n'a ete trouve correspondant aux exigences
      print "Pas d'avion pour ce trajet a cette date.\nVoulez-vous
saisir une autre date (o/n)? ";
      chop($rep = <STDIN>);
      print "\n";
      close(A);

      #### traitement du choix de l'utilisateur
      if ($rep=~/^o/)
      {
        #### nettoyage de l'ecran
           system("clear");

        #### saisie d'une nouvelle date
        print "Date de depart: ";
        chop($date = <STDIN>);
        print "\n";

        #### on relance la recherche
        &renseign($VD, $VA, $date, $no_resa);
       }

       else
       {
         #### nettoyage de l'ecran
            system("clear");

         #### retour au menu principal
         &menu_principal($no_resa);
       }
       }

   else
       {
      #### Il n'y a plus d'autres vols correspondant aux exigences
      print "\n\n\n\n";
      print "Pas d'autre vol.";
         close(A);
```

```
      print "\n\n\n    Tapez q pour quitter\n          a pour autre
recherche\n          r pour menu principal\n";
      chop($choix = <STDIN>);

      #### traitement du choix de l'utilisateur
      if ($choix eq 'q')
      {
        #### on quitte le programme
        exit 0;
         }

      elsif ($choix eq 'a')
      {
        #### nettoyage de l'ecran
           system("clear");

        #### une autre recherche est lancee
        &renseign(&saisie_cherch, $no_resa);
      }

      else
      {
        #### nettoyage de l'ecran
           system("clear");

        #### retour au menu principal
        &menu_principal($no_resa);
      }
       }
}



sub facture
{
   #### initialisation de l'index
   $trouve = 0;

   #### saisie du numero de resa
   print "Donnez le numero de reservation: ";
   chop($no_resa = <STDIN>);
   print "\n";

   #### recherche dans le fichier des reservations
   open(B, "< reservation");
   while (<B>)
   { $sauv = $_;
     if (/^$no_resa/)
     {
       ($no_resa, $nom, $codevol, $VD, $HD, $VA, $HA, $date,
$classe, $prix, $verse) = split(/:/, $sauv);
       $trouve ++;

       #### affichage
       $~ = "FACTURE";
       write;
     }
   }
```

```
   if ($trouve == 0)
   {
     #### aucune reservation n'a ete trouvee
     print "Aucune reservation ne correspond a ce numero.\n";
   }


   #### autre facture?
   print "    Tapez a pour autre facture\n          r pour menu
principal\n          q pour quitter\n";
   chop($choix = <STDIN>);

   #### traitement du choix de l'utilisateur
   if ($choix eq 'a')
   {
     #### nettoyage de l'ecran
     system("clear");

     #### impression d'une autre facture
     &facture;
   }

   elsif ($choix eq 'r')
   {
     #### nettoyage de l'ecran
     system("clear");

     #### retour au menu principal
     &menu_principal;
   }

   else
   {
     #### on quitte le programme
     exit 0;
   }
}
```

3.3. RESULTATS

```
************************************************************
**     Bienvenue sur notre systeme de reservation!          **
************************************************************

Nous vous offrons les services suivants :

        1.  renseignements/reservations
        2.  facture
        q.  quitter


Votre choix: 1


Ville de depart: paris

Ville d'arrivee: londres

Date de depart: 21/08/95

Vol paris      -> londres    le 21/08/95


============================================================
heure depart| heure arrivee| code vol| classe| nb places| prix
============================================================
15h40       | 17h00        | BA120   |  prem |  8       | 2500



    Tapez q pour quitter
          r pour reserver
          s pour suivant
          a pour A/R
s

Pas d'autre vol.


    Tapez q pour quitter
          a pour autre recherche
          r pour menu principal
a
```

```
Ville de depart: paris

Ville d'arrivee: nice

Date de depart: 22/06/95



Vol paris      -> nice       le 22/06/95

============================================================
heure depart| heure arrivee| code vol| classe| nb places| prix
============================================================
10h00       | 11h20        | AI132   |  eco  |  8       | 400



    Tapez q pour quitter
          r pour reserver
          s pour suivant
          a pour A/R
a



Ville de depart: nice

Ville d'arrivee: paris

Date de retour: 21/07/95


Vol nice       -> paris      le 21/07/95

============================================================
heure depart| heure arrivee| code vol| classe| nb places| prix
============================================================
21h00       | 22h20        | AF562   |  eco  |  45      | 400



    Tapez q pour quitter
          r pour reserver
          s pour suivant
          a pour A/R
s
```

```
Vol nice       -> paris      le 21/07/95


============================================================
heure depart| heure arrivee| code vol| classe| nb places| prix
============================================================
22h00       | 23h20        | AF457   |  eco  |  21      | 400

    Tapez q pour quitter
          r pour reserver
          s pour suivant
          a pour A/R
r



Nom du client: poiraud

Accompte: 100

numero de resa = 1



Reservation effectuee.


    Tapez q pour quitter
          r pour menu principal
r



************************************************************
**     Bienvenue sur notre systeme de reservation!        **
************************************************************

Nous vous offrons les services suivants :

        1.  renseignements/reservations
        2.  facture
        q.  quitter


Votre choix: 2
```

```
Donnez le numero de reservation: 1


============================================================
                        FACTURE
============================================================

        no reservation : 1

        nom : poiraud

21/07/95   nice         22h00
           paris        23h20

        code du vol : AF457
        classe : eco


                                somme a payer : 400
                                somme versee  : 100
                                                -----
                                somme due     : 300


    Tapez a pour autre facture
          r pour menu principal
          q pour quitter
r



**************************************************************
**     Bienvenue sur notre systeme de reservation!          **
**************************************************************

Nous vous offrons les services suivants :

        1.  renseignements/reservations
        2.  facture
        q.  quitter


Votre choix: 1


Ville de depart: lille

Ville d'arrivee: toulouse

Date de depart: 22/07/95

Pas d'avion pour ce trajet a cette date.
Voulez-vous saisir une autre date (o/n)? o
```

```
Date de depart: 23/07/95

Vol lille      -> toulouse   le 23/07/95


============================================================
heure depart| heure arrivee| code vol| classe| nb places| prix
============================================================
22h30       | 23h50        | AF330   |  club |  17      | 1030



    Tapez q pour quitter
          r pour reserver
          s pour suivant
          a pour A/R
r


Nom du client: grossiord

Accompte: 0


numero de resa = 2

Reservation effectuee.


    Tapez q pour quitter
          r pour menu principal
q
```

Après ces tests, l'état du fichier *reservation*, initialement vide, est:

```
1   :poiraud   :AF457:nice     :22h00:paris     :23h20:21/07/95:eco  :400  :100
2   :grossiord :AF330:lille    :22h30:toulouse  :23h50:23/07/95:club :1030 :0
```

4. EXERCICES

4.1. ENONCES

4.1.1. Chapitre 2.1.

Ecrire un programme qui attend que l'utilisateur saisisse un rayon avant de calculer le périmètre du cercle.

4.1.2. Chapitre 2.2.

Ecrire un programme qui lit un chiffre et une liste de chaînes (chacune sur une ligne) et qui retourne la ligne sélectionnée par le chiffre.

4.1.3. Chapitre 2.3.

1. Ecrire un programme demandant la température extérieure, et affichant « trop chaud! » si elle est supérieure à 30°, « trop froid! » si elle est inférieure à 20°, « impeccable! » si elle est comprise entre 20 et 30 °.

2. Ecrire un programme qui lit une liste de chaînes (chacune sur une ligne) et les affiche dans l'ordre inverse (sans utiliser *reverse*, bien sûr!).

4.1.4. Chapitre 2.4.

Ecrire un programme qui lit une série de mots (un mot par ligne) et qui affiche le nombre de fois que figure chaque mot dans la liste.

4.1.5. Chapitre 2.5.

Ecrire un programme faisant la même chose que *cat* mais inversant l'ordre des lignes.

4.1.6. Chapitre 2.6.

Ecrire un programme qui parcourt */etc/passwd*, cherche deux utilisateurs ayant le même nom et affiche ces noms.

4.1.7. Chapitre 2.7.

Ecrire une subroutine qui prend les valeurs numériques de deux chiffres, les additionne et affiche le résultat sous la forme: `"deux plus deux égale quatre. "`

4.1.8. Chapitre 2.8.

Modifier le problème précédent de façon à ce que l'opération soit répétée jusqu'à ce que le mot `end` soit saisi.

4.1.9. Chapitre 2.9.

Ecrire un programme prenant un nom de fichier d'entrée, un nom de fichier de sortie, un pattern à chercher, une chaîne de remplacement, et qui remplace toute occurrence du pattern dans le fichier d'entrée par la chaîne de remplacement et met le résultat dans le fichier de sortie.

4.1.10. Chapitre 2.10.

Ecrire un programme ouvrant */etc/passwd* et affichant les logins utilisateur, les UID et les vrais noms sous forme de colonnes.

4.1.11. Chapitre 2.11.

Ecrire un programme qui permet de changer de répertoire et de se retrouver dans celui saisi par l'utilisateur, et qui affiche la liste des fichiers de ce répertoire dans l'ordre alphabétique.

4.1.12. Chapitre 2.12.

Ecrire un programme réalisant la même chose que *mv*.

4.1.13. Chapitre 2.13.

Ecrire un programme qui récupère tous les vrais noms d'utilisateur de */etc/passwd* et transforme la sortie de la commande *who*, en remplaçant les logins par les vrais noms.

4.1.14. Chapitre 2.14.

Ecrire un programme affichant les logins et les vrais noms d'utilisateur du fichier */etc/passwd*, ordonnés par le prénom.

4.1.15. Chapitre 2.15.

Ecrire un programme créant une correspondance entre UID et vrai nom d'utilisateur à partir de */etc/passwd* et utiliser cette correspondance pour afficher une liste de vrais noms appartenant à chaque groupe du fichier */etc/group*.

4.1.16. Chapitre 2.16.

Ecrire deux programmes: l'un qui lit les données de l'opérateur diamant, les découpe en mots, et met à jour un fichier DBM indiquant le nombre de fois qu'apparaît chaque mot; l'autre qui ouvre un fichier DBM et affiche les résultats dans l'ordre décroissant.

4.1.17. Chapitre 2.17.

Convertir le script shell suivant en Perl:

```
cat /etc/passwd |
awk -F: '{print $1, $6}' |
while read user home
do
 newsrc="$home/.newsrc"
 if [ -r $newsrc ]
 then
      if grep -s  '^comp\.lang\.perl: ' $newsrc
      then
       echo "$user is a good person, and read comp.lang.perl! "
      fi
 fi
done
```

4.2. SOLUTIONS

Ces solutions ne sont pas uniques!

4.2.1.

```
$pi = 3.14;
print "Donnez le rayon du cercle : \n";
chop($rayon = <STDIN>);
print "Le périmètre du cercle est : ".(2*$pi*$rayon);
```

4.2.2.

```
print "Donnez un chiffre: \n";
chop($compte = <STDIN>);
print "Donnez une liste de chaînes, terminez l'entrée par ^D:
\n";
@tab = <STDIN>;
print "La $compteième chaîne est $tab[$compte-1].\n";
```

4.2.3.

```
1.   print "Donnez la température ambiante : \n";
     chop($temp = <STDIN>);
     if ($temp > 30)
     { print "Trop chaud!\n"; }
     elsif ($temp < 20)
     { print "Trop froid! \n"; }
     else
     { print "Impeccable!\n");
```

2.
```
print "Donnez la liste de chaînes, terminez l'entrée avec ^D:
\n";
@liste = <STDIN>;
$i = $#liste-1;
unless ($i == -1)
{ print "$liste[$i--]\n"; }
# on peut aussi utiliser pop()
```

4.2.4.

```
print "Donnez une liste de mots : \n";
@liste = <STDIN>;
$tab{chop(shift(@liste))} ++;
while ( ($cle, $val) = each(%tab) )
{ print "Le mot $cle apparaît $val fois.\n";  }
```

4.2.5.

```
print reverse <>;
```

4.2.6.

```
while (<>)
{ ($user, $pass, $uid, $gid, $gcos) = split(/:/);
   ($real) = split(/,/,$gcos);
   ($first) = split(/\s+/, $real);
   $seen{$first}++;
}
foreach (keys %seen)
{  if ($seen{$_} > 1)
    { print "$_ a été vu $seen{$_} fois.\n"; }
}
```

4.2.7.

```
sub card
{  @card_map  =  (0,  "un",  "deux",  "trois",  "quatre",  "cinq",
"six", "sept", "huit", "neuf");
   local($nombre) = @_;
   if ($card_map[$nombre])
   {  $card_map[$nombre]; }
   else
   { $nombre; }

print "Donnez le premier nombre: \n";
chop($prem = <STDIN>);
print "Donnez le deuxième nombre: \n";
chop($deux = <STDIN>);
$message  =  &card($prem)  .  "plus"  .  &card($deux).  "égale"  .
&card($prem+$deux) . ".\n";
$message =~ s/^./\u$&/;
print $message;
```

4.2.8.

```
sub card { } # même que pour l'exercice précédent

while ()
{ ##NEW##
   print "Donnez le premier nombre: \n";
   chop($prem = <STDIN>);
   last if $prem eq "end"; ##NEW##
   print "Donnez le deuxième nombre: \n";
   chop($deux = <STDIN>);
   last if $deux eq "end"; ##NEW##
      $message=&card($prem)   ."plus"   .&card($deux).   "égale"
.&card($prem+$deux) . ".\n";
  $message =~ s/^./\u$&/;
   print $message;
} ##NEW##
```

4.2.9.

```
print "Nom du fichier d'entrée: \n";
chop($fic_in = <STDIN>);
print "Nom du fichier de sortie: \n";
chop($fic_out = <STDIN>);
print "Pattern à rechercher: \n";
chop($pattern = <STDIN>);
print "Chaîne de remplacement: \n";
chop($chaine = <STDIN>);
open (IN, $fic_in) || die("Impossible d'ouvrir $fic_in!\n");
die("Je n'écraserai pas $fic_out!\n) if -e $fic_out;
open (OUT, ">$fic_out") || die("Impossible de créer
$fic_out!\n");
while (<IN>)
{ s/$pattern/$chaine/g;
   print OUT $_; }
close(IN);
close(OUT);
```

4.2.10.

```
open (PW, "/etc/passwd") || die("Comment vous êtes-vous loggé?
\n");
while (<PW>)
{ ($user, $passwd, $uid, $gid, $gcos) = split(/:/);
   ($real) = split(/,/, $gcos);
   write; }

format STDOUT =
@<<<<<<< @>>>>>> @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$user,          $uid,          $real
.
```

4.2.11.

```
print "Où voulez-vous aller? :\n";
chop($rep=<STDIN>);
chdir ($rep) || die ("Impossible d'accéder ce répertoire!\n");
foreach (<*>)
{ print "$_\n"; }
```

4.2.12.

```
($old, $new) = @ARGV;
if (-d $new)
{  ($base = $old) =~ s#.*/##;
    $new .= "/$base";  }
rename ($old, $new);
```

4.2.13.

```
open (PW, "etc/passwd");
while (<PW>)
{ chop;
   ($user, $passwd, $uid, $gid, $gcos) = split(/:/);
   ($real) = split(/,/, $gcos);
   $real{$user} = $real;  }
close(PW);

open (WHO, "who |") || die ("Impossible d'ouvrir le pipe du
who!\n");
while (<PW>)
{ ($login, $reste) = /^(\S+)\s+(.*)/;
   $login = $real{$login} if $real{$login};
   printf "%-30s %s\n", $login, $reste;  }
```

4.2.14.

```
open (PW, "etc/passwd");
while (<PW>)
{ chop;
   ($user, $passwd, $uid, $gid, $gcos) = split(/:/);
   ($real) = split(/,/, $gcos);
   $real{$user} = $real;
   ($prenom =  $real) =~ s/^(.*[^a-z])?([a-z]+.*).*/$2/i;
   $prenom =~ tr/A-Z/a-z/;
   $prenom{$user} = $prenom; }
close(PW);
for (sort by_last keys %prenom)
{ print "%30s  %8s\n", $real{$_}, $_;  }

sub by_last
{ ($prenom{$a} cmp $prenom{$b}) || ($a cmp $b); }
```

4.2.15.

```
$: = " ";
while (@pw = getpwent)
{ ($user, $gid, $gcos) = @pw[0, 3, 6];
   ($real) = split(/,/, $gcos);
   $real{$user} = $real;
   $members{$gid} .= " $user";
   ($prenom = $user) =~ s/^(.*[^a-z])?([a-z]+.*).*/$2/i;
   $prenom =~ tr/A-Z/a-z/;
   $prenom{$user} = $prenom; }

while (@gr = getprent)
{ ($gname, $gid, $members) = @gr[0, 2, 3];
   $members{$gid} .= " $members";
   $gname{$gid} = $gname;  }

for $gid (sort by_gname keys %gname)
{ %all = ( );
   for (split(/\s+/, $members{$gid}) )
   { $all{$_}++ if length $_;  }
   @members = ( );
   foreach (sort by_last keys %all)
   { push(@members, "$real{$_} ($_)"); }
   $memberlist = join(", ", @members);
   write;
}

sub by_gname
{ $gname{$a} cmp $gname{$b}; }

sub by_last
{ ($last{$a} cmp $last{$b}) || ($a cmp $b) ; }

format STDOUT =
@<<<<<<<< @<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$gname{$gid}, "($gid) ",    $memberlist
~~
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$memberlist
.
```

4.2.16.

```
# programme 1
dbmopen (%MOTS, "mots", 0644);
while (<>)
{ foreach $mot ( split(/\W+/) )
    { $MOTS{$mot}++;  }
}
dbmclose(%MOTS);

# programme 2
dbmopen (%MOTS, "mots", undef);
foreach $mot ( sort { $MOTS{$b} <=> $MOTS{$a} } keys %MOTS )
{  print "$mot $MOTS{$mot}\n" ;  }
dbmclose(%MOTS);
```

4.2.17.

```
for ( ; ; )
{ ($user, $home) = (getpwent)[0,7];
   last unless $user;
   next unless open(N, "$home/.newsrc");
   next unless -M N < 30;
   while (<>)
   {  if (/^comp\.lang\.perl:/)
 { print "$user is a good person and reads comp.lang.perl!\n";
    last;
 }
   }
}
```

ANNEXE : LE GUIDE DE REFERENCE DE PERL