Débuter en Perl

-
François Dagorn
-
Olivier Salaun
-
19 avril 2004

Résumé

Ce document est une présentation tutoriale du langage Perl. Il ne couvre pas tous les aspects du langage, sa lecture ne dispense pas de consulter les ouvrages de références cités dans la bibliographie.

Une version électronique est disponible à l'adresse suivante :
`http://perso.univ-rennes1.fr/Francois.Dagorn/perl`

Table des matières

Chapitre 1

Introduction

1.1 Généralités

Perl est un langage de programmation adapté au traitement des fichiers textes ainsi qu'aux tâches d'administration des systèmes et des applications. Disponible dans le domaine public pour les environnements Unix, Linux, Windows ... il connait aujourd'hui un développement très important.

Crée en 1986 par Larry Wall, Perl a depuis connu de nombreuses versions, aujourd'hui on utilise Perl5, la version 5.6 [1] est maintenant définitive.

Perl est un langage interprété et les programmes Perl se diffusent en format source. Le terme interprété est toutefois ambigu car un programme Perl s'exécute en deux phases : la pré-compilation du programme qui produit du pseudo-code, puis l'interprétation du pseudo-code (l'exécution proprement dite). Le pseudo-code de Perl contient plus de 300 méta-instructions, à l'issue de la pré-compilation on obtient une suite de pseudo-codes autonomes qui disposent de pointeurs vers leurs opérandes et la méta-instruction suivante.

Perl est très référencé sur le Web, parmi les principaux sites citons :

- http://www.perl.org (tout sur Perl)
- http://www.perl.com (tout sur Perl également)
- http://www.cpan.org (distribution de Perl et des modules)

1. http://www.perl.com/pub/language/info/software.html#stable

1.2 Un aperçu de la syntaxe

Perl est sensible à la casse (les majuscules et minuscules sont différenciées), c'est à dire que **print** est une fonction du langage, alors que **Print** ne l'est pas, mais peut bien sûr être un identificateur (de variables, de fonction, de fichier) à la charge du programmeur.

Il n'y a pas de formatage particulier dans l'écriture d'une instruction Perl, le séparateur peut être un espace, une tabulation ou même un retour à la ligne. Les instructions sont séparées entre elles par le caractère **;** (le point virgule) et un commentaire est introduit en plaçant le caractère # à n'importe quel endroit d'une ligne (mais il ne s'étend que sur une ligne).

Un programme Perl (souvent appelé **script**) débute en général par une ligne qui indique l'endroit où se trouve le compilateur/interpréteur[2], c'est ainsi que l'on peut sur des systèmes Unix démarrer une application Perl en citant simplement son nom. Il n'est pas inopportun de procéder de même sur des systèmes Windows[3], du moins dans un environnement WWW/CGI (cf. section 18) car des serveurs http (tel APACHE) savent tenir compte de cette première ligne et lancer Perl en fonction de ce qu'elle indique.

2. #!/usr/local/bin/perl par exemple ...

3. Sur lesquels on lance une application Perl en la citant derrière le chemin d'accès à l'interpréteur ou en effectuant une association entre un suffixe de fichier (**.pl** par exemple) et l'interpréteur Perl.

Chapitre 2

Les types de données

En Perl il n'existe que 3 types de variables : les scalaires, les tableaux et les tableaux associatifs (*hash*).

2.1 Les scalaires

Une variable de type scalaire est un nombre ou une chaîne de caractères. Il n'existe pas de type particulier pour les valeurs entières, les nombres réels, les caractères ... Lorsque dans une expression figurent des variables scalaires correspondants à des valeurs de genres différents, Perl effectue des conversions totalement transparentes (et dangereuses).

2.1.1 Les nombres

Les valeurs littérales numériques qui peuvent être affectées à une variable scalaire s'expriment classiquement :

```
12     (une valeur entière positive)
+10    (une autre valeur entière positive)
-34    (une valeur entière négative)
+3.14  (un réel positif)
5e15   (5 fois 10 puissance 15)
033    (33 en code octal soit 27 en décimal)
x1F    (1F en hexa soit 31 en décimal)
```

Les opérateurs pour les nombres

Les opérateurs arithmétiques sont les suivants :

```
+  (addition)          -   (soustraction)
*  (multiplication)    **  (puissance)
/  (division)          %   (modulo)
```

Les opérateurs de comparaison de nombres sont les suivants :

```
<  (inférieur)          >  (supérieur)
<= (inférieur ou égal) >= (supérieur ou égal)
== (égalité)           != (différence)
```

2.1.2 Les chaînes de caractères

Une chaîne de caractères Perl comprend de 0 à *n* caractères ASCII dont le code est compris entre 0 et 255. Il n'y a donc pas de restriction sur le caractère NUL ou sur les autres caractères non imprimables. Il y a deux manières de spécifier la valeur littérale d'une chaîne :

– utiliser une **simple quote** comme délimiteur, le contenu est pris tel que, il n'y a pas d'interprétation de caractères particuliers. Dans ce cas pour introduire une simple quote dans la chaîne, il convient de la précéder d'un antislash (\) , pour introduire un antislash, il faut le doubler.

```
'Bonjour'              une chaîne
'Bonjour \n'           le \n n'a pas de sens  ici
'L\'école'             le \ devant ' pour l'apostrophe
'c:\\windows\\system' antislash doublés
''                     chaîne vide
```

– utiliser une **double quote** comme délimiteur, certaines séquences de caractères seront interprétées. L'antislash (\) est utilisé ici comme caractère d'échappement et s'utilise comme suit :

```
"\n"     une nouvelle ligne
"\t"     une tabulation
"\033"   un caractère codé en octal (33 ici)
"\x1F"   un caractère codé en hexa (1F ici)
"\cA"    un caractère de Contrôle (Ctrl-A ici)
"\\"     un \
"\""     une double quote
```

Les chaînes de caractères présentées entre double quotes permettent également d'interpréter le contenu de variables (cf. 2.1.4).

Les opérateurs pour les chaînes de caractères

L'opérateur de concaténation est le **.** (le point) :

```
"Bonjour"."Monsieur"       <==> "BonjourMonsieur"
'Il'.' '."fait beau \n"    <==> "Il fait beau \n"
```

Les opérateurs de comparaison de chaînes de caractères sont les suivants :

```
eq  (égalité)              ne  (différence)
lt  (inférieur)            gt  (supérieur)
le  (inférieur ou égal)    ge  (supérieur ou égal)
```

Les opérateurs **q** et **qq** permettent de remplacer respectivement les délimiteurs de chaîne **quote** et **double quote** par un délimiteur au choix du programmeur :

```
$chaine='S\'il vous plait';
peut aussi s'écrire
$chaine=q@s'il vous plait@;
C'est le caractère @ qui est ici utilisé comme délimiteur
```

2.1.3 Les variables scalaires

Une variable scalaire est représentée par une suite de caractère précédée du symbole **$**. Le premier caractère suivant le **$** doit être une lettre, ensuite on peut trouver un nombre

quelconque de chiffres et de lettres. Les majuscules et minuscules sont différenciées et le caractère _ (souligné) est autorisé :

```
$Une_Variable       un nom de variable correct
$une_variable       un autre nom de variable
$une-variable       un nom de variable incorrect (- vs. _)
```

L'affectation des variables scalaires

L'opérateur d'affectation en Perl est le signe =

```
$I = 10;     (affecte la constante 10 à la variable I)
$K = $I + 1; (affectation de I + 1 à la variable K soit 11)
$chaine = 'Bonjour' . 'à tous';  (affectation d'une chaîne)
```

Les programmeurs SH ou CSH noteront que pour référencer une variable, il convient de précéder son nom par le symbole $ de part et d'autre de l'opérateur d'affectation (le signe =).

Il existe d'autres opérateurs d'affectation, ce sont des raccourcis d'écriture familiers aux programmeurs **C** :

– **affectation et opération combinée**

```
$var += 2;        <==> $var = $var + 2;
$var *= 3;        <==> $var = $var * 3;
$chaine .= 'xxx'; <==> $chaine = $chaine . 'xxx';
```

– **Incrémentation et décrémentation automatique**

```
$I = 10;
$I++;        <==> $I = $I + 1;  donc I = 11
$K = ++$I;   <==> $K = $I = $I + 1; donc K = I = 12
$K = $I++;   <==> $K = $I; $I = $I + 1;
                  donc K = 12 et I = 13
```

L'incrément se fait avant l'affectation si l'opérateur précède le nom de la variable (++**$I**) ou après l'affectation si l'opérateur suit le nom de la variable (**$I**++).

2.1.4 Interprétation des variables dans une chaîne de caractères

Des variables scalaires peuvent être interprétées lorsqu'elles sont contenues dans des chaînes de caractères présentées entre double quotes. À cet effet, Perl recherche le caractère **$** et considère ce qui le suit comme un nom de variable potentiel. Si la variable existe, son contenu est intégré à la chaîne source, si elle n'existe pas, elle est remplacée par une chaîne vide. Il n'y a qu'un seul niveau d'interprétation, si une variable contient elle même une chaîne faisant référence à une autre variable, il n'y aura aucun traitement. Pour empécher l'interprétation comme un nom de variable des caractères qui suivent le caractère **$**, il convient de le précéder d'un antislash ou de placer le morceau de chaîne considéré entre simple quotes.

```
$chaine1 = "Au revoir";
$chaine2 = "$chaine1 les petits amis";
$chaine3 = "au plaisir ";
$chaine4 = "${chaine3}de vous revoir";  Les accolades sont
            utilisées ici comme en sh, csh pour indiquer
            explicitement la variable à interpréter.
```

2.1.5 Des fonctions pour manipuler les scalaires

En plus des opérateurs du langage, Perl propose des fonctions (cf. section 11) adaptées au traitement des variables scalaires, parmi celles-ci citons :

- **chop()** enlève le dernier caractère d'une chaîne;
- **chomp()** enlève le dernier caractère d'une chaîne si celui-ci est un délimiteur de fin de ligne;
- **chr()** convertit une valeur entière en le caractère correspondant du code ASCII;
- **hex()** renvoie la valeur décimale correspondant à une chaîne de caractères héxadécimaux;
- **oct()** renvoie la valeur décimale correspondant à une chaîne de caractère octaux;
- **index()** renvoie la position de la première occurrence d'une sous chaîne dans une chaîne;
- **lc()** convertit une chaîne de caractères en caractères minuscules;
- **length()** indique la longueur en caractère d'une variable scalaire;
- **rindex()** renvoie la position de la dernière occurrence d'une sous chaîne dans une chaîne;
- **substr()** permet d'extraire d'une chaîne une sous chaîne définie en position et longueur.

```
$Esc = chr(27);      # convertit l'entier 27 en ASCII
$B10 = hex("1F3C");  # $B10 vaut 7996
$B10 = oct("1073");  # $B10 vaut 571
$Min = lc("ABCDE");  # $Min vaut "abcde"
$Lg  = length($Min); # $Lg vaut 5  (longueur en caractères
$Lg  = length($B10); # $Lg vaut 3   d'un scalaire)
```

2.2 Les tableaux

Une variable de type tableau est une liste de données scalaires (nombres et/ou chaînes). Chaque élément du tableau est une variable scalaire accessible comme telle.

2.2.1 Les éléments d'un tableau

Les valeurs littérales qui peuvent être affectées à une variable de type tableau s'expriment comme une liste de valeurs séparées par des virgules et encadrées par des parenthèses.

```
()                 <==> tableau vide
(8,4,9,4,5)        <==> 5 éléments (des nombres)
('toto',"$var",3)  <==> 3 éléments (la chaîne 'toto',
 la valeur courante de la variable $var et le nombre 3)
```

Un opérateur de construction liste (**..**) peut être utilisé pour spécifier un élément de tableau.

```
(1..10)          <==> 10 éléments (les nombres de 1 à 10)
(1..5,10,20..30) <==> 16 éléments (les chiffres de 1 à 5,
                      puis le nombre 10 et les 11 nombres
                      de 20 à 30
($debut..$fin)   <==> (($fin+1) - $debut) éléments ayant
           pour valeurs de $debut à $fin par incrément de 1
```

2.2.2 Les variables tableaux

Le nommage d'une variable tableau suit les mêmes conventions que celui d'une variable scalaire, hormis le premier caractère qui est ici **@** (un arobas). C'est le premier caractère qui determine le type de la variable utilisée, si **@tab** est une variable de type tableau, elle n'a rien à voir avec **$tab** qui est une variable de type scalaire. Une variable de type tableau qui n'a jamais été affectée a pour valeur une liste vide **()**.

L'affectation des variables tableaux

Le signe égal est l'opérateur d'affectation des variables tableaux.

```
$I = 5;
@tab1 = ();          <==> @tab1 est vide
@tab2 = (5,3,15);    <==> @tab2 contient 3 éléments de
                          valeurs 5, 3 et 15
@tab1 = @tab2;       <==> @tab1 est une copie de @tab2
@tab2 = $I;          <==> @tab2 reçoit un scalaire et
     ne contient plus qu'un seul élément (de valeur 5)
($a,$b,$c) = @tab1; <==> $a = 5; $b = 3; $c = 15;
```

accès aux éléments d'une variable tableau

Les éléments constitutifs d'une variable de type tableau sont des scalaires, la référence à un élément commence donc par le symbole **$**, un indice exprimé entre crochets (**[]**) indique l'occurrence ciblée. Le premier élément d'un tableau est accessible par l'indice 0.

```
$I = 5;
@tableau = ('x',2,3,4,5,'x');
$tableau[0] = 1;         <==> accès au 1er élément du tableau
$tableau[$I] = 6;        <==> accès au Ième élément
```

L'indice du dernier élément d'une variable tableau est accessible par la variable scalaire **$#nom-du-tableau**. Si on affecte un élément au delà de cette limite, les *trous* sont

bouchés par des élèments de valeur **undef**[1].

```
@tab = (10,20,30);
$tab[$#tab+1] = 40;  <==> $tab[3] = 40;
                          $#tab vaut maintenant 3
$tab[10] = 1000;     <==> $#tab vaut maintenant 10, les
                    éléments intermédiaires $tab[4]..$tab[9]
                    prennent la valeur undef
```

L'affectation d'une variable tableau dans une variable scalaire rend le nombre d'éléments de la variable tableau :

```
@tab = (1,2,3,4);
$scal = @tab;
#  $scal vaut 4
```

2.2.3 Des fonctions pour manipuler les variables tableaux

Diverses fonctions permettent d'effectuer des décalages, des tris ... sur des listes (des variables tableaux).

Les fonctions push() et pop()

Les fonctions **push()** et **pop()** permettent de travailler sur l'extrémité droite d'une liste (ajout ou suppression d'éléments) :

```
@liste=('a','b');
push(@liste,('c','d','e')); <==> @liste=('a','b','c','d','e')
pop(@liste);                <==> @liste=('a','b','c','d')
```

Les fonctions shift() et unshift()

1. **undef** vaut 0 pour un nombre ou vide pour une chaîne. On peut indifféremment utiliser les opérateurs de comparaison == ou **eq** pour comparer un scalaire à la valeur **undef**

Les fonctions **shift()** et **unshift()** permettent de travailler sur l'extrémité gauche d'une liste (ajout ou suppression d'éléments) :

```
@liste=('c','d','e');
unshift(@liste,('a','b')); <==> @liste=('a','b','c','d','e')
shift(@liste);             <==> @liste=('b','c','d','e')
```

La fonction sort()

La fonction **sort()** permet de trier une liste dans l'ordre ASCII de ses éléments (les nombres sont convertis en chaînes avant le tri) :

```
@liste=(100,1,2,58,225);
@tri=sort(@liste);  <==> @tri=('1','100','2','225','58')
@liste=('rouge','vert','bleu');
@tri=sort(@liste);  <==> @tri=('bleu','rouge','vert')
```

La fonction reverse()

La fonction **reverse()** permet d'inverser les éléments d'une liste :

```
@liste=(1,2,3,4,5);
@rev=reverse(@liste);      <==> @rev=(5,4,3,2,1)
```

La fonction chop()

La fonction **chop()** permet de supprimer le dernier caractère de tous les éléments d'une liste. Elle fonctionne également sur une variable scalaire et est surtout utilisée pour supprimer les caractères de fin de lignes contenus dans des lignes (résultats de saisies ...) :

```
@liste=("tutu\n","toto\n");
chop(@liste);              <==>  @liste=("tutu","toto")
```

La fonction chomp()

La fonction **chomp()** permet de supprimer le dernier caractère (seulement s'il s'agit d'un délimiteur de fin de ligne) de tous les éléments d'une liste. Il est donc préférable d'utiliser **chomp()** pour supprimer les éventuels délimiteurs de fin de lignes résultant d'une saisie, d'une lecture de fichier, ...

La fonction qw()

L'affectation de constantes litérales dans un tableau est fréquente et d'une écriture un peu fastidieuse, Perl propose la fonction **qw()** qui travaille sur une liste de mots :

```
@liste=("tutu","toto",'machin'); # est équivalent à
@liste = qw(tutu toto machin);   # ou même à
@liste = qw(tutu
            toto
            machin);
```

2.3 Les tableaux associatifs (hashes)

Un tableau associatif (ou hash) est un tableau dont les éléments (des scalaires) sont accédés au moyen d'un index qui est une valeur scalaire quelconque. Les éléments d'un hash sont des couples (clef, valeur).

2.3.1 Les variables tableaux associatifs

Le nommage d'un tableau associatif suit les mêmes conventions que celui d'une variable scalaire, hormis le premier caractère qui est ici un % (le caractère pour-cent).

```
%trad = ('january','janvier','february','avril');

#  On affecte le hash %trad par une liste qui contient un
#  nombre pair d'éléments correspondant à des couples
#  clef, valeur.

$trad{'february'}='fevrier';
# l'élément du hash %trad dont la clef est 'february' reçoit
# une nouvelle valeur.
```

Une autre forme peut être utilisée pour spécifier les valeurs d'un tableau associatif, elle met en évidence la correspondance clef/valeur :

```
%trad = ( 'january'  => 'janvier',
          'february' => 'fevrier',
          'march'    => 'mars' );
```

2.3.2 Des fonctions adaptées aux tableaux associatifs

keys

La fonction **keys(%var_hashee)** rend une liste des clefs d'un hash.

```
%hash = ('un',1,'deux',2,'trois',3,'quatre',4);
@liste_clef = keys(%hash);
# <==> @liste_clef = ('un','deux','trois','quatre');
```

values

La fonction **values(%var_hashee)** rend une liste des valeurs d'un hash.

```
%hash = ('un',1,'deux',2,'trois',3,'quatre',4);
@liste_val = values(%hash);
# <==> @liste_val = (1,2,3,4);
```

each

La fonction **each(%var_hashee)** rend un couple clef,valeur dans une liste à deux éléments. Chaque appel de **each** sur une même variable de type hash rend le couple suivant.

```
%hash = ('un',1,'deux',2,'trois',3,'quatre',4);
($clef,$valeur) = each(%hash);
# au 1er appel <==> $clef='un'; et $valeur=1;
```

each est la fonction adaptée pour parcourir entièrement un hash, elle rend une liste vide lorsque la fin du hash est atteinte.

delete

La fonction **delete** permet de supprimer un élément d'un hash en indiquant sa clef.

```
%hash = ('un',1,'deux',2,'trois',3,'quatre',4);

delete($hash{'deux'});
```

Chapitre 3

Les structures de contrôle

Diverses structures de contrôle sont proposées en Perl, elles évaluent une expression et proposent un traitement selon les schémas algorithmiques classiques. L'expression à évaluer est convertie en chaîne, une chaîne non vide ou différente de "0" correspond à la valeur **vrai**. Les opérateurs de comparaison (==,>,gt,...) retournent 1 pour vrai et 0 pour faux ("0" lorsque converti en chaîne), ci-dessous quelques exemples d'expressions vraies ou fausses :

```
0           <==> faux
"0"         <==> faux, c'est 0 converti en chaîne
''          <==> faux, chaîne vide
'00'        <==> vrai, différent de 0 ou '0'
1           <==> vrai
"n'importe quoi" <==> vrai
undef       <==> undef rend une chaîne vide donc faux
```

Perl permet de structurer des instructions en utilisant des blocs délimités comme en C par les accolades ({}). La structuration en blocs a un impact sur la portée des variables (cf. section 10).

3.1 L'instruction if

La syntaxe générale de l'instruction **if** est la suivante :

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

Si il n'y a rien de particulier à faire dans le cas ou l'expression est fausse, on peut se passer du **else** {**...**}. Par contre même s'il n'y a qu'une seule instruction à exécuter, il convient de la placer dans un bloc {}.

```
if ($I == 0) { $I++; }       <==> le else est omis
if ((10 == 10) == 1) { ... } <==> vrai car == rend 1 si la
                                  condition est réalisée
```

Le test de plusieurs cas peut être enchainé en utilisant **elsif** :

```
if ($Couleur eq "Bleu") { ... }
elsif ($Couleur eq "Rouge") { ... }
    elsif ($Couleur eq "Vert") { ... }
        else { ... }

if ($Couleur eq "Noir") { ... }
else if ($Couleur eq "Jaune")   <==> erreur de syntaxe,
               un else doit être suivi par une accolade {
```

3.2 L'instruction unless

L'instruction **unless** fonctionne comme un **nif** (non if : si l'expression est fausse alors faire).

```
unless ($I > 0) { ... }
else { ... }
```

Un **elsif** peut suivre un **unless**, le **elseunless** n'existe pas.

3.3 Les instructions while et until

Perl dispose d'une structure *tant que* (**while**) et d'une structure *jusqu'à ce que* (**until**).

```
$I = 0;
while ($I < 10) { $I++; }  <==> tant que I est
                               inferieur à 10
$I = 0
until ($I == 10) { $I++; } <==> jusqu'à ce que
                               I soit égal à 10
while (1) { ... }          <==> sans fin
while ()  { ... }          <==> sans fin, une expression
                           inexistante  n'est pas vide !
until (0) { ... }          <==> sans fin
```

Pour forcer la sortie d'une boucle **while** ou **until** sans s'occuper de la condition d'arrêt, on utilise l'instruction **last** (de la même façon qu'un **break** en Langage C). Pour faire évaluer la condition sans avoir déroulé la totalité des instructions du bloc, on utilise l'instruction **next**.

3.4 L'instruction for

L'instruction **for** correspond à une boucle *pour* dans laquelle on fournit une valeur de départ, une expression à évaluer et une expression permettant une réinitialisation (incrément,

décrément, ...).

```
for ($I=0; $I < 10; $I++) { ... }
```

Dans l'exemple ci-dessus, on affecte 0 à I, on évalue ($I < 10), si c'est vrai on exécute les instructions incluses dans le bloc et on passe à l'incrémentation de I, si c'est faux on passe à l'instruction suivante.

```
for (;;) { ... }   <==> boucle sans fin, les expressions
                        ne sont pas obligatoires
```

Les instructions **next** et **last** fonctionnent de la même manière que pour **while** et **until** (cf. section 3.3).

3.5 L'instruction foreach

L'instruction **foreach** charge dans une variable les valeurs successives d'une liste et effectue le traitement spécifié dans le bloc d'instructions. Quand la fin de la liste est atteinte, on passe à l'instruction suivante :

```
foreach $Couleur ('jaune','vert','bleu') { ... }
        la variable couleur prend successivement les
        valeurs jaune, vert et bleu
foreach $var (@tableau1, @tableau2, $I, $K, 10)
        la variable $var prend successivement toutes les
        valeurs scalaires contenues dans $tableau1 ...
        jusqu'à la valeur 10
```

Ici aussi, **last** provoque une sortie forcée de la boucle et **next** un passage à la valeur suivante.

3.6 Exécuter si l'expression précédente est vraie ou fausse

L'opérateur **&&** permet d'évaluer une expression si celle qui la précède est vraie, l'opérateur || permet d'évaluer une expression si celle qui la précède est fausse :

```
$J = $I % 2 && printf ("I est impair \n");

   I modulo 2 vaut 0 ou 1, si 1 le nombre est impair
                           si 0 le nombre est pair

$J = $I % 2 || printf ("I est pair \n");
```

Les opérateurs **&&** et || peuvent être utilisés pour former des expressions conditionnelles évaluées par l'instruction **if** :

```
if (($a == $b) || ($a == $c)) { ... }
```

Une dernière construction est possible, elle admet 3 expressions séparées par les opérateurs **?** et **:**, la seconde est évaluée si la première est vraie, la troisième est évaluée si la première est fausse :

```
$J = $I % 2 ? printf ("impair \n") : printf ("Pair \n");
```

Chapitre 4

Entrée standard et sortie standard

Compte tenu du passé Unix de Perl, on lit sur le STDIN et on écrit sur le STDOUT. Par défaut il s'agit du clavier et de l'écran, mais Perl ne s'en occupe pas, c'est au système d'exploitation de lui fournir des descripteurs valides.

4.1 Lecture sur l'entrée standard

L'opérateur de lecture sur l'entrée standard est <**STDIN**>.

```
$ligne = <STDIN>;   <==> on affecte à la variable $ligne
                         la ligne suivante
@lignes = <STDIN>;  <==> le tableau @lignes reçoit
                         un élément par lignes lues
```

Pour lire et traiter ligne à ligne on procède en général comme suit :

```
while ($ligne = <STDIN>)    Lorsque la dernière ligne est
   { ... }                  atteinte, <STDIN> retourne undef
                            et la boucle while s'arrête
```

Perl autorise une autre forme de lecture du STDIN avec l'opérateur <> qui passe au programme tous les fichiers indiqués sur la ligne de commande (chaque argument de la ligne de commande est alors considéré comme un nom de fichier).

4.2 Écriture sur la sortie standard

L'instruction **print** écrit une liste de chaîne de caractères sur la sortie standard.

```
print (@liste1, $chaine, @liste2, 'toto');
```

Il est possible de contrôler la présentation des impressions sur la sortie standard en utilisant l'instruction **printf** qui fonctionne comme en **C**. Elle admet comme argument une liste dont le premier élément est une chaîne de caractères qui contient le masque d'impression des variables ainsi que des valeurs constantes à intercaler entre elles (les autres éléments de la liste sont les variables à imprimer).

```
printf ("%s %s %5d \n",$chaine1,$chaine2,$I);
%s  est le masque d'impression de $chaine1
%s  est le masque d'impression de $chaine2
%5d est le masque d'impression de $I
\n  constante qui introduit une nouvelle ligne
```

Si un argument de **printf** est une variable tableau, il convient de spécifier autant de masques d'impression que de valeurs scalaires distinctes contenues dans le tableau. Les masques d'impression indiquent les conversions à réaliser et s'utilisent comme dans l'exemple suivant :

```
%10s     <==> une chaîne sur sa longueur réelle complétée
              éventuellement par des espaces (10 maxi)
%.5s     <==> les 5 premiers caractères d'une chaîne
%3d      <==> un entier tel que ou sur 3 caractères s'il
              est inférieur à 99 (espace à gauche)
%o       <==> un nombre en octal
%x       <==> un nombre en héxadécimal
%f       <==> un réel avec 6 décimales
%.3f     <==> un réel avec 3 décimales
%e       <==> un réel en représentation mantisse/exposant
```

Chapitre 5

La variable $_

Perl utilise une variable scalaire fourre-tout nommée **$_** ($ souligné) comme résultat par défaut d'un certain nombre d'instructions. C'est notamment le cas avec la lecture sur l'entrée standard :

```
while (<STDIN>)          <==>  while ($_ = <STDIN>)
{                        <==>  {
    chop;                <==>     chop($_);
}                        <==>  }
```

La variable **$_** est également implicitement utilisée par les instructions **foreach** et **print**[1], ainsi que par les instructions qui manipulent des expressions régulières (cf. section 7) ou celles qui testent l'existence de fichiers et répertoires (cf. section 13.3).

```
@liste=(1,2,3,4,5);
foreach (@liste)     <==>   foreach $_ (@liste)
{                    <==>   {
  print ;            <==>     print $_;
}                    <==>   }
```

1. Mais **printf** n'utilise pas la variable **$_** par défaut.

Chapitre 6

Premiers exercices

6.1 Exercice numéro 1

```
#!/usr/bin/perl
#  TP1
#  Lecture de nombres, pour chacun d'entre eux on indique
#  s'il est pair ou impair. À l'issue de la saisie (Ctrl-d
#  sous Unix ou Ctrl-z sous Windows) on affiche la moyenne
#  des nombres entrés.
print ("Entrer un nombre \n");
$n=0;
$total=0;
while(<STDIN>) {   # le résultat de la saisie dans $_
    chomp();       # enlève le délimiteur de fin de ligne
    if ($_ % 2) {  # rend 1 si impair et 0 sinon
        print ("$_ est impair \n");
    }
    else {
        print ("$_ est pair \n");
    }
    $n++;  $total += $_;
    print ("Entrer un nombre \n");
}
if ($n) {
    print ("La moyenne est : ",$total / $n,"\n");
}
```

6.2 Exercice numéro 2

```
#!/usr/bin/perl


# TP2
#
# Constituer une liste de mots entrés au clavier l'un après
# l'autre.
# Trier la liste constituée par ordre croissant puis par
# ordre décroissant.
#


print ("Entrer un mot \n");
while (<STDIN>) {
    chomp();
    push (@liste, $_);
    print ("Entrer un mot \n");
}

if (@liste) {
    @tri=sort(@liste);
    @rev=reverse(@tri);
    print("Par ordre croissant : \n");
    print ("@tri \n");
    print ("Par ordre decroissant : \n");
    print ("@rev \n");
}
```

6.3 Exercice numéro 3

```
#!/usr/bin/perl


# TP3
#
# On utilise un Hash pour gérer un stock. Le nom d'un
# vin est utilisé comme clef, une valeur saisie au clavier
# vient en + ou - de l'éventuelle valeur deja existante.
#


%Stock=();

print ("Nom du vin : \n");
while ($Vin=<STDIN>) { #Vin sera la clef du Hash de stock.

  chomp($Vin);         # Enlève le car de fin de ligne.
  print ("Quantité : ");
  $Quant=<STDIN>;      # Il faudrait vérifier qu'il s'agit
                       # bien d'un nombre entier relatif.
  chomp($Quant);
  unless ($Quant) {# Si l'utilisateur ne saisit pas de
     last;          # quantité, on affiche le stock.
  }
  $Stock{$Vin} += $Quant;
  print ("Nom du vin : \n");
}

# Parcourir le stock
while (($Vin,$Quant) = each(%Stock)) {
  print ($Vin,"=",$Quant,"\n");
}
```

Chapitre 7

Les expressions régulières

Perl est très adapté au traitement des chaînes de caractères car il permet de spécifier des masques qui peuvent être utilisés pour rechercher des occurrences de séquences de caractères qui leurs correspondent. Les masques sont appelés **expressions régulières**, ils sont familiers des utilisateurs des commandes Unix telles que **ed**, **sed**, **awk** ...

7.1 L'opérateur de recherche d'occurrences d'expressions régulières

En plaçant une expression régulière entre *slashs* (/ *expr* /), on recherche son existence éventuelle dans la variable **$_** (de la gauche vers la droite). L'opérateur de recherche retourne **vrai** si l'expression régulière trouve une correspondance (et retourne **faux** dans le cas contraire). De plus l'opérateur de recherche s'il retourne **vrai** positionne les variables suivantes :

- $& contient le sous ensemble de **$_** qui correspond à l'expression régulière ;
- $` contient le sous ensemble de **$_** qui se trouve avant $& ;
- $' contient le sous ensemble de **$_** qui se trouve après $&.

```
$_ = "Il fait beau";
if (/fait/) {
    print ($&,$',$`,"\n"); # <==> $& = 'fait', $`='Il '
                           # <==> $' = ' beau'
  }
```

7.1.1 Recherche sur une variable quelconque

Lorsque la variable à osculter n'est pas $_, on utilise l'opérateur =~ :

```
$var = "Il fait beau";
if ($var =~ /fait/)
  {
    print ($&,$',$`,"\n"); #  <==> $& = 'fait', $`='Il '
                           #  <==> $' = ' beau'
  }
```

7.2 Construction des expressions régulières

7.2.1 Les sélecteurs de caractères

Des sélecteurs de caractères sont utilisés pour construire les expressions régulières. Les plus simples recherchent l'existence d'un caractère, ils peuvent être répétés pour construire des expressions plus compliquées :

```
.         recherche n'importe quel caractère (sauf le
          changement de ligne)

[abc]     recherche un caractère parmi ceux situés entre
          les accolades (classe de caractères)

[^abc]    recherche un caractère qui ne soit pas un de ceux
          situés entre les accolades

[a-z]     recherche un caractère dont le code ASCII est
          situé entre le 1er et le 2ème caractère cité

ab.[a-z]  dans cet exemple, on recherche un a, suivi d'un b,
          suivi d'un caractère quelconque, suivi d'une
          lettre minuscule
```

Il existe des classes de caractères prédéfinies :

- (**\d**) correspond à un chiffre donc à **[0-9]** ;
- (**\w**) correspond à une lettre (plus le caractère souligné !) donc à [a-zA-Z0-9_] ;
- (**\s**) correspond aux caractères de séparation usuels (espace, tabulation, retour charriot, nouvelle ligne, saut de page).

Ces 3 classes de caractères ont une construction négative **D**, **W**, **S** signifiant respectivement [∧0-9], [∧a-zA-Z0-9_] et [∧ \r\t\n\f].

7.2.2 Les multiplicateurs de sélecteurs

Des multiplicateurs peuvent être utilisés pour spécifier les expressions régulières :

- ***** indique 0 ou plus de caractères identiques à celui placé à gauche ;
- **+** indique 1 ou plus de caractères identiques à celui placé à gauche ;
- **?** indique 0 ou 1 caractère identique à celui placé à gauche.

chaîne source	expr. reg	V/F	contenu de $&
'xxxF'	x?	V	x
'xxxF'	x*	V	xxx
"n'importe quoi"	.*	V	"n'importe quoi"
'abcdefg'	a.+d	V	'abcd'
'aabbbcdde'	a+b+	V	'aabbb'
'aabbbcdde'	a+c	F	

Il faut manipuler avec précaution le multiplicateur ***** car il signifie 0 ou n occurrences (0 inclus évidemment) et de plus, comme le multiplicateur **+**, il est glouton, c'est à dire que s'il y a correspondance, il retournera la plus longue chaîne possible.

chaîne source	expr. reg	V/F	contenu de $&
'xxxF'	x*	V	xxx
'abcxxxF'	x*	V	chaîne vide
'abcxxxF'	abcx*	V	abcxxx

Dans l'exemple ci-dessus, la chaîne est parcourue de la gauche vers la droite, dans le premier cas l'opérateur glouton retourne xxx, dans le second cas il retourne 0x (donc une chaîne vide), le fonctionnement du 3ème cas est analogue au premier.

Pour inhiber le fonctionnement glouton d'un multiplicateur, on le fait suivre d'un **?** qui signifie : le nombre minimum de caractères correspondant au sélecteur.

chaîne source	expr. reg	V/F	contenu de $&
'xxalloallo '	a.*o	V	'alloallo'
'xxalloallo '	a.*?o	V	'allo'
'xyzaaaxyz'	xyza*	V	'xyzaaa'
'xyzaaaxyz'	xyza*?	V	'xyz'

7.2.3 Mise en mémoire d'une sélection partielle

Il est possible dans une même expression régulière d'utiliser comme sélecteur le résultat d'une sélection déjà effectuée, elle est mise en mémoire et numérotée au préalable par l'opérateur (). L'opérateur *numéro* sert à référencer une mise en mémoire :

```
if (/<(.*)>.*<\/\1>/) printf ("balise : $& \n");
```

L'exemple ci-dessus[1] peut servir à rechercher certaines balises d'un texte HTML. Les parenthèses autour du premier .* indiquent qu'il faut stocker le résultat de la sélection, le \1 fait référence à cette sélection.

```
'<H1> xxxx </H1>' ou '<EM> xxxxx </EM>'
 correspondent à l'expression régulière
 donnée ci-dessus, mais
'<A HREF="xxxx"> ancre </A>' ne le fait pas !
```

1. recherche n'importe quelle chaîne de caractères située entre <> suivie d'un texte (éventuellement vide), suivi entre <> de la chaîne trouvée initialement précédée d'un /

L'opérateur () est également utilisé pour appliquer un multiplicateur au résultat d'une sélection :

```
if (/ab(cde)+/) {
   # vrai pour abcde, abcdecde, abcdecdecde, ...
   # le multiplicateur + s'applique au résultat de
   # la sélection entre parenthèses.
}
```

7.2.4 La sélection alternative

L'opérateur | est utilisé pour marquer une sélection avec alternative :

```
/abc|def/ sélectionne des chaînes de caractères contenant
          consécutivement les caractères abc ou def
```

7.2.5 Balisage des frontières de sélection

Des opérateurs particuliers permettent de spécifier des frontières auquelles s'appliquent les sélecteurs :

- **b** indique que le sélecteur précédent ne s'applique que sur une frontière de mot ;
- **B** indique que le sélecteur précédent ne s'applique pas sur une frontière de mot ;
- ∧ marque le début d'une ligne ;
- **$** marque la fin d'une ligne.

```
^http       <==> chaîne commençant par la séquence http
[^http]     <==> chaîne ne comportant ni h, ni t, ni t, ni p
\^http      <==> chaîne contenant la suite de caractères
                 ^http
xxx$        <==> chaîne terminant par xxx
xxx\$       <==> chaîne contenant la suite de caractères
                 xxx$
\bpays\b    <==> chaîne contenant le mot pays, paysage ou
                 paysan ne correspondent pas
```

7.2.6 sélection d'un nombre exact d'occurrences

Il est possible de sélectionner un nombre exact d'occurrences en utilisant l'opérateur {} :

```
/\d{1,5}/     recherche un nombre composé de 1 à 5 chiffres
              (au moins 1  et au plus 5)
/x{3}/        recherche exactement 3 x
/x{3,}        recherche une suite d'au moins 3 x (la borne
              supérieure n'est pas précisée)
```

7.2.7 Priorité des opérateurs de construction d'expressions régulières

Comme pour rédaction d'expressions arithmétiques, il peut y avoir des ambiguités dans l'écriture des expressions régulières :

```
[a-z]|[A-Z]+   veut il dire 1 ou n occurrences d'une lettre
               minuscule ou d'une lettre majuscule
                           ou plutôt
               une lettre minuscule ou 1 ou n occurrences
               d'une lettre majuscule
```

Des règles de priorités existent, en en tenant compte dans l'exemple ci-dessus, on a programmé le deuxième cas (le | est moins prioritaire que le plus). Pour éviter de connaître précisemment les règles de priorités des opérateurs, il suffit de noter que l'opérateur () est le plus prioritaire, pour programmer à coup sûr le cas numéro 1, on pourrait écrire :

```
([a-z]|[A-Z])+   <==> la mise entre parenthèses de
l'alternative assure qu'elle sera prioritaire sur le +
```

7.3 L'opérateur de substitution

Pour effectuer des substitutions correspondant à des expressions régulières, on utilise l'opérateur **s/expr-reg/chaine/**, qui par défaut utilise et affecte la variable **$_**.

```
$_ = '123456789';
s/123/abc/;                <==> $_ = 'abc456789'
$var = 'abctototototodef';
$var =~ s/(to)+//;         <==> $var = 'abcdef'
```

Les variables **$&**, **$'** et **$`** conservent le même sens qu'avec l'opérateur de recherche, la chaîne supprimée est contenue dans $& ...

Lorsque la chaîne à traiter n'est pas contenue dans la variable **$_**, on utilise l'opérateur de recherche =~ qui combiné avec l'opérateur de substitution, utilise et affecte la variable citée.

L'opérateur de substitution s'arrête sur la première occurrence de la chaîne correspondant à l'expression régulière donnée. Pour effectuer une substitution de toutes les occurrences, il convient de l'indiquer en rajoutant un **g** (global) derrière le dernier / de l'opérateur de substitution :

```
$var = 'aa123aaa456aaaa789';
$var =~ s/a+/ /g;          <==> $var = ' 123 456 789'
```

Il peut être intéressant de référencer la chaîne correspondant à une expression régulière dans une opération de substitution :

```
while ($ligne = <STDIN>)
{  $ligne =~ s/(images|applet)/$1\/monprojet/g;  }
```

L'exemple précédent peut servir à rajouter un répertoire (monprojet ici) dans les références à des images ou des applets à l'intérieur d'un document HTML.

7.4 Ignorer la casse

Il est possible d'indiquer aux opérateurs de recherche et de substitution d'ignorer la casse (de traiter indifféremment les lettres majuscules et minuscules) en utilisant le symbole **i** (ignore) :

```
s/begin/debut/i   remplace begin, Begin, BEGIN .... par
                  la chaine "debut"
```

7.5 Début de ligne, fin de ligne, ...

Les frontières de sélection ∧ et **$** marquent le début et la fin de ligne. Si la variable à traiter (par les opérateurs de recherche ou de substitution) contient plusieurs lignes (un scalaire contenant la totalité d'un fichier par exemple), il convient de préciser si **$** marquera la fin du fichier ou succéssivement la fin de chaque ligne ? On utilise pour celà les modifieurs **m** et **s** :

- **m** indique que la variable scalaire à traiter est considérée comme multi-ligne. ∧ et **$** sélectionnent un début ou une fin de ligne à n'importe quel emplacement dans la variable à traiter (et non plus le 1er et le dernier caractère) ;
- **s** indique que la variable à traiter est considérée comme ne comportant qu'une seule ligne (un '.' sélectionne alors n'importe quel caractère, le délimiteur de fin de ligne y compris). Cette dernière fonctionnalité peut être utile pour isoler une partie de texte comportant plusieurs lignes (cf; exemple suivant).

```
# Si la variable $html contient la totalité d'un fichier
# HTML alors la variable $body contiendra le corps
# du document :

$body = $html;
$body =~ s/(.*)<body>(.*)<\/body>/\2/s;
```

7.6 Travailler sur les champs d'une ligne

7.6.1 split

Perl traite les flux de données comme une suite de lignes composées de champs séparés par les séparateurs habituels (espace, tabulation).

La fonction **split** découpe une ligne en champs en utilisant comme délimiteur une expression régulière, toutes les parties de la ligne qui ne correspondent pas à l'expression régulière sont stockées dans une liste.

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

while ($ligne=<STDIN>)
{
 @champ = split (/:/,$ligne);
 printf ("User: %s Sh: %s \n",$champ[0],$champ[$#champ]);
}
```

Le programme ci-dessus examine le contenu d'un fichier style */etc/passwd* (dont les champs sont séparés par le caractère **:**) et isole les noms d'usagers (le premier champ) ainsi que leurs interpréteurs de commandes favori (le dernier champ, sous Unix bien sûr).

split travaille implicitement sur la variable **$_**, et on aurait donc pu écrire l'exemple précédent sous la forme suivante :

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

while (<STDIN>)
{
 @champ = split (/:/);
 printf ("User: %s Sh: %s \n",$champ[0],$champ[$#champ]);
}
```

S'il s'agit d'utiliser les espaces ou les tabulations comme délimiteur, on peut tout simplement omettre l'expression régulière. Si on travaillle sur le variable **$_**, on peut alors

écrire :

```
while (<STDIN>)
{
 @champ = split();    # ou même @champ=split;
}
```

7.6.2 join

join permet l'opération inverse d'un **split** ie. recoller les morceaux.

```
$ligne = join(':',@champ);  <==> la variable  $ligne reçoit
    la liste @champ avec insertion d'un séparateur (: ici).
```

Chapitre 8

Exercices sur les expressions régulières

8.1 Exercice numéro 4

```
#!/usr/bin/perl
#  TP4
#  On vérifie que des chaines saisies peuvent correspondre
#  à une immatriculation de véhicule en france.
#  On vérifie donc que la chaine saisie
#  commence par un chiffre compris entre 1 et 9
#  éventuellement suivi de 4 chiffres, suivi de une, deux
#  ou trois lettres, suivi de
#      - le chiffre 0 suivi d'un chiffre ou
#      - un chiffre suivi d'un chiffre ou
#      - 2a ou 2b
print ("Entrer une immatriculation de véhicule : \n");
while (<STDIN>) {
  # chomp(); n'est pas obligatoire car $ matche la fin de la ligne
  # ou juste avant le retour charriot s'il y en a un ...
    if (/^[1-9][0-9]{0,4}[a-zA-Z]{1,3}(0[1-9]|[1-9][0-9]|2a|2b)$/) {
        print ("C'est bon $& \n");
    }
    else { print ("C'est faux $& \n");}
    print ("Entrer une immatriculation de véhicule : \n");
}
```

8.2 Exercice numéro 5

```
#!/usr/bin/perl


# TP5
#

#   Calcul d'expressions arithmétiques simples (non parenthésées)
#   saisies à la volée.
#
#   La chaine doit commencer par un nombre
#   suivi d'un opérateur arithmétique suivi d'un nombre (n fois).

#   On admet que eval(expr) fait le calcul.


print ("Entrer une expression à calculer \n");
while (<STDIN>) {

    if (/^\d+([+*\/-]\d+)+$/) {
        # si le - situé entre [] n'était pas positionné
        # juste avant ], il faudrait le précéder d'un
        # / car il serait alors pris comme démimiteur de
        # classe (cf. [a-z]).
        print ($_," = ",eval($_),"\n");
    }
    else {
        print ("Expression arithmétique incorrecte \n");
    }
    print ("Entrer une expression à calculer \n");
}
```

8.3 Exercice numéro 6

```
#!/usr/bin/perl


#  TP6
#
#    On vérifie que des nombres entrés au clavier sont
#    bien des entiers relatifs ou des réels.
#

print ("Entrer un nombre : \n");
while (<STDIN>) {

    chomp();
    if (/^[+|-]?\d+$/) {
        #   commence  éventuellement par un + ou un -
        #   suivi et terminé par au moins un chiffre.
        print ("$_ est un nombre entier relatif \n");
    }
    elsif (/^[+-]?\d+\.\d+$/) {
         #   commence éventuellement par un + ou un -
         #   suivi par au moins un chiffre suivi par
         #   un . suivi et terminé par au moins un
         #   chiffre.
        print ("$_ est un nombre réel \n");
    }
    else {
        print ("$_ n'est pas un nombre \n");
    }
    print ("Entrer un nombre : \n");
}
```

8.4 Exercice numéro 7

```
#!/usr/bin/perl

# TP7
#
# On entre un texte à la volée.
# Lorsque la saisie est terminée, on calcule le nombre de lignes,
# de mots et de caractères du texte.
#

printf ("Entrer un texte libre sur plusieurs lignes \n");
@texte=<STDIN>;
chomp(@texte);

$nbmots = 0;
$nblig = 0;
$nbcar = 0;

foreach (@texte) {
  $nblig++;
  @mots=split();
  $nbmots = $nbmots + @mots;
  foreach (@mots) {
    $nbcar += length();    # length($-)
  }
}

print ("Nb de lignes saisies : $nblig \n");
print ("Nb de mots saisis : $nbmots \n");
print ("Nb de caractères saisis : $nbcar \n");
```

Chapitre 9

Quelques trucs utiles

9.1 Exécuter des commandes

Une commande placée entre accents graves est soumise par Perl à l'interpréteur de commande du système, le résultat peut facilement être récupéré dans une variable :

```
@who = ‘\usr\bin\who‘;
foreach (@who) {
   @User=split(/\s+/);
   printf ("User : %s \n", $User[0]);
}
```

L'exemple ci-dessus exécute la commande Unix *who* et affiche le nom des usagers (le premier champ des lignes retournées par *who*). C'est la sortie STDOUT qui est redirigée dans la variable spécifiée. Pour rediriger également les erreurs éventuelles, on pourrait écrire (sous Unix) :

```
$rm = ‘/bin/rm fichier 2>&1‘;
if ($rm) { printf ("Erreur ---> %s \n", $rm);}
```

9.2 La fonction die

La fonction **die** arrête l'exécution d'un programme en affichant une liste passée en argument.

```
die ("c'est fini !\n");
```

On l'utilise généralement avec l'instruction de contrôle || pour terminer l'exécution d'un programme si l'instruction précédente se passe mal (ouvertures de fichiers, ...) ou si un problème survient dans la logique de déroulement :

```
($chaine) || die ("La chaîne à traiter est vide \n");
```

En indiquant **$!** dans les paramètres passés à la fonction **die**, on affiche le code retour des fonctions systèmes :

```
open (FIC,"MonFichier") || die ("Pb ouverture : $! \n");
```

9.3 Évaluation à la volée

Perl permet d'évaluer à l'exécution des variables contenant du code. Cette fonctionnalité permet de construire dynamiquement des programmes ou morceaux de programmes. On utilise pour celà l'instruction **eval** qui va passer à Perl une chaîne à évaluer :

```
$var = '$som = $val1 + $val2';
eval $var;   # <==> à l'exécution, le contenu de $var est
                   passé à Perl qui exécutera son contenu
```

Il convient d'être prudent en utilisant **eval**, ne pas demander à l'utilisateur de rentrer une chaîne de caractères et la faire ensuite évaluer ainsi, il pourrait par exemple avoir accès à l'interpréteur de commandes et saisir '/bin/rm * ' ...

Perl traite les instructions trouvées dans une variable passée à **eval** comme un bloc (ie. comme si elles étaient délimitées par {}) pouvant contenir des variables locales (cf. section 10).

L'instruction **eval** est également utilisée pour traiter les exceptions d'exécution d'un programme Perl, dans ce cas le contenu de la variable à évaluer doit explicitement être écrit comme un bloc complet (avec les accolades de début et de fin).

9.4 Les arguments de la ligne de commande

Les arguments de la ligne de commande sont accessibles par l'intermédiaire du tableau **@ARGV** qui contient un élément par arguments passés au programme. Contrairement au langage C, le premier élément du tableau **@ARGV** ne contient pas le nom de la commande, **$ARGV[0]** contient le premier argument passé à la commande.

```
foreach (@ARGV) {
    printf ("Arg = %s \n",$_);
}
```

Chapitre 10

La portée des variables

Par défaut une variable Perl est globale, elle est donc *visible* dans l'ensemble du programme. De plus Perl n'oblige pas à déclarer les variables avant de les utiliser. Ces deux propriétés sont sources de bien des problèmes de mise au point, Perl permet donc de déclarer des variables locales et d'utiliser un mécanisme obligeant le programmeur à déclarer les variables avant de les utiliser.

Les programmeurs utilisant les langages de programmation les plus courants sont en général habitués à un fonctionnement différent : les variables sont automatiquement locales aux blocs de déclarations, pour les rendre globales, il est nécessaire de le préciser.

10.1 Déclarer des variables locales avec my

Il est possible de rendre locale une variable en précédent sa déclaration par **my**, elle ne sera alors connue que du bloc ou de la fonction (cf. section 11) qui contient sa déclaration. Les variables locales sont par défaut initialisées à **undef** [1] et, il n'existe pas de variables locales statiques (celles dont la visibilité est limitée au bloc de déclaration mais dont la valeur est remanente d'une entrée dans le bloc à une autre).

1. On peut les comparer aux variables automatiques de C.

```
#!/usr/local/bin/perl
$I = 10;
{
   my $I = 2;
   {
      $I++;
      {
         my $I = 4;
         printf ("I = %d \n",$I);  <==> Affiche I = 4
      }
      printf ("I = %d \n",$I);   <==> Affiche I = 3
   }
   printf ("I = %d \n",$I);    <==> Affiche I = 3
}
printf ("I = %d \n",$I);     <==> Affiche I = 10
```

10.2 Une autre manière de déclarer des variables locales

Les variables locales déclarées avec **my** ne sont visibles que dans leurs blocs de déclarations. Perl propose une autre variété de variables locales : celles qui en plus ont la particularité d'être visibles dans toutes les fonctions (cf. section 11) appelées depuis leurs blocs de déclarations.

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

$I = 4;            <==> cet I est global
{
  local $I = 2;
  fonc();          <==> Affiche I = 2
}
{
  my $I = 3;
  fonc();          <==> Affiche I = 4
}
sub fonc {printf("I = %d \n", $I);}
```

Les variables déclarées avec **local** ne contribuent pas à la lisibilité d'un code, elles peuvent même être source de confusions, il convient de ne pas les utiliser.

10.3 use strict

strict est un module de la bibliothèque standard de Perl, il permet de générer une erreur à la compilation si une variable accédée n'a pas été préalablement déclarée avec **my** (ou complètement qualifiée, ou importée cf. section 16).

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

my $Compteur=10;

$compteur++;                # erreur de frappe
print ('$Compteur = ',"$Compteur \n");

#     La valeur affichée est 10, le programmeur
#     peut ne pas s'en apercevoir.
```

```
#!/usr/local/bin/perl
use strict;

my $Compteur=10;

$compteur++;                # erreur de frappe
print ('$Compteur = ',"$Compteur \n");

# Un message d'erreur est généré à la compilation :
# Global symbol "$compteur" requires explicit package name
```

Dans l'exemple ci-dessus l'erreur de saisie est signalée car Perl n'autorise plus (**use strict**) les déclarations implicites de variables globales. Toutes les variables doivent être déclarées à l'aide de **my** (elles sont locales au programme principal) ou déclaréees dans un module externe (cf. section 16). L'utilisation de **use strict** est donc très fortement recommandée.

Chapitre 11

Les fonctions

11.1 Définition d'une fonction

Perl permet d'écrire des fonctions qui admettent ou pas des arguments et qui rendent ou pas une valeur de retour. Elles sont définies par le mot-clef **sub** suivi d'un identificateur et d'un bloc qui va contenir le code de la fonction.

```
sub Ma_Fonction {
   instruction 1;
   instruction 2;
}
```

Les fonctions peuvent être placées n'importe où dans un source Perl (elles sont sautées à l'exécution), il est toutefois conseillé de les placer en début ou en fin de programme. La portée des fonctions est globale, il est possible d'enliser une fonction dans une autre mais celà ne restreint pas sa visibilité, cette forme d'écriture n'a donc pas de sens.

```
sub Ma_Fonction {
   instruction 1;
   instruction 2;
sub Fonction_Enlisee { ... }  <==> Fonction_Enlisee est
    tout de même accessible en dehors de Ma_Fonction
}
```

Si une fonction retourne une valeur elle est précisée par l'instruction **return** suivi d'une expression à évaluer. Les arguments d'une fonction sont passés par valeurs et récupérés dans le tableau **@_** (arobas souligné), en fonction du cas posé diverses solutions sont possibles pour y avoir accès :

```
my ($var1,$var2,$var3) = @_;
# ou
my $var1 = $_[0];
my $var2 = $_[1];
my $var3 = $_[2];
# ou encore
foreach (@_) { .... }
```

Dans les faits une fonction retourne toujours une valeur, c'est celle de la dernière expression évaluée avant la sortie. On peut donc se passer de l'instruction **return** mais sa présence ne nuit pas à lisibilité du code !

11.2 Appel d'une fonction

On appelle une fonction en indiquant son identificateur suivi entre paranthèses des arguments (éventuellement 0). L'appel à une fonction peut être placé dans une expression en fonction de sa valeur de retour :

```
#!/usr/local/bin/perl
$I = 10;
$J = 20;
resultat();  <==> appel d'une fonction qui ne retourne pas
                  de valeur (ou plus exactement dont la
                  valeur de retour est ignorée ici)

sub resultat {
 printf ("La somme est : %d \n",som($I,$J));
}
sub som {
 my ($a,$b) = @_;
 return $a + $b;
}
```

11.3 Prototyper les fonctions

Perl permet de vérifier (à la compilation) que les arguments passés à une fonction correspondent bien à ce qui est prévu par le programmeur. On utilise pour celà le prototypage des fonctions utilisées (qui doivent être *visibles* à la compilation). Le prototypage consiste à indiquer le nombre d'arguments attendus en précisant leur type. En Perl le prototype suit la déclaration du nom de la fonction (entre parenthèses) et utilise les conventions suivantes :

- un **$** indique la présence d'un scalaire;
- un **@** indique la présence d'une liste;
- un **%** indique la présence d'un tableau associatif;
- un **;** sépare les arguments obligatoires des arguments optionnels;
- lorsque le caractère \ précéde un **@** ou un **%** il rend obligatoire que l'argument correspondant soit effectivement une liste ou un tableau associatif (cf. l'exemple suivant).

Il faut faire attention à l'utilisation des **@** et **%** dans les prototypes car ils *mangent* les caractères suivants et forcent un contexte de liste. L'utilisation du \ devant ces caractères est donc très importante (cf l'exemple suivant).

```
#!/usr/bin/perl

# prototypes des fonctions utilisées
sub fonc1($$$);  # 3 scalaires
sub fonc2($$@);  # 2 scalaires et une liste
sub fonc3($$\@); # 2 scalaires et une variable de type liste

@liste = (1, 2, 3);
print (fonc1(@liste,5,6)," ",fonc2(5,6,7), fonc3(5,6,7));
#
# ici l'appel de fonc1 rendra 14 car @liste dans un contexte
# scalaire vaut 3 (nombre d'éléments). L'appel de fonc2
# rendra 18 car 7 est pris comme une liste de 1 élément.
# En l'état ce programme ne compile pas car l'appel à fonc3
# est incorrect, le 3ème argument n'est pas une variable de
# type liste.
#

sub fonc1($$$) {
###############
   my ($a, $b, $c) = @_;
   return($a+$b+$c);
}
sub fonc2($$@) {
###############
   my ($a, $b, @liste) = @_;
   my $res;
   foreach (@liste) {
     $res += $_;
   }
   return($a+$b+$res);
 }
sub fonc3($$\@) {
###############
   my ($a, $b, @liste) = @_;
   my $res;
   foreach (@liste) {
     $res += $_;
   }
   return($a+$b+$res);
 }
```

11.4 Passer des arguments par références

Les arguments d'une fonction sont récupérés dans le tableau **@_** , ce mécanisme est suffisant pour passer quelques valeurs scalaires, mais devient très difficile à gérer s'il est nécessaire de passer plusieurs tableaux en paramètres (comment les délimiter entre eux ?). Pour résoudre ce problème, Perl propose d'utiliser des références plutôt que des valeurs.

11.4.1 Les références de variables

On référence une variable Perl en précédant son identificateur d'un \ :

```
$scalaire = 4;
@tableau = (1,2,3,4,5,6);

$refscal = \$scalaire;
$reftab  = \@tableau;
$refhac  = \%hache;
```

Une reférence est un variable de type scalaire, on peut donc corriger ce qui a été indiqué en section 2.1, un scalaire est un nombre, une chaîne de caractère ou **une référence**.

Pour déréférencer une variable contenant une référence, on précéde son identificateur d'un **$**,**@** ou **%** en fonction du type de données qu'elle référence.

```
$scalaire = 4;
@tableau = (1,2,3,4,5,6);
%hache = ("a",1,"b",2,"c",3);

$refscal = \$scalaire;
$reftab  = \@tableau;
$refhac  = \%hache;

$$refscal = 5;            <==> $scalaire = 5;
@$reftab = (7,8,9)        <==> @tableau = (7,8,9);
$$reftab[0] = 2;          <==> $tableau[0] = 2;
%$refhac = ("d",4,"e",5,"f",6);
           <==> %hache = ("d",4,"e",5,"f",6);
```

Pour déréférencer une variable contenant une référence sur un tableau ou un tableau associatif, on dispose d'une notation fléchée qui est plus confortable à utiliser que la notation utilisant le $$:

```
@tableau = (1,2,3,4,5,6,7);
%hache = ('bleu',1,'rouge',2,'vert',3);
$reftab = \@tableau;
$refhache = \%hache;

$reftab->[1] = 10; <==> $$reftab[1] = 10;
$refhache->{'rouge'} = 4; <==> $$refhache{'rouge'} = 4;
```

Il est également possible de créer des références des tableaux anonymes ou des hashes anonymes. Cette forme d'écriture est utilisée pour manipuler des structures de données complexes (tableaux de tableaux, tableaux de hashes ... cf. section 14).

Une référence sur un tableau anonyme se crée en utilisant des crochets (**[]**) :

```
@tableau = (1,2,3,4);
$reftab = \@tableau;
#  peut aussi s'écrire
$reftab = [1,2,3,4];
#  l'accès aux éléments est inchangé
$reftab->[1] ou $$reftab[1]
```

Une référence sur un hash anonyme se crée en utilisant des accolades {} :

```
%hache = ('bleu',1,'rouge',2,'vert',3);
$refhache = \%hache;
#  peut aussi s'écrire
$refhache={'bleu',1,'rouge',2','vert',3};
# l'accès aux éléments est inchangé
$refhache->{'rouge'} = 4;
```

Les références anonymes en Perl sont très pratiques à utiliser car le programmeur (contrairement au langage C par exemple) n'a pas à se soucier de la gestion de la mémoire (taille demandée et libération). Perl gère un compteur des références à chaque valeurs, qu'elles soient référencées directement ou pas, celles qui ne sont plus référencées sont détruites automatiquement.

11.4.2 Un exemple

Dans l'exemple suivant on désire réaliser une fonction recevant deux listes en arguments (@tab1 et @tab2), elle va rendre une liste résultante contenant la première liste triée suivie de la seconde également triée. On utilise ici un passage des arguments par références car on utilise deux tableaux, par valeurs la variable **@_** ne pourrait pas nous renseigner sur la limite du tableau @tab1. Par références on peut passer un nombre très important de tableaux car les références sont des scalaires.

```
#!/usr/bin/perl

@tab1 = (7,3,2,8,9,1,2);
@tab2 = (3,2,1,4);
# Appel de la fonction tri avec deux arguments
# qui sont des références sur les tableaux.
@ltri = tri(\@tab1, \@tab2);
# Affichage du résultat.
print (@ltri);

# Fonction tri
sub tri {
#########
   #  Récupération des arguments dans 2 variables
   #  locales qui contiennent des références sur
   #  les tableaux @tab1 et @tab2.
   my ($reftab1, $reftab2) = @_;
   my (@tri1, @tri2);
   #  Tri de chaque tableau, on déréférence pour
   #  pouvoir appeler la fonction sort().
   @tri1=sort(@$reftab1);
   @tri2=sort(@$reftab2);
   #  Retour du résultat
   push(@tri1,@tri2);
   return @tri1;  # retour de @tri1 (par valeur)
}
```

11.5 Exercice numéro 8

```
#!/usr/bin/perl
# TP8
# On reprend ici le TP3 mais à la fin de la saisie on appelle un
# fonction qui va calculer la valeur du stock.
# Elle reçoit en paramètre le hash du stock, un hash donnant le
# prix H.T. d'une bouteille de xxxx et le taux de TVA.
sub ValStock($$$);
%Stock=();
%Prix = ( 'Bordeaux'  => 25,
          'Bourgogne' => 40,
          'Bourgueil' => 33,
          'Bandol'    => 29);
$Tva=18.6;

print ("Nom du vin : \n");
while ($Vin=<STDIN>) {   # $Vin sera la clef du Hash de stock
    chomp($Vin);         # enlève le caractère de fin de ligne
    print ("Quantité : ");
    $Quant=<STDIN>;      # Il faudrait vérifier qu'il s'agit bien
                         # bien d'un nombre entier relatif.
    chomp($Quant);
    unless ($Quant) {    # Si l'utilisateur ne saisit pas de Quantité
        last;            # on passe à l'affichage du stock.
    }
    $Stock{$Vin} += $Quant;
    print ("Nom du vin : \n");
}
# On appelle la fonction ValStock en lui passant des références
# sur les hashs (sinon le premier mangerait le second).
print ("Valeur totale : ",ValStock(\%Stock, \%Prix, $Tva),"\n");

sub ValStock($$$) {
####################
    my ($StockPtr, $PrixPtr, $Tva) = @_;
    my ($Vin, $Quant, $Total);
    while (($Vin,$Quant) = each (%$StockPtr)) { # %$ pour déréférencer
        $Total +=  ($Quant * $PrixPtr->{$Vin});
    }
    return($Total + ($Total*$Tva/100) );
}
```

Chapitre 12

Accès au contenu des fichiers

En Perl, l'accès au contenu des fichiers s'effectue à travers des descripteurs de fichiers qui font le lien entre le programme et son environnement extérieur. Comme vu en section 4 tout programme Perl hérite de 3 descripteurs positionnés par le système d'exploitation : STDIN (entrée standard), STDOUT (sortie standard) et STDERR (erreurs produites par les appels au système).

12.1 Ouverture

Pour utiliser un fichier (autre que STDIN, STDOUT ou STDERR) il faut d'abord l'ouvrir (effectuer l'association entre un descripteur de fichier et le nom externe du fichier considéré). On utilise l'instruction **open** suivie d'un descripteur de fichier et du nom externe du fichier ciblé (éventuellement précédé du mode d'ouverture) :

```
open (FIC1,'monfichier');    # ouverture en lecture
open (FIC2,'c:\\tmp\\truc'); #
open (FIC3,'>toto');    # >  pour ouverture en ecriture
open (FIC3,'>>tutu');   # >> pour ouverture en ajout
open (FIC4,'+<tata');   # +< pour ouverture en lect/ecr
```

open retourne vrai si l'ouverture se passe bien, il est important de le vérifier. On peut utiliser un **die** conditionnel pour traiter les valeurs de retour de **open** :

```
open(FIC,'MonFichier') || die("Pb d'ouverture\n");
```

12.2 Lecture

L'opérateur <> permet de lire une ligne [1] dans le fichier désigné par le descripteur ciblé. Il rend vrai tant que la fin du fichier n'est pas atteinte :

```
open (FIC,'MonFic') || die ("Le fichier n'existe pas\n");
while (<FIC>)
 {
  # par défaut chaque ligne lue (y compris le délimiteur
  # de fin de ligne) est stockée dans $_
 }
```

L'opérateur <> est adapté à la lecture des fichiers de textes, pour lire des fichiers formatés différemment, on peut utiliser la fonction **sysread()** qui permet de préciser le nombre de caractères à lire ainsi que la variable scalaire qui va contenir les caractères lus. La fonction **sysread()** rend le nombre de caractères effectivement lus :

```
open (FIC,'MonFic') || die ("Le fichier n'existe pas\n");
while (($nb=sysread(FIC,$enr,100) != 0)
 {
  # on a lu $nb caractères (100 maxi) dans $enr
 }
```

1. On peut aussi lire entièrement un fichier dans une variable de type tableau (cf. section 4).

12.3 Ecriture

print et **printf** permettent d'écrire des lignes dans un fichier. **printf** (cf. section 4.2) permet de préciser le format des variables à écrire, print est plus rudimentaire.

```
print  FIC ($a,':',$b,':',$c,"\n");
printf FIC ("%.5s:%3d:%10s\n",$a,$b,$c);
```

print retourne **1** si l'écriture s'est bien déroulée et **0** en cas d'erreur.

Lorsque l'on écrit un fichier en Perl, il faut se poser la question de sa lecture ultérieure, et notamment celle la délimitation des champs. Dans l'exemple ci-dessus, il sera facile d'utiliser **split** (cf. 7.6.1) pour récupérer les variables[2].

Comme pour la lecture, il est possible d'écrire dans un fichier en précisant le une variable scalaire et le nombre de caractères à écrire, c'est le rôle de la fonction **syswrite()** :

```
$nb=syswrite(FIC,$enr,100);
# ecriture du contenu de la variable $enr à concurrence
# de 100 caractères maximum. $nb contient le nombre de
# caractères effectivement ecrits.
```

12.4 Fermeture

On ferme un fichier en précisant son descripteur à l'aide de la fonction **close**.

```
close(FIC);
```

2. ($a,$b,$c) = split (/:/);

12.5 Exercice numéro 9

```
#!/usr/bin/perl

#  TP9
#
#  Parcourir l'ensemble des TP réalisés et rajouter un commentaire
#  dans chacun d'entre eux ...
#  On choisit ici de remplacer la première ligne par elle même
#  suivie d'un commentaire constant.

use strict 'vars';

my $Head = "#!/usr/bin/perl";
my $Comment= "\n#\n# Effectué en formation PERL \n#";

my @fic = <*.pl>;       # obtenir la liste des tp.

foreach $prog (@fic) {
    unless (open (IN,$prog)) {
      print ("Erreur d'ouverture $prog ($!) \n");
    }
    unless (open (OUT, ">${prog}.bis")) {
     print ("Erreur création ${prog}.bis ($!) \n");
    }
    print ("Traitement de $prog ... \n");
    while (<IN>) {
        s/($Head)$/\1$Comment/;
        print OUT;
    }
    close(IN);
    close(OUT);
#   Si le programme est au point, on peut décommenter la ligne
#   suivante ...
#    rename("${prog}.bis",$prog);
}
```

Chapitre 13

Manipulations du système de gestion de fichiers

13.1 Accès aux répertoires

Perl permet d'accéder au contenu des répertoires d'un système de fichiers en utilisant la *convention **, ou en appelant des fonctions qui procurent une interface d'accès à leur structure.

13.1.1 Utiliser la convention *

Une expression placée entre les symboles < et > va solliciter le système de gestion de fichiers et obtenir une liste des fichiers ou répertoires qui lui correspondent (dans le répertoire de travail ou dans un répertoire cité par l'expression elle même) :

```
# accès aux programme c d'un répertoire
@progc = <*.c>;
# affichages du nom des fichiers sélectionnés
foreach (@progc) {printf ("%s \n", $_);}
```

Utilisée dans un contexte scalaire la convention * va parcourir le répertoire ciblé et rendre successivement les fichiers qui correspondent (plus **undef** quand il n'y a plus de corres-

pondance). On pourrait donc écrire l'exemple précédent de la façon suivante :

```
while (<*.c>) {printf("%s \n",$_);}
```

L'utilisation de la convention * peut s'effectuer en remplaçant les symboles < et > par la fonction **glob** :

```
@include=</usr/local/include/*.h>;
peut aussi s'écrire
@include=glob('/usr/local/include/*.h');
```

13.1.2 Utiliser une interface d'accès

Il est possible de parcourir un répertoire en utilisant la fonction **readdir** qui fonctionne à partir d'un descripteur de répertoire préalablement ouvert par **opendir**. Utilisée dans un contexte scalaire, **readdir** rend successivement le noms des fichiers du répertoire (plus **undef** quand il n'y en a plus). Utilisée dans un contexte de liste, **readdir** rend la liste de tous les fichiers du répertoire ciblé. La fonction **closedir** ferme un descripteur de répertoire ouvert par **opendir**.

```
opendir (DIR,'.') || die ('Erreur Open Dir');
@fic = readdir(DIR);
foreach (@fic) {printf ("%s \n",$_);}
```

13.2 Manipulation des fichiers et répertoires

Perl procure des fonctions pour réaliser les actions les plus usuelles sur les fichiers et répertoires.

13.2.1 Changer de répertoire de travail

La fonction **chdir** permet de changer le répertoire de travail de l'application en cours.

```
chdir('c:\\windows\\system') || die ("Erreur chdir \n");
```

13.2.2 Créer un répertoire

La fonction **mkdir** permet de créer un répertoire en positionnant des droits d'accès à la mode Unix [1]

```
mkdir ('MonRepert',0755) || die ("Err. Cr. répertoire \n");
```

13.2.3 Supprimer un répertoire

Un répertoire vide peut être supprimé par la fonction **rmdir**.

```
rmdir ('MonRepert') || die ("Err. Sup.  répertoire \n");
```

13.2.4 Supprimer un fichier

La fonction **unlink** permet de supprimer une liste de fichiers.

```
foreach (<*.old>) {
  unlink($_); || die ("Erreur suppression \n");
}
```

1. Une équivalence est faite automatiquement pour Win32.

13.2.5 Renommer un fichier

La fonction **rename** permet de changer le nom d'un fichier.

```
rename("log","log-old"); || die ("Pb. logs \n");
```

13.2.6 Modifier les droits d'accès

La fonction **chmod** permet de positionner les droits d'accès à un fichier à la manière d'Unix[2] en utilisant un codage octal[3].

```
chmod(0755,$fic); || die ("Err. droits d'accès \n");
```

13.3 Fonctions utiles

Il est possible d'obtenir des informations sur un fichier ou un répertoire en utilisant des expressions dont l'écriture est dérivée de celle de la commande Unix **test**. Ces expressions s'écrivent à l'aide d'un opérateur suivi d'une chaîne de caractères[4] qui représente le nom potentiel d'un fichier ou répertoire[5]. Les opérateurs (sauf exceptions) rendent une valeur vraie ou fausse, les principaux sont les suivants :

```
-r     fichier ou répertoire accessible en lecture
-w     fichier ou répertoire accessible en écriture
-e     fichier ou répertoire existant
-x     fichier exécutable
-z     fichier existant mais vide
-s     fichier ou répertoire non vide, la valeur retournée
       est la taille en octet
-f     fichier normal (répertoire et spéciaux exclus)
-d     répertoire
```

2. On procède par équivalence pour Win32.
3. La signification du codage est documentée dans le manuel Unix, voir chmod(1).
4. Elles fonctionnent également sur des descripteurs de fichiers.
5. En l'absence de chaîne de caractères et de descripteur de fichier, c'est la variable $_ qui sera utilisée.

```
$InstallDir = "MonAppli";
(-d $InstallDir) || die ("Appli non installée \n");
```

La fonction **stat** permet d'obtenir plus de détails sur un fichier matérialisé par une chaîne de caractères ou un descripteur de fichier. Elle donne accès aux différents champs d'une entrée dans un système de fichiers[6]. On a alors accès à la taille d'un fichier, à sa date de création ... **stat** renvoie une liste de 13 valeurs dans un tableau, la 8ème correspond à la taille du fichier.

L'exemple ci-dessous utilise la fonction **stat** pour déterminer la taille d'un fichier :

```
#!/usr/local/bin/perl
print ("Entrez un nom de fichier : \n");
while (<STDIN>)
{ chomp;
  (-f) ? print (taille($_),"\n") : print ($_," inconnu\n");
  print ("Entrez un nom de fichier : \n");
}
```

```
sub taille {
  my ($fic) = @_;
  my ($dev,$inode,$perm,$liens,$uid,$gid,
      $ndev,$lg,$acces,$mod,$cr,$blksize,$bl)=stat($fic);
  return($lg);
}
```

13.4 Exercice numéro 10

6. Une inode sous Unix.

```
#!/usr/bin/perl
#  TP10
#  Supprime le contenu d'un répertoire passé en paramètre.
#  L'utilisation d'une fonction récursive permet de détruire
#  les répertoires emboîtes.
#  On informe l'utilisateur du nombre de fichiers contenus dans
#  le répertoire qu'il souhaite détruire.
use strict 'vars';
sub HowManyFiles($);   # prototypes des fonctions utilisées
sub DeleteDir($);      # $ ==> un scalaire comme param.
unless (@ARGV) {
    print ("Usage : tp10 répertoire à détruire \n");
    exit();
}
$_ = $ARGV[0];
if (-d) {
    if (/^\/tmp/) {    # on se limite au répertoires 'temporaires'.
      my $nbf=HowManyFiles($_);
      if ($nbf) {
        print ("Ce répertoire contient $nbf fichiers. \n");
        print ("Voulez vous vraiment le détruire ? \n");
        my $nbf = <STDIN>;
        if ($nbf =~ /^oui$/i) {
          DeleteDir($_);
          print ("Répertoire supprimé !");
        }
        else {
          print ("Répertoire non supprimé !");
        }
      }
      else {
        DeleteDir($_);
        print ("Répertoire vide supprimé !");
      }
    }
    else {
        print ("N'est pas un repertoire temporaire \n");
    }
}
else {
    print ("$_ n'est pas un repertoire \n");
}
```

```
sub HowManyFiles ($) {
######################
#   Avant de supprimer un répertoire, on souhaite connaitre
#   le nombre de fichiers qu'il contient.
    my ($Dir) = @_;
    my (@fichiers,$fic); my $count = 0;
    opendir(DIR,$Dir) || return(0);
    @fichiers=readdir(DIR); # le rep est lu on peut le fermer ...
    closedir(DIR);          # @fichiers est local, c'est recursable
    foreach $fic (@fichiers) {
      if (($fic ne ".") && ($fic ne "..")) {
        if (-d "${Dir}/$fic") {
          $count+=HowManyFiles("${Dir}/$fic");
        }
        else { $count++; }
      }
    }
    return($count);
}
sub DeleteDir ($) {
####################
#  Suppression d'un répertoire (l'utilisation récursive permet de
#  détruire les éventuels enboîtés).
    my ($Dir) = @_;
    my (@fichiers,$fic);
    opendir(DIR,$Dir) || return(0);
    @fichiers=readdir(DIR); # le rep est lu on peut le fermer ...
    closedir(DIR);          # @fichiers est local, c'est recursable
    foreach $fic (@fichiers) {
        if (($fic ne ".") && ($fic ne "..")) {
            if (-d "${Dir}/$fic") {
                DeleteDir("${Dir}/$fic");
            }
            else {
                unlink("${Dir}/${fic}") || die ("erreur delete");
            }
        }
    }
    rmdir($Dir) || return(0);
    return(1);
}
```

Chapitre 14

Les structures de données complexes

Perl autorise de manipuler des structures de données plus complexes que celles accessibles par les 3 types de base. Des références sur des listes anonymes ou des hashs anonymes sont utilisées pour celà.

14.1 Les listes de listes

```
@Liste = (
       ['toto','tutu','tata'],
       ['truc','much'],
       ['patati','patata']
       );
```

Dans l'exemple ci-dessus, **@Liste** est une liste de références sur des tableaux anonymes (**[]**).

L'affectation d'un élément s'opère à la manière d'un tableau à deux dimensions. L'affectation d'une liste complète implique l'utilisation de références.

```
#  remplacer 'truc' par 'machin'
$Liste[1][0]='machin';
#  remplacer ['patati','patata'] par ['bla','blabla']
@AutreListe = ('bla','blabla');
$Liste[2] = \@AutreListe;
```

```
#  Erreur à ne pas commettre
$Liste[2] = @AutreListe;
#  qui charge la valeur 2 (le nombre d'éléments) en lieu
#  et place d'une référence sur une liste ...

#  Il n'est pas forcément nécessaire de déclarer une liste
#  pour assurer l'opération. On utilise ci-dessous une
#  référence sur une liste anonyme ([]).

$Liste[2] = ['bla','blabla'];

#  ajouter une liste
push(@Liste, ['en','voila','du','texte']);
```

L'accès à un élément s'effectue à la manière d'un tableau à deux dimensions. L'utilisation d'une des listes d'un tableau de listes dans une opération d'affectation nécessite un peu d'attention :

```
@AutreListe = @$Liste[2];
```

Cette écriture semble naturelle, **$Liste[2]** est une référence sur une liste, pour déréférencer on la précède du caractère **@**. En fait, pour des raisons de priorités entre opérateurs **$Liste** sera considéré comme une référence (qui n'existe pas), puis on fera le déréférencement et enfin on cherchera le 2ème élément. L'utilisation de **use strict** (cf. section 10.3) indiquerait d'ailleurs une erreur à la compilation !

Il faut donc écrire :

```
@AutreListe = @{$Liste[2]}
```

Les { et } sont utilisés pour clairement indiquer que la référence est $Liste[2].

14.2 Les hashes de hashes

```
%User = (
        toto => {
            Homedir => '/Home/toto',
            Passwd  => 'A1PefjWNa03H2',
            Email   => 'toto@ici.fr'
        },
        tutu => {
            Homedir => '/Home/tutu',
            Passwd  => '31PrfjWNa08Ha',
            Email   => 'tutu@ici.fr'
        },
        tata => {
            Homedir => '/Home/tata',
            Passwd  => '41aqfjer508Ha',
            Email   => 'tata@ici.fr'
        }
);
```

Dans l'exemple ci-dessus, **%User** est un hash de hashs anonymes ({}) (à une clef correspond une référence sur un hash anonyme).

```
# Changer une valeur
$User {'tutu'} {'Email'} = 'tutu@parla.fr';

# Ajouter une nouvelle clef
$User {'titi'} =  {HomeDir => '/home/titi',
                   Passwd  => '32drtyuoiXes',
                   Email   => 'titi@ici.fr'};

# Accéder à la totalité des éléments
foreach $util (keys %User) {
    foreach (keys %{$User{$util}}) {
        print ("$util->$_ : $User{$util}{$_} \n");
    }
}
```

Dans l'exemple ci-dessus on écrit **%**{**$User**{**$util**}} et non pas **%$User**{**$util**} pour les mêmes raisons qu'indiquées en section 14.1.

14.3 Autres structures de données

La construction de structures de données ne s'arrête pas aux listes de listes ou aux hashes de hashes. Il est possible de généraliser le procédé pour bâtir des structures hybrides mélant des listes et des hashes sur plusieurs niveaux. Pour bien comprendre le fonctionnement, il faut bien avoir à l'esprit qu'un tableau en Perl est toujours uni-dimensionnel. Un tableau à 4 dimensions ne sera en réalité qu'un tableau à une dimension contenant des références vers des tableaux contenant des références vers des tableaux contenant des références vers des tableaux.

Dans l'exemple suivant, on gère un tableau (à l'allure tri-dimensionnelle) de valeurs correspondant à des années, mois et jours. On commence par installer quelques valeurs dans le tableau avant de le parcourir entièrement pour calculer la somme de toutes les valeurs installées.

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

# Exemple d'utilisation d'un tableau à 3 dimensions.

@Valeurs = ();

# Initialisation du tableau

AddValeurs(1999,12,1,150);
AddValeurs(1999,12,1,250);
AddValeurs(1994,12,2,100);
AddValeurs(1999,12,3,500);
AddValeurs(1999,11,3,500);
AddValeurs(1999,11,8,500);
AddValeurs(1990,11,10,500);

$NbElem = @Valeurs;

# $NbElem vaut 2000 [0..1999], car Perl bouche
# les trous.

print ("Total Annuel : ",StatAn(1999),"\n");
```

```
sub AddValeurs {
################
    my ($an, $mois, $jour, $montant) = @_;

#   On utilise ici une écriture traditionnelle pour
#   un tableau à 3 dimensions.

    $Valeurs [$an] [$mois] [$jour] += $montant;
}


sub StatAn {
############
    my ($an) = @_;
    my $total;

    foreach  $mois (@{$Valeurs[$an]}) {

#   @{$Valeurs[$an]} est une liste de références
#     13 ici ([0..12]) car le plus grand mois cité
#     vaut 12 pour l'année 1999.
#   $mois contient une référence vers un tableau qui
#   contient les valeurs associées. Il y a ici 9
#   valeurs pour le 11éme mois de l'année 1999 et
#   4 valeurs ([0..3]) pour le 12ème mois de 1999.

        foreach $jour (@$mois) {
            $total += $jour;
        }
    }
    return($total);
}
```

On peut maintenant compléter l'exemple de la section 14.2 pour associer à chaque utilisateur une liste de machine sur lesquelles il est autorisé à se connecter. On obtient ainsi un hash de hash de liste.

```
%User = (
        toto => {
            Homedir => '/Home/toto',
            Passwd  => 'A1PefjWNa03H2',
            Email   => 'toto@ici.fr',
            Login   => ['mach1','mach2','mach3']
        },
        tutu => {
            Homedir => '/Home/tutu',
            Passwd  => '31PrfjWNa08Ha',
            Email   => 'tutu@ici.fr',
            Login   => ['mach2','mach4','mach3','mach8']
        }
);

# pour accéder à la liste des machines d'un utilisateur

@liste = @{$User{'tutu'}{'Login'}};

# $User{'tutu'}{'Login'} est une référence sur une liste,
# on la place entre @{...} pour la déréférencer.
```

14.4 Exercice numéro 11

```
#!/usr/bin/perl
#  TP11
#
#  Parcourir le fichier access.log (log d'APACHE) et indiquer
#  les URLs consultées par machines ainsi que les machines
#  ayant consultés par URL.
#  On utilise pour celà un hash de hash machine->URL->NbrHits
#  ainsi qu'un hash de hash URL->machine->NbrHits
use strict 'vars';
my (@Champ1, @Champ2, @Champ3, %HostTab, %UrlTab);
my ($Url, $Host);

open(LOG,"access.log") || die ("Erreur d'ouverture du log : $!");
while (<LOG>) {
    @Champ1 = split;         # récupère le nom de la machine
    $Host = $Champ1[0];
    @Champ2 = split(/\"/);   # récupère la requête HTTP effectuée
    @Champ3 = split(/ /,$Champ2[1]);
    $Url = $Champ3[1];       # récupère l'URL demandée
    $HostTab{$Host}{$Url}++; # ajout dans le hash de hash des mach
    $UrlTab{$Url}{$Host}++;  # ajout dans le hash de hash des URL
}
#
#  Impression des URL consultées par machines
#
foreach $Host (keys %HostTab) {
    print ('*** ',"  $Host  ",'***',"\n");
    foreach (keys %{$HostTab{$Host}}) {
        print ("         $_ : $HostTab{$Host}{$_} \n");
    }
}
#
#  Impression des machines ayant consultés par URL
#
foreach $Url (keys %UrlTab) {
    print ('### ',"  $Url  ",'###',"\n");
    foreach (keys %{$UrlTab{$Url}}) {
        print ("         $_ : $UrlTab{$Url}{$_} \n");
    }
}
```

14.5 Exercice numéro 12

```
#!/usr/bin/perl
#  TP12
#  Présente la liste des fichiers et répertoires contenus dans le
#  répertoire courant "à la windows" :
#     - les répertoires d'abord
#     - dans l'ordre alphanumérique sans respecter la casse

use strict 'vars';

#  obtenir la liste des fichiers et répertoires

my(@List) = <* .*>;

#  Construire un hash qui pour chaque nom de fichier minusculisé
# donne en correspondance la liste des vrais noms qui correspondent.

my (%WithUC);
foreach (@List) {
  push (@{$WithUC{lc($_)}},$_);  # C'est un hash de listes.
}

#  Construire une liste de tous les noms fichiers minusculisés.

my ($prec, @lcased, @result);
foreach (@List) {
  push(@lcased,lc($_));
}

#  Trier cette liste en enlevant les doublons. A partir du hash dont
#  la clef est un nom de fichier minusculisé on accède à tous les
#  fichiers portant ce nom (majuscules et minuscules non confondues).

my (@sorted) = sort(@lcased);
foreach (@sorted) {
  unless ($_ eq $prec) {
    push(@result,@{$WithUC{$_}});
    $prec=$_;
  }
}
```

```
# Obtenir une liste des répertoires et une liste des fichiers.

my (@dirs, @files);
foreach (@result) {
  if (-d) {
    push(@dirs,$_);
  }
  else {
    push(@files,$_);
  }
}

# Impression du résultat

foreach (@dirs) {
  print ("-d $_ \n");
}
foreach (@files) {
  print ("$_ \n");
}
```

14.6 Exercice numéro 13

```
#!/usr/bin/perl
#  TP13
#
#  On souhaite rechercher l'existence d'un mot passé en
#  argument dans tous les fichiers du répertoire courant.
#  À la fin on affiche le nom des fichiers qui contiennent
#  le mot recherché avec en regard le nombre d'occurrences
#  et les numéros de lignes ou il est présent. Le résultat
#  est présenté par ordre décroissant du nombre d'occurrences.

#  On utilise un tableau associatif dont la clef est le
#  nom du fichier et la valeur est une référence sur une
#  liste contenant le nombre d'occurrences ainsi qu'une
#  référence sur une liste contenant les numéros de lignes
#  ou le mot est présent.
```

```
use strict 'vars';

unless (@ARGV) {
    print ("Usage : tp13 'mot à rechercher' \n");
    exit();
}
my $mot = $ARGV[0];
my %GrepResult=();
my %Sorted=();

#  Constitution du tableau associatif (la clef est le nom
#  des fichiers qui contiennent le mot rcherché).

Grep('.');

#  Constitution d'un hash dont la clef est le nombre
#  d'occurrences et la valeur est une liste des fichiers
#  qui contiennent ce mot "occurrences" fois.

foreach (keys(%GrepResult)) {
  my @liste = @{$GrepResult{$_}};
  my $nb = $liste[0];
  push(@{$Sorted{$nb}}, $_);
}

#
#  Présentation du résultat dans l'ordre décroissant.
#
my $occur;
foreach $occur (sort {$b <=> $a} (keys(%Sorted))) {
  foreach (@{$Sorted{$occur}}) {
    # $_ contient un nom de fichier qui contient le mot
    # $occur fois.
      my @liste = @{$GrepResult{$_}};
      print ("$_ $liste[0] @{$liste[1]} \n");
  }
}
```

```
sub Grep () {
##############

    my ($Dir) = @_;
    my (@fichiers,$fic);

    opendir(DIR,$Dir) || return(0);
    @fichiers=readdir(DIR);
    closedir(DIR);
    foreach $fic (@fichiers) {
      if (($fic ne ".") && ($fic ne "..")) {
        if (-d "${Dir}/$fic") {
          Grep("${Dir}/$fic");
        }
        else {
          unless (open(FIC,$fic)) {
            print ("Impossible d'ouvrir $fic : $! \n");
            next;
          }
          my $lig = 0;;
          while (<FIC>) {
          my $count = 0;
            $lig++;
            my $chaine = $_;
            if (/$mot/) {
              while ($chaine =~ $mot) {
                $count++;
                $chaine = $';
              }
              ${$GrepResult{$fic}}[0] += $count;
              push(@{${$GrepResult{$fic}}[1]},$lig);
            }
          }
          close(FIC);
        }
      }
    }
}
```

Chapitre 15

Le débogueur de Perl

Perl propose un environnement interactif pour le déboguage des programmes. Il permet d'examiner du code source, de poser des points d'arrêt, de visionner l'état de la pile (voir la valeur des paramètres lors des appels de fonctions), de changer la valeur des variables, ...

Le débogueur est appelé en spécifiant l'option **-d** lors de l'appel de Perl, on entre alors dans un mode interactif, le débogueur attend des commandes pour dérouler le code comme dans l'exemple suivant (où le programme comporte au moins une erreur) :

```
1 #!/usr/local/bin/perl
2 my ($moy, $total, $nb);
3 print ("Entrez un nombre \n");
4 while (<STDIN>) {
5   $total += $_;
6   print ("Entrez un nombre \n");
7 }
8 $moy = $total / $nb;
9 print ("la moyenne est : $moy \n");
```

Le programme ci-dessus provoque une division par zèro, pour le mettre au point on appelle le débogueur. Dans l'exemple suivant, le texte précédé par >> est généré par Perl, celui précédé par << est fourni par le programmeur qui effectue la mise au point.

```
perl -d debug.pl
  >> Loading DB routines from perl5db.pl version 1.0402
  >> Emacs support available.
  >> Enter h or 'h h' for help.

  >> main::(debug.pl:2):     my ($moy, $total, $nb);
  >> DB<1>  << /\$moy/
  >> 8:     $moy = $total / $nb;
  >> DB<2>  << b 8
  >> DB<3>  << c
  >> Entrez un nombre
  << 2
  >> Entrez un nombre
  << 4
  >> Entrez un nombre
  << Ctrl-d
  >> main::(debug.pl:8):    $moy = $total / $nb;
  >> DB<3>  << x $total
  >> 0  6
  >> DB<4>  << x $nb
  >> 0  undef
  >> DB<5>  << $nb=2
  >> DB<6>  << c
  >> la moyenne est : 3

  >> Debugged program terminated.  Use q to quit or R
  >> to restart,
  >> use O inhibit_exit to avoid stopping after
  >> program termination,
  >> h q, h R or h O to get additional info.
  >> DB<6> << q
```

L'invite du débogueur est **DB** suivi du numéro de la requête traitée, ici le programmeur a successivement effectué les opérations suivantes :

- recherche de la prochaine ligne de code exécutable qui contient la chaine **$moy** (/**\$moy**/);
- pose d'un point d'arrêt sur la ligne 8 qui correspond à la recherche effectuée (**b 8**);
- demande l'exécution du programme jusqu'au prochain point d'arrêt (**c**);
- rentre 2 valeurs (2 et 4) et marque la fin de saisie (**Ctrl-d**);
- demande à consulter le contenu de la variable **$total** lorsque le débogueur s'arrête

sur le point d'arrêt (**x $total**);
- demande l'affichage de la variable **$nb** (**x $nb**) et constate qu'elle est la source de l'erreur (**undef**, c'est à dire 0);
- affecte 2 à la variable $nb (**$nb=2**).
- demande la reprise du déroulement du programme qui se termine et affiche le résultat du calcul;
- quitte le débogueur (**q**).

Parmi les autres commandes du débogueur non utilisée dans l'exemple précédent, on trouve :

- `S`, demander un déroulement pas à pas;
- `T`, afficher la pile (et voir les paramètres transmis à une fonction);
- `t`, afficher ou ne plus afficher les lignes de programmes exécutées (c'est une bascule désamorcée par défaut);
- `D`, supprimer tous les points d'arrêts.

Chapitre 16

Écriture et utilisation de modules

Perl autorise l'écriture de modules qui peuvent être assemblés pour former une application. Cette fonctionnalité a permis le développement de nombreuses contributions, des modules ont été développés pour interfacer TCP, CGI, FTP ... c'est une des raisons du succès de Perl.

16.1 Inclure du code

La forme la plus simple de l'écriture modulaire est l'inclusion de code, autorisée en Perl par la fonction **require** suivie d'une expression (en général une chaîne de caractère représentant un nom de fichier contenant le code à inclure). Il s'agit d'une fonctionnalité équivalente au **#include** du langage C, on regroupe en général les inclusions en début de programme :

```
#!/usr/local/bin/perl
require "cgi-lib.pl";   # Inclusion de la bibliothèque
                        # gérant l'interface CGI
```

Les modules Perl écrits ainsi sont souvent un peu *anciens* et proposent des interfaces rudimentaires :

```
#!/usr/local/bin/perl
#
# Programme principal qui utilise le module monmodule.pl
#
require "monmodule.pl";

# on peut simplement appeler la fonction fonc() définie dans
# monmodule.pl et aussi avoir accès à sa variable $I ...

fonc();
print ("I = $I \n");
```

```
#   monmodule.pl
#
#   Petit module accessible simplement, les variables et
#   fonctions sont directement accessibles ...

$I = 2;
sub fonc {
        print (\"appel de fonc() \n\");
}
```

Perl cherche les modules à inclure dans le répertoire courant et dans les répertoires cités dans le tableau **@INC**. Pour ajouter des répertoires au tableau **@INC** il convient de modifier la variable d'environnement **Perl5LIB** ou de modifier **@INC** avant le **require** :

```
#!/usr/local/bin/perl
unshift(@INC,"/usr/local/perl/lib/maison");
require "cgi-lib.pl";   # Inclusion de la bibliothèque
                        # gérant l'interface CGI
```

Les fichiers inclus par **require** sont chargés à l'exécution du programme. Il en résulte une souplesse d'usage liée au fait que l'on peut inclure des fichiers dynamiquement (en fonction du contenu d'une variable). On n'est toutefois pas sûr lorsqu'un programme démarre que tout est prêt pour son bon déroulement (un fichier à inclure qui n'existe pas, qui comporte une erreur de syntaxe ...).

Si l'expression citée par **require** est numérique, il s'agit du numéro de version de Perl requis pour ce qui suit.

16.2 Les packages

En écrivant classiquement des modules destinés à être inclus par **require**, le programmeur doit faire attention à la visibilité des variables (cf. section 10). Une première solution est l'usage de **my**, Perl offre toutefois une autre solution : les **packages** inclus par **use**.

Une partie de code isolée dans une déclaration package est destinée à avoir une visibilité externe organisée, le programmeur peut indiquer explicitement quelles sont les variables, les fonctions qui peuvent être simplement utilisées par les consommateurs.

On peut trouver plusieurs déclarations de packages dans un fichier, ce n'est toutefois pas très utilisé, on place généralement un package dans un fichier dont le suffixe est **.pm** (Perl Module).

```
package essai;      # Le nom du package et du fichier .pm

use Exporter;       # appel au module gérant la visibilité
@ISA=('Exporter'); # hérite d'Exporter (non traité ici,
                    # voir la section sur les objets)

@EXPORT_OK=('Fonc1,'Fonc2','$Var');

# Le tableau @EXPORT_OK est utilisé pour préciser les
# identificateurs qui sont visibles de l'extérieur du
# package.

$Var = "visible";
$I = 10;

sub Fonc1 { ... }
sub Fonc2 { ... }
sub Fonc3 { ... }
sub Fonc4 { ... }
```

Dans l'exemple ci-dessus, seulement deux fonctions (*Fonc1* et *Fonc2*) ainsi qu'une variable (*$Var*) peuvent être utilisée depuis l'extérieur du package.

Pour utiliser le package essai (placé dans le fichier essai.pm) il convient de procéder

comme suit :

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

# use est suivi du nom du package  et des variables
# et fonctions à importer

use essai ('Fonc1','Fonc2','$Var');

Fonc1();
Fonc2();
print ("$Var \n");
# Un appel à Fonc3 produirait une erreur
```

Il est possible d'utiliser le tableau **@EXPORT** à la place de **@EXPORT_OK**, dans ce cas, même si le programmeur déclare vouloir importer seulement un sous ensemble des identificateurs exportés, il aura une visibilité sur l'ensemble. Il est donc préférable d'utiliser **@EXPORT_OK** qui évite les conflits potentiels.

Les identificateurs exportés constituent l'interface du package avec l'extérieur, PERL procure toutefois une autre interface avec les packages : le nommage explicite. Il est possible d'accéder à un identificateur (exporté ou non) en le précédant du nom de son package d'appartenance :

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

use essai;      # il n'y a pas d'obligation a importer

$I = 20;
essai::Fonc1(); # les identificateurs exportés ou non
essai::Fonc2(); # sont accessibles.
essai::Fonc3();
printf ("%d \n",$essai::I); # affiche 10 (le I de essai.pm)
```

La démarche import/export est évidemment beaucoup plus claire et se généralise.

Les packages peuvent être dotés de constructeurs et de destructeurs rédigés sous la forme de fonctions appelées respectivement **BEGIN** et **END**. Les constructeurs sont exécutés

dès que possible (à la compilation) et les destructeurs le plus tard possible (lorsque le programme se termine normalement ou pas).

```
package essai;

BEGIN {  # sub n'est pas obligatoire
    print ("debut du package \n");
}

beurk();

END {    # sub n'est pas obligatoire
    print ("fin du package \n");
}
```

Le package *essai.pm* de l'exemple ci-dessus produit les sorties suivantes :

```
debut du package
   Undefined subroutine &essai::beurk called
                        at essai.pm line 7.
   BEGIN failed--compilation aborted at ...
fin du package
```

16.3 Remarques

La différence indiquée ici entre **use** et **require** est fictive (elle correspond toutefois à un usage largement répandu), il est en fait parfaitement possible d'inclure un package en utilisant **require**. Dans ce cas la routine d'import n'est pas appelée mais elle peut l'être *à la main*, ceci permet d'inclure des packages dynamiquement :

```
if ($a) {
   require "moda";  # Inclusion de moda.pm ou modb.pm
   moda->import();  # en fonction du contenu d'une variable
}
else {              # L'import n'est pas effectué (require)
   require "modb";  # on le fait donc explicitement.
   modb->import();
}
```

16.4 Exercice numéro 15

```
#
#  Petit package d'outils HTML pour montrer comment on exporte
#  des variables et des fonctions.
#  Évidemment les fonctions ci-dessous n'ont pas d'intérêt en
#  dehors du contexte de ce TP.
package html;
use Exporter;
@ISA=('Exporter');

#  Le package exporte une variable et trois fonctions.

@EXPORT=('$html_Version','html_Escape','html_Title',
         'html_Background');

# variable globale au package, elle est donc exportable.

$html_Version = 'html.pm version 1.0';

use strict 'vars';

# L'emploi de use 'strict rend obligatoire de manipuler
# les variables globales au package en les préfixant par
# le nom du package : $html::html_Version par exemple ...

sub html_Escape {
##################

# On modifie les caratères accentués les plus usuels ...

  my (@content) = @_;
  my @res;
  foreach (@content) {
    s/é/&eacute;/g;
    s/è/&Egrave;/g;
    s/ç/&ccedil;/g; # etc etc etc ....
    push(@res,$_);
  }
  push(@res,"<!-- *** modifié avec $html::html_Version *** -->");
  return(@res);
}
```

```
sub html_Title {
################

  my ($title, @content) = @_;
  my (@res);

#
#  On modifie le titre (sous réserve qu'il existe ...).
#

  foreach (@content) {
    s/<TITLE>.*<\/TITLE>/<TITLE>$title<\/TITLE>/gi;
    push (@res,$_);
  }
  return(@res);
}

sub html_Background {
####################

  my ($background, @content) = @_;
  my (@res);

#
# On ajoute un attribut bgcolor à la balise BODY
# si elle existe.
#

  foreach (@content) {
    if (/.*<BODY/i) {
      $_ = $&." bgcolor='$background'".$';
    }
    push (@res,$_);
  }
  return(@res);
}

#  Quand on importe un package en utilisant en utilisant 'use'
#  le contenu du package est passé à 'eval'. La dernière expression
#  évaluée doit rendre vrai car sinon 'eval' échoue ...
return(1);
```

```
#!/usr/bin/perl
#  TP15
#  Utilisation du package html.pm pour modifier
#  un document HTML.
use html;              # Inclusion du package html
use strict 'vars';
my $File;
# S'il n'est pas donné de fichier, on traite index.html
unless (@ARGV) {
  $File = "index.html";
}
else {
  $File = $ARGV[0];
}
unless (open(IN,$File)) {
  print ("Impossible d'ouvrir $File : $! \n");
  exit();
}
my @Content = <IN>;
close(IN);
#  Appel des fonctions du package
@Content = html_Escape(@Content);
@Content = html_Title("nouveau titre",@Content);
@Content = html_Background("#ff0000",@Content);
#  Mise à jour du fichier
unless (open(OUT,">$File.new")) {
  print ("Impossible de réécrire $File.new : $! \n");
  exit();
}
unless (print OUT (@Content)) {
  print ("Impossible de réécrire $File.new : $! \n");
  exit();
}
close(OUT);
unless (rename("$File.new",$File)) {
  print ("Impossible de mettre à jour $File : $! \n");
  exit();
}
else {
  print ("$File modifié à l'aide de $html_Version \n");
}
```

Chapitre 17

Écriture orientée objet

17.1 Un exemple limité à l'usage classique des packages

Si l'on considère une classe d'objet comme une structure de données accompagnée de procédures qui régissent le comportement de ses membres potentiels, alors en Perl on peut essayer d'utiliser les packages.

Dans l'exemple suivant on désire gérer des *objets* d'une *classe* appelée *voiture* caractérisés chacun par une marque et une couleur, à tout moment on souhaite connaître le nombres de voitures en cours.

```
#!/usr/local/bin/perl
use voiture;

$voit1 = voiture::nouvelle('verte','citroen');
$voit2 = voiture::nouvelle('bleue','renault');
$voit3 = voiture::nouvelle('rouge','citroen');

printf (" %s \n", voiture::couleur($voit1));
printf (" %s \n", voiture::marque($voit2));
printf (" %s \n", voiture::couleur($voit3));
printf (" %d \n", voiture::total());
```

```
package voiture;
BEGIN { @liste=();}

sub nouvelle {
    my ($color,$marque) = @_;
    my $objptr = {};
    $objptr->{'couleur'}=$color;
    $objptr->{'marque'}=$marque;
    push(@liste,$objptr);
    return $objptr;
}
sub couleur {
    my ($objptr) = @_;

    return ($objptr->{'couleur'});
}
sub marque {
    my ($objptr) = @_;

    return ($objptr->{'marque'});
}
sub total {
    return($#liste+1);
}
1
```

Dans l'exemple précédent, la *classe voiture* est un package, le constructeur de package BEGIN est utilisé pour initialiser la liste des *objets* à vide.

La fonction *nouvelle* est le constructeur de la *classe*, elle reçoit en arguments la couleur et la marque d'un *objet* et retourne la référence de l'*objet* instancié. La référence de l'*objet* est ici une référence sur un hash anonyme ($objptr = {} cf. section 11.4.1). Les fonctions *couleur* et *marque* sont les *méthodes d'instance* de la *classe voiture*. Les fonctions *nouvelle* et *total* sont des *méthodes de classe*.

Si le début du code qui utilise la *classe voiture* s'apparente à du code orienté objet (appel au constructeur notamment), l'utilisation des méthodes s'en éloigne. La référence à l'objet ($voit3 par exemple) n'a pas de souvenir du type de l'objet référencé et on ne peut l'utiliser qu'en citant le nom du package.

Perl propose donc un type de références particulier, il va permettre d'appeler les méthodes directement à partir de la référence à l'objet. Dans l'exemple précédent on souhaite plutôt

écrire **$voit3->couleur()** et non pas **voiture::couleur($voit3)**.

17.2 Référencer une classe

Dans l'exemple de la classe *voiture* les références $voit1, $voit2 et $voit3 n'ont pas connaissance de la classe d'appartenance des objets qu'elles référencent. La fonction **bless** est utilisée pour celà, on peut maintenant modifier le constructeur de la classe *voiture* comme suit :

```
sub nouvelle {
    my ($classe,$color,$marque) = @_; # le nom de la classe
                                      # est le 1er argument
    my $objptr = {};

    $objptr->{'couleur'}=$color;
    $objptr->{'marque'}=$marque;
    bless $objptr;                    # référence la classe
    push(@liste,$objptr);
    return $objptr;
}
```

Un affichage de la valeur de **$objptr** indique explicitement la classe de l'objet référencé (*voiture=HASH(0xca454)*).

Le programme qui utilise la classe *voiture* peut maintenant être modifié de la manière suivante :

```
#!/usr/local/bin/perl
use voiture;

$voit1 = voiture->nouvelle('verte','citroen');
$voit2 = voiture->nouvelle('bleue','renault');
$voit3 = voiture->nouvelle('rouge','citroen');

printf (" %s \n", $voit1->couleur());
printf (" %s \n", $voit2->marque());
printf (" %s \n", $voit3->couleur());
printf (" %d \n", voiture->total());
```

Il convient ici de noter la forme d'appel des méthodes de classe (**voiture->nouvelle()**) et la forme d'appel des méthodes d'instance (**$voit1->couleur()**). Les méthodes de classe s'appellent en précédant la méthode par le nom de la classe, et, le nom de la classe est alors passé en premier argument (ce qui justifie la changement intervenu dans notre classe *voiture*).

Les méthodes d'instance recoivent comme premier argument une référence d'objet, il n'est donc pas nécessaire de modifier le code de notre classe *voiture*.

Comme bien souvent en Perl, il existe plusieurs notations pour un même résultat et, il est possible de faire précéder le nom d'un objet ou d'une classe par le nom de la méthode. L'exemple précédent peut donc s'écrire

```
#!/usr/local/bin/perl
use voiture;

$voit1 = nouvelle voiture ('verte','citroen');
$voit2 = nouvelle voiture ('bleue','renault');
$voit3 = nouvelle voiture('rouge','citroen');

printf (" %s \n", $voit1->couleur());
printf (" %s \n", $voit2->marque());
printf (" %s \n", $voit3->couleur());
printf (" %d \n", voiture->total());
```

Si on rebaptise la méthode *nouvelle* en *new*, on utilise une notation qui est familière au programmeur C++ ou Java.

17.3 Hériter d'une classe

Perl autorise de construire de nouvelles classes à partir de classes existantes en utilisant des techniques d'héritages :

- la liste **@ISA** indique à un package (à une classe) quels sont les ancêtres dont il hérite;
- de manière à pouvoir être hérité, le constructeur d'une classe de base doit rendre une référence *blessee* de la classe spécifiée (de la classe dérivée).

Pour construire une classe appelée *location* (gérant la location de véhicules) héritant de notre classe *voiture*, on peut procéder comme suit :

```
package location;
use voiture;        # inclure voiture.pm pour la compile
@ISA = ("voiture"); # hériter des méthodes de la classe
                    # voiture.
sub loc {
  my ($classe, $couleur, $marque, $nbjour, $pu) = @_;
#  appel du constructeur hérité de la classe de base, il
#  rendra un objet de classe location
  my $locptr = $classe->nouvelle($couleur,$marque);
  $locptr->{'pu'}=$pu;
  $locptr->{'nbjour'}=$nbjour;
  return($locptr);
}

sub tarif {
    my ($locptr) = @_;
    return ($locptr->{'pu'} * $locptr->{'nbjour'});
}
```

Il convient de bien noter que la classe *voiture* doit être modifiée pour que le constructeur *nouvelle()* rende un objet de la classe *location* et non pas un objet de la classe *voiture* :

```
sub nouvelle {

    my ($classe,$color,$marque) = @_; # le nom de la classe
                                      # est le 1er argument
    my $objptr = {};

    $objptr->{'couleur'}=$color;
    $objptr->{'marque'}=$marque;
    bless $objptr,$classe;  <==================
    push(@liste,$objptr);
    return $objptr;

}
```

Un programme utilisant la classe *location* peut s'utiliser comme dans l'exemple ci-dessous :

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

use location;

$voit1 = location->loc('verte','peugeot',10,100);
print $voit1;
print $voit1->couleur();
print $voit1->tarif();
```

17.4 Exercice numéro 16

```
package champ;

#
#  Une classe champ qui gère la largeur, la longueur et
#  le type de culture d'un champ. La classe permet de
#  connaitre la surface d'un champ ainsi que le nombre
#  de champs et la surface cultivée pour un type de
#  culture passé en argument.
#

my @champs = ();

sub new {
#########

#
#  Le constructeur reçoit le nom de la classe comme premier
#  argument.
#

  my ($classe, $largeur, $longueur, $culture) = @_;

  my $champtr = {}; # pour gérer les données d'instances.

  $champtr->{'largeur'}  = $largeur;
  $champtr->{'longueur'} = $longueur;
  $champtr->{'culture'}  = $culture;

  bless($champtr,'champ');  # typer la référence sur l'objet de
                            # classe champ.
  push(@champs, $champtr);
  return($champtr);
}
```

```
sub nbrchamps {
###############

#
#  Une méthode de classe, le nom de la classe est reçu comme
#  premier argument.
#

  my ($classe) = @_;

  return($#champs+1);
}

sub surface {
#############

#
#  Une méthode d'instance de classe. Elle reçoit en argument
#  une référence sur un objet de classe champ.

  my ($champtr) = @_;

  return($champtr->{'largeur'} * $champtr->{'longueur'});

}

sub surface_cultivee {
######################

  my ($classe, $culture) = @_;

  foreach (@champs) {
    if ($_->{'culture'} eq $culture) {
      $total += $_->{'largeur'} * $_->{'longueur'};
    }
  }
  return($total);

}
1
```

```
#!/usr/bin/perl

#
#  TP16
#
#  utilisation de la classe champ.
#

use champ;
use strict 'vars';

print ("Largeur du champ : ");

while (my $Largeur=<STDIN>) {
    # saisie des données d'un champ.
    chomp($Largeur);
    print ("Longueur : ");
    my $Longueur=<STDIN>;
    unless ($Longueur) {
        last;
    }
    chomp($Longueur);
    print ("Culture : ");
    my $Culture=<STDIN>;
    unless ($Culture) {
        last;
    }
    chomp($Culture);
    # création d'un objet de classe champ.
    my $unchamp = new champ($Largeur,$Longueur,$Culture);
    # affichage de la surface du champ rentré.
    printf ("La surface de ce champ est %s\n", $unchamp->surface());
    print ("Largeur du champ : ");
}

printf ("\n Nombre de champs %s \n", champ->nbrchamps());
print ("Culture à mesurer : ");
my $cult=<STDIN>;
chomp($cult);
printf ("Surface cultivée en %s %s \n",
        $cult, champ->surface_cultivee($cult));
```

Chapitre 18

L'écriture de scripts CGI

18.1 Introduction

Common Gateway Interface (CGI [1]) est une spécification d'interface permettant d'exécuter des procédures externes depuis un service WWW. CGI spécifie (du côté du serveur) comment doit être initialisée la procédure externe et comment elle doit accéder aux informations à traiter. Les informations transmises à la procédure externe sont :

- des variables qui correspondent aux en-têtes de la requête HTTP qui a déclenché l'appel de la procédure externe (HTTP_REFERER, HTTP_USER_AGENT, HTTP_CONNECTION, HTTP_HOST, ...);
- des variables relatives au corps d'un document transmis ou propres au contexte CGI (REQUEST_METHOD, CONTENT_TYPE, CONTENT_LENGTH, REMOTE_HOST, QUERY_STRING, ...);
- un corps de document éventuel.

CGI indique également que si un corps de document est transmis, il doit être accédé via l'entrée standard de la procédure externe. De plus la procédure externe (le script CGI) doit toujours rendre un résultat via sa sortie standard. La spécification des ces deux mécanismes impose (au serveur HTTP) un mode de création assez lourd et couteux du script CGI (fork + redirection de l'entrée et de la sortie standard + exec). C'est la raison principale du manque de performance des scripts CGI, à chaque requête, le système d'exploitation est fortement sollicité.

1. En service depuis 1993, CGI n'a jamais porté le status de RFC, une proposition est en cours de réalisation cf. http://web.golux.com/coar/cgi

Dans la pratique, malgré son manque de performance, CGI reste très utilisé pour transmettre des informations d'un client WWW vers un service particulier (via un serveur HTTP). C'est notamment ainsi que fonctionnent nombre de formulaires WWW, le synoptique des échanges entre client, serveur et script CGI est alors le suivant :

- un client WWW affiche un formulaire ;
- l'utilisateur complète les champs à transmettre et valide le contenu qui est transmis. Si la méthode HTTP utilisée pour transmettre le formulaire est **GET**, alors les champs (l'information à transmettre) sont transmis dans l'URL d'appel du script CGI et lui sont passés via la variable QUERY_STRING.
  Si la méhode HTTP est **POST**, alors les champs sont transmis dans un corps de document (dont le **CONTENT_TYPE** est **application/x-www-form-urlencoded**) et sont passés au script CGI via son entrée standard. Cette dernière façon de procéder est fortement conseillée.
  Le type de document **application/x-www-form-urlencoded** implique que les champs du formulaire sont séparés entre eux par le caractère &, qu'ils sont précédés de leur nom (attribut NAME de la balise HTML <FORM>) et du signe =, que les caractères accentués et les espaces sont encodés comme spécifié dans le RFC1738 sur les URL;
- un serveur HTTP réceptionne la requête et initialise le script CGI référencé par le formulaire WWW ;
- le script CGI effectue le traitement demandé et se charge d'émettre un résultat à destination du client WWW instigateur de la requête.

En dehors de toute aide extérieure, un programmeur de script CGI doit donc décoder le **application/x-www-form-urlencoded** pour accéder individuellement aux champs transmis, il doit également prendre en charge l'émission du résultat (et donc générer des entêtes HTTP correctes suivies d'un corps de document HTML). En Perl diverses contributions permettent de faciliter ces deux tâches, les plus connues sont **cgi-lib.pl** et **CGI.pm**.

18.2 Un exemple

Dans l'exemple suivant on désire compléter un formulaire WWW comportant deux champs (nom et prénom), le transmettre à une procédure CGI qui pourrait stocker les champs transmis dans un fichier ... mais se contentera ici d'avertir l'utilisateur de la prise en compte de l'information. Le code HTML du formulaire est placé dans un document à part (extérieur à la procédure CGI[2]), on vérifie que la méthode HTTP utilisée est bien **POST** et on montre à l'utilisateur les renseignements obtenus sur lui (machine, client

2. Il est possible de placer le source du formulaire dans la procédure, elle doit alors émettre le source du formulaire si la méthode est **GET**, traiter les champs saisis si la méthode est **POST**.

WWW, ...).

```
<HTML>
<!-- Source HTML permettant d'afficher le formulaire et
     d'appeler la procédure CGI référencée par l'attribut
     action de la balise <FORM>.
-->
<HEAD>
<TITLE> Test CGI </TITLE>
</HEAD>
<BODY>
<H1> Test CGI </H1>
<HR>
<FORM action="http://madeo.irisa.fr/cgi-bin/tstcgi"
      method="POST">
  Nom :   <INPUT TYPE="text" NAME="nom"    SIZE="25"> <BR>
  Prénom: <INPUT TYPE="text" NAME="prenom" SIZE="15"> <BR>
  <INPUT TYPE="submit" VALUE="OK">
  <INPUT TYPE="reset" VALUE="Annuler">
</FORM>
<HR>
</BODY>
</HTML>
```

18.2.1 Utiliser cgi-lib.pl

La première bibliothèque qui a été diffusée pour écrire des scripts CGI en Perl est cgi-lib.pl[3]. Elle reste aujourd'hui très utilisée et présente les caractéristiques suivantes :

- permet de récupérer les champs transmis dans un tableau associatif ;
- gère le transfert de fichiers dans le sens client serveur (*upload*) ;
- globalement très simple à utiliser.

Le script Perl suivant propose d'utiliser cgi-lib.pl pour réaliser notre exemple, il utilise les fonctions suivantes :

- **Methpost()** qui rend vrai si la méthode HTTP utilisée est **POST**;

3. http://cgi-lib.stanford.edu/cgi-lib/

– **ReadParse()** qui prend une référence sur un hash et le complète par une association entre le nom des champs (tel que spécifiés par l'attribut NAME de la balise <FORM>) et leurs valeurs respectives;

– **PrintHeader()** qui indique au navigateur qu'il va recevoir un document HTML. Cette fonction prend aussi à sa charge l'émission d'une ligne vide pour séparer les en-têtes du corps de document (cf. RFC2068 sur HTTP/1.1);

– **CgiDie** pour quitter la procédure en émettant un message d'avertissement à l'utilisateur;

– ...

```
#!c:\perl\bin\perl.exe
#  TestCGI utilise cgi-lib pour acquerir les champs du
#  formulaire et generer le HTML resultat. On verifie
#  que la methode est POST, que les champs ne sont pas
#  vides et on informe l'utilisateur du resultat.

require "cgi-lib.pl";
$champs= {};   # $champs est une reference sur un hash vide

if (MethPost()) {
  ReadParse($champs);
  if ($champs->{"nom"} eq "" || $champs->{"prenom"} eq "")
 CgiDie("Il faut remplir les champs !");
  }
  print (PrintHeader(),
         HtmlTop("Resultat de votre requete"),
         "Nom : ", $champs->{"nom"}, "<BR>",
         "Prenom :", $champs->{"prenom"}, "<BR>",
         "vous utilisez $ENV{HTTP_USER_AGENT}",
         "depuis la machine $ENV{REMOTE_ADDR}",
         HtmlBot());
}
else {
    CgiDie("Hum ... Que faites vous !");
}
```

18.2.2 Utiliser CGI.pm

CGI.pm[4] est un autre outil facilitant la réalisation de script CGI en Perl. CGI.pm peut être utilisé de deux façons :

- en important des fonctions dans l'espace du script, CGI.pm est alors utilisé comme un package;
- en utilisant un objet de classe CGI comme dans la réalisation (ci-après) de notre exemple.

```
#!c:\perl\bin\perl.exe
#  TestCGI utilise CGI.pm pour acquerir les champs du
#  formulaire et generer le HTML resultat. On verifie
#  que la methode est POST, que les champs ne sont pas
#  vides et on informe l'utilisateur du resultat.
use CGI;
$req = new CGI;

print $req->header();
print $req->start_html('Résultat de la requête');
if ($req->request_method() eq "POST") {
   if ($req->param('nom')    eq " " ||
       $req->param('prenom') eq " " ) {
   print $req->h1('Il faut remplir les champs');
   }
   else {
     print ("Nom : ", $req->param('nom'),
            $req->br,
            "Prenom : ", $req->param('prenom'),
            $req->br,
            " vous utilisez", $req->user_agent(),
            "depuis la machine ",$req->remote_addr());
   }
}
else {
print $req->h1('Hum ... Que faites vous !');
}
print $req->end_html;
```

4. http://www.wiley.com/compbooks/stein/index.html

CGI.pm peut semble run peu plus lourd à utiliser que cgi-lib.pl, il est toutefois plus élaboré pour :

- générer les en-têtes HTTP, dans notre exemple, *$req->header* se contente de positionner *Content-Type: text/html* (suivi d'une ligne vide), mais pourrait aussi indiquer une en-tête expire (*$req->header(-expires=>'now');*) pour inhiber les caches;
- générer du code HTML (notamment des tables, des formulaires, ...);
- être utilisé dans un contexte FastCGI (cf. 18.4) ;
- mettre au point les scripts sans passer par un navigateur WWW et un serveur HTTP. CGI.pm reconnait s'il est appelé dans un contexte dit *offline* et invite alors l'utilisateur à rentrer à la main des couples *clef=valeur* pour simuler la chaîne complète d'un environnement CGI.

18.3 Envoyer un fichier (*upload*)

Il est parfois intéressant de transférer des documents complets depuis un client WWW vers un serveur HTTP. CGI.pm ainsi que cgi-lib.pl offrent cette possibilité. Elle s'appuie sur le transfert dans le corps d'un document (comprenant plusieurs parties), des champs éventuels d'un formulaire suivi des différents fichiers.

On peut compléter notre exemple en demandant à l'utilisateur de fournir deux fichiers qui seront transférés sur le serveur et transmis à notre script (via cgi-lib.pl ici). Le code HTML peut être modifié de la façon suivante :

```
<HTML>
<!-- Le  document est composé de plusieurs parties, donc
 (ENCTYPE="multipart/form-data") + ajout de 2 champs "file".
-->
<HEAD> <TITLE> Test CGI </TITLE> </HEAD>
<BODY>
<H1> Test CGI </H1><HR>
<FORM action="http://madeo.irisa.fr/cgi-bin/tfic.pl"
      ENCTYPE="multipart/form-data" method="POST">
  Nom :   <INPUT TYPE="text" NAME="nom"    SIZE="25"> <BR>
  Prénom: <INPUT TYPE="text" NAME="prenom" SIZE="15"> <BR>
  <INPUT TYPE="file" NAME="fic1" SIZE=50> <BR>
  <INPUT TYPE="file" NAME="fic2" SIZE=50> <BR>
  <INPUT TYPE="submit" VALUE="OK">
  <INPUT TYPE="reset" VALUE="Annuler">
</FORM> </BODY> </HTML>
```

Dans le cas d'un type de document *multipart/form-data* les arguments de la fonction ReadParse sont 4 références à des tableaux associatifs correspondants aux champs du formulaire, aux nom des fichiers transmis, aux *Content-Type* des fichiers transmis et aux noms des fichiers reçus. Notre procédure peut donc être modifiée de la façon suivante :

```
#!c:\perl\bin\perl.exe
#
#  Réception des deux fichiers.

require "cgi-lib.pl";
$champs = {};       # $champs, $client_fn,
$client_fn = {};    # $client_ct, $serveur_fn
$client_ct = {};    # sont des références sur
$serveur_fn = {};   # des hashes vides.
$cgi_lib::writefiles = "c:\tmp";  # Répertoire dans lequel
                                  # les fichiers transmis
                                  # seront déposés.
if (MethPost()) {
  ReadParse ($champs, $client_fn, $client_ct, $serveur_fn);
  if ($champs->{"nom"} eq "" || $champs->{"prenom"} eq "") {
        CgiDie("Il faut remplir les champs !");
  }
  print (PrintHeader(),
         HtnlTop("Resultat de votre requete"),
         "Nom : ", $champs->{"nom"}, "<BR>",
         "Prenom :", $champs->{"prenom"}, "<BR>",
         $serveur_fn->{'fic1'}, # nom du fichier 1 (serveur)
         $serveur_fn->{'fic2'}, # nom du fichier 2 (serveur)
         $client_fn->{'fic1'},  # nom du fichier 1 (client)
         $client_fn->{'fic2'},  # nom du fichier 2 (client)
         $client_ct->{'fic1'},  # Content-Type fichier 1
         $client_ct->{'fic2'},  # Content-Type fichier 2
         HtmlBot());
# Une procedure opérationnelle pourrait maintenant ouvrir
# les fichiers correspondants et effectuer un traitement.
}
else {
    CgiDie("Hum ... Que faites vous !");
}
```

Pour qu'une procédure effectuant du transfert de fichiers se déroule normalement, il convient que le serveur HTTP soit autorisé à écrire dans le répertoire *$cgi_lib::writefiles*,

il faut également que celui-ci soit correctement dimensionné.

18.4 Utiliser FastCGI

18.4.1 Introduction

Pour contourner le manque de performance des scripts CGI écrits en Perl, deux solutions sont envisageables :

- utiliser mod_perl[5] qui lie APACHE[6] et Perl et permet d'éviter la création de processus à chaque appel de procédure. Cette solution présente l'avantage de permettre l'écriture en Perl de modules pour APACHE[7], mais elle introduit quelques inconvénients, notamment une très forte augmentation de la taille mémoire utilisée par un processus *httpd*;
- utiliser FastCGI[8] qui fait communiquer le serveur HTTP et la procédure externe via une *socket*. La procédure externe fonctionne alors comme un serveur à l'écoute sur un port, elle traite les requêtes de son client (le serveur HTTP) qui lui transmet les données émises par les navigateurs WWW. La procédure externe est démarrée à l'initialisation du serveur HTTP, elle est persistante, le gain de performance par rapport à CGI est très significatif.
  FastCGI permet également de délocaliser la procédure externe (la faire se dérouler sur une machine distincte de celle qui héberge le serveur HTTP), c'est une fonctionnalité très intéressante, elle permet de répartir les traitements.

CGI.pm permet d'utiliser FastCGI comme canal de communication, en conséquence les changements à apporter pour faire migrer une procédure de CGI vers FastCGI sont très minimes (cf. exemple ci-dessous).

18.4.2 Un exemple

On reprend ici l'exemple de la section 18.2.2, les changements à effectuer ont pour but d'indiquer à CGI.pm de basculer dans un contexte FastCGI et de rendre la procédure

5. Consulter ftp://ftp.bora.net/pub/CPAN/modules/by-module/Apache/ la version actuelle est mod_perl-1.18.tar.gz

6. Le serveur HTTP le plus utilisé au monde, consulter http://www.apache.org

7. Pour ajouter des fonctionnalités au serveur HTTP de base.

8. Consulter http://www.fastcgi.com/ et installer le module fastcgi pour APACHE ainsi que FCGI pour Perl

persistante (boucler sur l'attente de l'arrivée de données à traiter). Pour mettre en évidence l'effet de la persistance de la procédure, on ajoute un compteur de requêtes.

```
#!/usr/local/bin/perl
#  TestCGI utilise CGI.pm pour acquerir les champs du
#  formulaire et generer le HTML resultat. On verifie
#  que la methode est POST, que les champs ne sont pas
#  vides et on informe l'utilisateur du resultat.

use CGI::Fast;
$Cptr = 0;
while ($req=new CGI::Fast) {
$Cptr++;
print $req->header();
print $req->start_html('Résultat de la requête');
if ($req->request_method() eq "POST") {
   if ($req->param('nom')    eq " " ||
       $req->param('prenom') eq " " ) {
   print $req->h1('Il faut remplir les champs');
   }
   else {
     print ("Requête numéro : ", $Cptr, $req->br);
     print ("Nom : ", $req->param('nom'),
            $req->br,
            "Prenom : ", $req->param('prenom'),
            $req->br,
            " vous utilisez", $req->user_agent(),
            "depuis la machine ",$req->remote_addr());
   }
}
else {
print $req->h1('Hum ... Que faites vous !');
}
print $req->end_html;
}
```

Chapitre 19

La programmation d'applications réseaux

19.1 Introduction

Perl permet le développement d'applications utilisant les *sockets*. Il s'agit de canaux de communications utilisant un couple *numéro IP/numéro de port* pour mettre en rapport deux machines. Le numéro IP cible une machine sur le réseau, le numéro de port adresse une application sur la machine référencée. On peut faire communiquer des processus distants en utilisant des *sockets* TCP ou UDP. TCP assure un service de communication fiable entre applications (remise en ordre des paquets, demande de réémission des paquets corrompus), UDP est beaucoup plus léger (pas de contrôles sur la validité des paquets) et est utilisé dans des cas bien spécifiques (transmission audios et vidéos où les contrôles sur les paquets n'ont pas de sens, ...).

On peut également utiliser les *sockets* pour faire communiquer des processus résidant sur une même machine (*sockets* dites Unix), pour plus de détails, consulter le manuel des fonctions socket, bind, listen, accept ...

19.2 Un exemple d'application client/serveur

L'exemple suivant permet de faire communiquer un serveur et des clients en utilisant une *socket* TCP. Le serveur est à l'écoute sur le port 8888, il reçoit et stocke des messages en provenance de clients distants.

Le code qui suit n'est pas à sortir du contexte de cet exemple, une véritable application devrait utiliser un service TCP enregistré, acquitter les messages reçus, ...

```
#!/usr/local/bin/perl
# Serveur qui réceptionne sur le port 8888 des messages
# qu'il stocke dans un fichier.

use Socket;
my $port = 8888;
my $protoc = getprotobyname('tcp');

open (FICLOG,">Log") || die ("Erreur Création Log : $!");
# Création de la socket TCP de reception des messages
socket (SockPerm, PF_INET, SOCK_STREAM, $protoc)
       || die("Erreur Creation Socket : $!");
setsockopt (SockPerm, SOL_SOCKET, SO_REUSEADDR, 1)
       || die ("setsockopt : $!");
bind (SockPerm, sockaddr_in($port, INADDR_ANY))
       || die("Erreur bind : $!");
listen (SockPerm, SOMAXCONN);

$SIG{INT}='arret';   # Fermer le fichier des messages si ^C
# Réception des messages
while(1) {
    $refcli = accept(SockTmp,SockPerm);
    my ($port, $addr) = sockaddr_in($refcli);
    $client=gethostbyaddr($addr, AF_INET);
    #
    # Traitement d'un message
    #
    $ligne=<SockTmp>;
    chop($ligne);
    until ($ligne =~ /^FIN/) {
       print FICLOG ("$client : ",$ligne,"\n");
       $ligne=<SockTmp>;
       chop($ligne);
    }
    close(SockTmp);
}
sub arret {
    close(FICLOG); exit;
}
```

```
#!/usr/local/bin/perl
#
# Client qui emet les messages.
#
use Socket;

my $port = 8888;
my $protoc = getprotobyname('tcp');
my $adrserv = 'localhost';   # machine locale pour les tests
my $iaddr = inet_aton($adrserv);
my $paddr = sockaddr_in($port,$iaddr);
#
# création de la socket et connexion au serveur
#
socket (SOCK, PF_INET, SOCK_STREAM, $protoc)
       || die ("Erreur Creation Socket: $!");
connect(SOCK, $paddr) || die ("Erreur Connect: $!");
#
# Saisie et émission ligne à ligne du message
#
while (<STDIN>) {
    print SOCK ("$_");
}
#
# Emission de la marque de fin de message
#
print SOCK ("FIN \n");
close(SOCK);
```

Bibliographie

[1] Larry WALL Tom CHRISTIANSEN et Randal L. SCHWARTZ.
Programmation en Perl, 2ème édition en français.
O'REILLY, 1996. ISBN 2-84177-004-4.

[2] Sriram SRINIVASAN.
Advanced Perl Programming.
O'REILLY, 1997. ISBN 1-56592-220-4.

[3] Randal L. SCHWARTZ Erik OLSON and Tom CHRISTIANSEN.
Learning Perl on Win32 Systems.
O'REILLY, 1997. ISBN 1-56592-324-3.

[4] Bruno POULIQUEN.
Introduction au langage Perl.
http://www.med.univ-rennes1.fr/~poulique/cours/perl/.

[5] Olivier AUBERT.
Introduction à perl.
http://perso-info.enst-bretagne.fr/~aubert/perl/html/perl.html.