ENSTA ParisTech
École nationale supérieure de techniques avancées

Le langage Perl

Marc Baudoin
<babafou@babafou.eu.org>

1 Introduction

Il existe de nombreux langages de programmation. Certains sont adaptés à une tâche bien particulière, d'autres sont généralistes et permettent de réaliser tout type de programme. Cependant, la plupart de ces langages généralistes sont d'un usage assez lourd. Par exemple, en C, langage généraliste par excellence, il faut que le programmeur s'occupe lui-même de l'allocation mémoire, ce qui peut vite se révéler très pénible. De même, le traitement de chaînes de caractères en C est souvent cauchemardesque.

C'est en partant de ce constat qu'a été créé le langage Perl, qui est un acronyme pour *Practical Extraction and Report Language*. Perl se veut un langage généraliste, mais plus pratique que le C pour tout ce qui concerne le traitement de données.

Perl n'est pas un langage compilé, comme le C, mais un langage semi-interprété. En effet, il n'est pas interprété ligne par ligne, comme le BASIC, mais il est d'abord transformé en un code intermédiaire par l'interpréteur, code qui est ensuite exécuté. Ce système a de nombreux avantages. En particulier, les erreurs de syntaxe sont détectées avant l'exécution. De plus, comme Perl utilise un code intermédiaire assez efficace, un script [1] Perl n'est pas beaucoup plus lent qu'un programme C équivalent.

Perl, dont la syntaxe est très proche de celle du C (ce qui permet de l'apprendre rapidement) s'est vite développé et est devenu le langage préféré, voire fétiche, de tous ceux qui doivent régulièrement écrire des petits programmes de traitement de données.

Ces quelques lignes proposent une rapide introduction au langage Perl, afin de montrer ses qualités et de mettre en évidence l'intérêt qu'il peut avoir.

Terminons cette introduction par le slogan de Perl, qui reflète bien sa philosophie :

There's more than one way to do it.

1. On parle généralement de script plutôt que de programme pour un langage interprété.

2 Mon premier script Perl

Prenez votre éditeur de texte préféré et tapez le script qui suit.

```
1 $toto = 'coucou' ;
2 print "toto = $toto\n" ;
```

Sauvez-le sous le nom `toto.pl` (les scripts Perl ont généralement l'extension `.pl`) et lancez-le en tapant :

```
perl toto.pl
```

Vous devriez voir apparaître à l'écran :

```
toto = coucou
```

Nous allons maintenant analyser le script en détail. La première ligne affecte la chaîne de caractères « `coucou` » (les apostrophes délimitent la chaîne de caractères et n'en font pas partie) à la variable `$toto`. Vous pouvez remarquer qu'en Perl les noms de variables commencent par un `$` et que, contrairement au C, il n'y a pas besoin de déclarer les variables qu'on va utiliser. En Perl, les lignes se terminent par un `;` comme en C.

La seconde ligne affiche le texte « `toto =` », puis le contenu de la variable `toto` (`$toto` est remplacé par « `coucou` », on appelle ceci la *substitution de variable*) suivi d'un retour à la ligne (`\n`, comme en C).

Les choses auraient été différentes si l'on avait écrit :

```
1 $toto = 'coucou' ;
2 print 'toto = $toto\n' ;
```

La seule différence entre les deux scripts est l'emploi d'apostrophes à la place des guillemets dans la seconde ligne. Ceci aurait affiché littéralement et sans retour à la ligne :

```
toto = $toto\n
```

En Perl, il y a deux façons de délimiter les chaînes de caractères :

- avec des guillemets, auquel cas les substitutions de variables (c'est-à-dire le remplacement de `$nom_de_la_variable` par le contenu de cette variable) sont effectuées et les séquences de contrôle (telles que `\n`) sont interprétées ;
- avec des apostrophes, auquel cas la chaîne est laissée telle quelle, sans substitution de variable ni interprétation des séquences de contrôle.

Bien entendu, une chaîne qui ne contient ni substitution de variable ni séquence de contrôle peut être délimitée indifféremment par des guillemets ou des apostrophes mais on utilise généralement les apostrophes, afin d'éviter toute substitution non désirée.

3 Lancement direct d'un script Perl

Sous UNIX, il y a un moyen assez pratique pour lancer des scripts. Modifiez donc `toto.pl` en lui ajoutant en première ligne :

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

Selon la version d'UNIX utilisée, le chemin d'accès absolu du programme `perl` peut varier. Par exemple, sous NetBSD, il s'agit de `/usr/pkg/bin/perl`. Sur d'autres systèmes, il pourra s'agir de `/usr/local/bin/perl`.

Notre script `toto.pl` ressemble donc maintenant à ceci :

```
1 #! /usr/bin/perl
2
3 $toto = 'coucou' ;
4 print "toto = $toto\n" ;
```

Rendez-le exécutable par la commande :

```
chmod 755 toto.pl
```

ou

```
chmod a+x toto.pl
```

puis tapez simplement :

```
./toto.pl
```

Le script est alors lancé comme précédemment.

En effet, lorsqu'il lance un fichier exécutable, UNIX vérifie de quel type de fichier il s'agit en examinant ses deux premiers octets. S'il s'agit des codes ASCII de `#` et `!`, il sait alors que le fichier est un script et pas un véritable programme. Donc, au lieu d'exécuter le script, ce qui n'a aucun sens puisque un script contient du texte et pas des instructions pour le microprocesseur, il lance le programme dont le nom suit les caractères `#!` avec pour argument le nom du fichier qui contient le script.

Donc, lorsqu'on tape :

```
./toto.pl
```

UNIX le comprend en fait comme :

```
/usr/bin/perl ./toto.pl
```

ce qui revient à ce que l'on a fait au début (en effet, le programme `perl` se trouve généralement dans le répertoire `/usr/bin`, qui est dans le chemin de recherche des commandes standard).

4 Quelques précautions

Perl est un langage qui entrave rarement la liberté du programmeur. Il offre néanmoins certains garde-fous qu'il est bon d'utiliser systématiquement.

Ainsi, la directive :

```
4 use warnings ;
```

placée en début de programme demande à Perl d'afficher le plus d'avertissements possibles lorsqu'il détecte des problèmes potentiels.

La directive :

```
3 use strict ;
```

quant à elle impose la déclaration des variables. Celle-ci n'est normalement pas obligatoire mais déclarer les variables permet également d'éviter certaines erreurs.

La déclaration d'une variable se fait simplement grâce à l'instruction **my** :

```
1 my $toto ;
```

Comme il est possible de déclarer une variable à n'importe quel endroit du programme, il est préférable de la déclarer et de l'initialiser en même temps :

```
6 my $toto = 'coucou' ;
```

Notre programme ressemble donc maintenant à ceci :

```
1 #! /usr/bin/perl
2
3 use strict ;
4 use warnings ;
5
6 my $toto = 'coucou' ;
7 print "toto = $toto\n" ;
```

L'ordre d'utilisation des directives est sans importance. Cependant, les indiquer dans l'ordre alphabétique permet de s'y retrouver rapidement si l'on en utilise beaucoup (ce qui peut arriver).

Dans ce qui suit, afin d'aller à l'essentiel, les noms des fichiers contenant les scripts ne seront plus mentionnés et il ne vous sera plus indiqué comment les lancer. Il est préférable de toujours faire débuter les scripts par les lignes :

```
1 #! /usr/bin/perl
2
3 use strict ;
4 use warnings ;
```

de les rendre exécutable et de les lancer tels quels. Les exemples de scripts Perl qui suivront ne comporteront pas ces lignes, n'oubliez pas de les rajouter.

5 Les variables scalaires

Dans le premier exemple, nous avons initialisé et affiché le contenu de la variable `$toto`. Ce type de variable est appelé *variable scalaire*, et peut contenir un nombre (entier ou flottant) ou une chaîne de caractères (ou encore une référence, qui est à Perl ce que le pointeur est au C mais que nous aborderons plus tard). On reconnaît une variable scalaire au fait que son nom commence par un `$`. C'est d'ailleurs un principe général en Perl, chaque type de variable (nous en verrons d'autres un peu plus loin) est introduit par un caractère particulier.

Quelques exemples d'initialisation de variables scalaires :

```
1  my $entier = 5 ;
2  my $decimal = 6.5 ;
3  my $chaine = 'coucou' ;
4  my $autre_chaine = "entier = $entier\n" ;
```

6 Les opérateurs arithmétiques

Perl possède les mêmes opérateurs arithmétiques que le C. Le script suivant montre comment utiliser les différents opérateurs arithmétiques de Perl.

```
1  my $a = 7.2 ;
2  my $b = 3 ;
3  my $x ;            # variable déclarée mais non initialisée
4
5  $x = $a + $b ;     # addition
6  print "x = $x\n" ;
7
8  $x = $a - $b ;     # soustraction
9  print "x = $x\n" ;
10
11 $x = $a * $b ;     # multiplication
12 print "x = $x\n" ;
13
14 $x = $a / $b ;     # division
15 print "x = $x\n" ;
16
17 $x = $a ** $b ;    # puissance (n'existe pas en C)
18 print "x = $x\n" ;
19
20 $a = 7 ;
21 $b = 3 ;
22
23 $x = $a % $b ;     # modulo (reste de la division entière)
24 print "x = $x\n" ;
```

Ce script affiche :

```
x = 10.2
x = 4.2
x = 21.6
x = 2.4
x = 373.248
x = 1
```

Vous remarquerez au passage que les commentaires en Perl commencent par un # et se terminent à la fin de la ligne.

Perl possède également les opérateurs d'incrémentation, de décrémentation et les raccourcis du C.

```
1  my $x = 5 ;
2
3  $x ++ ;     # incrémentation
4
5  $x -- ;     # décrémentation
6
7  $x += 5 ;   # équivalent à $x = $x + 5
8
9  $x -= 3 ;   # équivalent à $x = $x - 3
10
11 $x *= 2 ;   # équivalent à $x = $x * 2
12
13 $x /= 4 ;   # équivalent à $x = $x / 4
14
15 $x %= 3 ;   # équivalent à $x = $x % 3
```

7 Les tableaux et les listes

Comme tout langage évolué, Perl possède des variables de type tableau. Les tableaux sont créés automatiquement lorsqu'on affecte un de leurs éléments et leur taille s'adapte au nombre d'éléments sans qu'on ait à s'occuper de l'allocation mémoire. Comme en C, les indices des tableaux commencent à 0.

```
1  my @tab ;   # nous verrons plus tard à quoi cela correspond
2
3  $tab[0] = 4 ;
4  $tab[1] = 'bonjour' ;
5  $tab[2] = 5.6 ;
6  print "$tab[0] $tab[1] $tab[2]\n" ;
```

Ce script affiche :

```
4 bonjour 5.6
```

Chaque élément du tableau est une variable scalaire, c'est pourquoi leur nom commence par un `$`.

En Perl, les tableaux peuvent contenir des trous, c'est-à-dire que certains indices peuvent ne pas exister.

```
1 my @tab ;
2
3 $tab[0] = 4 ;
4 $tab[2] = 5.6 ;
5 print "$tab[0] $tab[2]\n" ;
```

La variable spéciale `$#nom_du_tableau` représente l'indice du dernier élément du tableau.

```
1 my @tab ;
2
3 $tab[0] = 4 ;
4 $tab[1] = 'bonjour' ;
5 $tab[2] = 5.6 ;
6 print "$#tab\n" ;
```

affiche :

```
2
```

Il est possible d'utiliser des indices négatifs. `$tab[-1]` est le dernier élément du tableau, `$tab[-2]`, l'avant-dernier, etc.

```
1 my @tab ;
2
3 $tab[0] = 4 ;
4 $tab[1] = 'bonjour' ;
5 $tab[2] = 5.6 ;
6 print "$tab[-1] $tab[-2] $tab[-3]\n" ;
```

affiche :

```
5.6 bonjour 4
```

En fait, on peut affecter directement l'ensemble du tableau grâce à une *liste*. Une liste est un ensemble de variables scalaires, séparées par des virgules, le tout entre parenthèses. Lors de cette affectation, on fait précéder le nom du tableau par `@`, qui permet d'introduire les tableaux de manière globale de même que `$` permet d'introduire les variables scalaires. `@tab` désigne donc le tableau tout entier.

```
my @tab = ( 4 , 'bonjour' , 5.6 ) ;
print "$tab[0] $tab[1] $tab[2]\n" ;
```

affiche :

```
4 bonjour 5.6
```

On peut même faire directement :

```
my @tab = ( 4 , 'bonjour' , 5.6 ) ;
print "@tab\n" ;
```

puisque `@tab` représente `$tab[0]` jusqu'à `$tab[$#tab]`.

En Perl, listes et tableaux sont équivalents et on emploie l'un ou l'autre dans les mêmes conditions indifféremment. Somme toute, un tableau n'est qu'un nom pour représenter une liste et une liste n'est que l'ensemble des éléments d'un tableau.

Les listes sont très pratiques et sont donc très utilisées en Perl. En effet, contrairement au C, les fonctions en Perl (voir le paragraphe 13) peuvent renvoyer plusieurs valeurs (c'est-à-dire une liste ou un tableau, c'est équivalent). Par exemple, la fonction **`split`** découpe son second argument (une chaîne de caractères) aux endroits qui correspondent à son premier argument (une expression rationnelle, voir le paragraphe 11).

```
my ( $prenom , $nom ) = split ( /:/ , 'Séraphin:Lampion' ) ;
print "$prenom $nom\n" ;
```

affiche :

```
Séraphin Lampion
```

On aurait également pu faire :

```
my @tab = split ( /:/ , 'Séraphin:Lampion' ) ;
print "@tab\n" ;
```

puisque listes et tableaux sont équivalents.

Citons une dernière utilisation des listes, l'affectation multiple, qui permet d'affecter plusieurs variables en une seule ligne.

```
my ( $a , $b , $c ) = ( 4 , 'bonjour' , 5.6 ) ;
```

Ce script affecte respectivement 4, `'bonjour'` et 5.6 aux variables `$a`, `$b` et `$c`.

8 Les tableaux associatifs

Les tableaux associatifs sont des tableaux dont les indices sont des chaînes de caractères, et pas des entiers. C'est très pratique pour programmer simplement des bases de données.

```
my %assoc ;   # nous verrons plus tard à quoi cela correspond

$assoc{'nom'}    = 'Lampion' ;
$assoc{'prénom'} = 'Séraphin' ;

print "Nom    : $assoc{'nom'}\n" ;
print "Prénom : $assoc{'prénom'}\n" ;
```

affiche :

```
Nom    : Lampion
Prénom : Séraphin
```

On peut bien entendu utiliser une liste pour affecter un tableau associatif. Chaque élément de la liste est de la forme « `indice => valeur` ». Le tableau associatif tout entier est spécifié en faisant précéder son nom de `%`.

```
my %assoc = ( 'nom'    => 'Lampion' ,
              'prénom' => 'Séraphin' ) ;

print "Nom    : $assoc{'nom'}\n" ;
print "Prénom : $assoc{'prénom'}\n" ;
```

9 Les structures de contrôle de flux

Les structures de contrôle de flux permettent de contrôler l'exécution d'un script, d'exécuter telle ou telle partie de code suivant la valeur de certaines variables ou de répéter une même séquence tant qu'une condition est satisfaite.

9.1 `if`, `elsif`, `else`, `unless`

L'instruction **`if`** permet d'exécuter la série d'instructions entre accolades si une condition est satisfaite.

```
my $a = 5 ;

if ( $a == 5 )
{
   print "Il y a 5 lapins dans mon chapeau.\n" ;
}
```

Ici, le bloc d'instructions entre accolades est exécuté si la variable `$a` est égale à 5.

Les accolades sont obligatoires, même s'il n'y a qu'une seule instruction à exécuter (en C, elles sont alors optionnelles).

Les opérateurs de comparaison numérique sont identiques à ceux du C :

`==`	égalité
`!=`	différence
`>`	supérieur strict
`<`	inférieur strict
`<=`	inférieur ou égal
`>=`	supérieur ou égal

Pour comparer des chaînes de caractères, il faut utiliser les opérateurs `eq` (égalité) et `ne` (différence) :

```
1  my $c = 'coucou' ;
2
3  if ( $c eq 'coucou' )
4  {
5     print "coucou\n" ;
6  }
```

On peut regrouper plusieurs tests à l'aide des opérateurs `&&` (et) et `||` (ou) :

```
1  my $a = 5 ;
2  my $b = 3 ;
3
4  if ( $a == 5 && $b == 3 )
5  {
6     print "a vaut 5 et b vaut 3\n" ;
7  }
```

Les opérateurs `and` et `or` s'utilisent de la même façon mais ont une priorité moindre que celle de `&&` et `||`. Pour cette raison, ils sont beaucoup plus souvent utilisés.

Si la condition n'est pas satisfaite, on peut exécuter une autre série d'instructions, placée dans la clause **`else`** :

```
1  my $a = 4 ;
2
3  if ( $a == 5 )
4  {
5     print "a vaut 5\n" ;
6  }
7  else
8  {
9     print "a est différent de 5\n" ;
10 }
```

Enfin, on peut enchaîner plusieurs tests grâce à l'instruction **`elsif`** :

```
my $a = 3 ;

if ( $a == 1 )
{
   print "a vaut 1\n" ;
}
elsif ( $a == 2 )
{
   print "a vaut 2\n" ;
}
elsif ( $a == 3 )
{
   print "a vaut 3\n" ;
}
else
{
   print "Je ne sais pas !\n" ;
}
```

L'instruction **unless** est le contraire de l'instruction **if**, la série d'instructions entre accolades est exécutée si la condition n'est pas satisfaite :

```
my $a = 4 ;

unless ( $a == 5 )
{
   print "Il n'y a pas 5 lapins dans mon chapeau.\n" ;
}
```

Ici, le bloc d'instructions entre accolades est exécuté si la variable `$a` n'est pas égale à 5.
Lorsque le test est simple, on peut aussi écrire :

```
my $a = 5 ;

print "Il y en a 5.\n" if ( $a == 5 ) ;
print "Il n'y en a pas 5.\n" unless ( $a == 5 ) ;
```

9.2 while, until

L'instruction **while** permet d'exécuter le bloc d'instructions entre accolades tant qu'une condition est satisfaite :

```
my $a = 0 ;

while ( $a < 5 )
```

```
{
   print "Pas encore.\n" ;
   $a ++ ;
}
```

Le bloc d'instructions entre accolades est exécuté tant que `$a` est inférieur à 5.

L'instruction **`until`** permet d'exécuter le bloc d'instructions entre accolades jusqu'à ce qu'une condition soit satisfaite :

```
my $a = 0 ;

until ( $a >= 5 )
{
   print "Pas encore.\n" ;
   $a ++ ;
}
```

Le bloc d'instructions entre accolades est exécuté jusqu'à ce que `$a` soit supérieur ou égal à 5.

9.3 do

Dans les boucles **`while`** et **`until`**, le test est effectué avant d'exécuter le bloc d'instructions, si bien que ce bloc n'est pas exécuté si la condition n'est pas satisfaite d'entrée de jeu. Dans certains cas, on aimerait que le test soit effectué après l'exécution du bloc d'instructions, ce qui garantirait que celui-ci soit exécuté au moins une fois. Pour cela, on peut utiliser les boucles **`do`** :

```
my $a = 0 ;

do
{
   print "Pas encore.\n" ;
   $a ++ ;
}
while ( $a < 5 ) ;
```

Il existe de même une boucle **`do`** ... **`until`** :

```
my $a = 0 ;

do
{
   print "Pas encore.\n" ;
   $a ++ ;
}
until ( $a >= 5 ) ;
```

9.4 foreach

L'instruction **foreach** permet d'affecter successivement à une variable chaque élément d'une liste et, à chaque fois, d'exécuter un bloc d'instructions.

```
foreach my $i ( 1 , 'coucou' , 5.6 )
{
   print "$i\n" ;
}
```

affiche :

```
1
coucou
5.6
```

On peut aussi utiliser une liste :

```
my @tab = ( 1 , 'coucou' , 5.6 ) ;

foreach my $i ( @tab )
{
   print "$i\n" ;
}
```

L'opérateur .. permet de générer facilement un intervalle de nombres entiers.

```
foreach my $i ( 1 .. 5 )
{
   print "$i\n" ;
}
```

affiche :

```
1
2
3
4
5
```

9.5 for

La boucle **for** de Perl est identique à celle du C :

```
for ( my $i = 0 ; $i < 5 ; $i ++ )
{
   print "$i\n" ;
}
```

9.6 `next` et `last`, les boucles nommées

Lorsqu'on est dans une boucle, il peut être utile de passer à l'itération suivante sans attendre la fin du bloc d'instructions. L'instruction **`next`** permet de le faire :

```
1  while ( $a != 5 )
2  {
3     # ...
4
5     if ( $b == 3 )
6     {
7        next ;   # itération suivante
8     }
9
10    # ...
11 }
```

C'est l'équivalent de **`continue`** en C.

De même, il peut être utile de pouvoir sortir immédiatement d'une boucle, ce que permet l'instruction **`last`** :

```
1  while ( $a != 5 )
2  {
3     # ...
4
5     if ( $b == 3 )
6     {
7        last ;   # on sort de la boucle
8     }
9
10    # ...
11 }
```

C'est l'équivalent de **`break`** en C.

Cependant, dans le cas où plusieurs boucles sont imbriquées, ces instructions ne permettent d'agir que sur la boucle la plus interne. Parfois, on aimerait pouvoir agir sur l'une des boucles externes, ce qui n'est pas possible en C (ou alors il faut recourir à un affreux **`goto`**). En Perl, il suffit de donner un nom aux boucles sur lesquelles on souhaite agir.

```
1  EXTERNE: while ( $a != 5 )   # cette boucle s'appelle EXTERNE
2  {
3     # ...
4
5     while ( $b != 3 )
6     {
7        # ...
8
```

```
9       if ( $c == 7 )
10      {
11         last EXTERNE ;   # on sort de la première boucle
12      }
13
14      # ...
15   }
16
17   # ...
18 }
```

10 Les fichiers

Le traitement des informations contenues dans des fichiers est l'un des buts premiers de Perl. Les opérations de lecture et d'écriture de fichiers sont donc extrêmement simples à réaliser.

10.1 Traitement de la ligne de commande

Perl permet très simplement de lire ligne par ligne les fichiers dont les noms sont indiqués sur la ligne de commande. Considérons le script suivant, que l'on appelera `cat.pl` :

```
1 while ( my $ligne = <> )
2 {
3    print "$ligne" ;
4 }
```

Exécutons-le comme ceci :

```
./cat.pl toto tata
```

où `toto` et `tata` sont deux fichiers. Le résultat de cette commande est que le contenu de ces fichiers est affiché à l'écran, comme si l'on avait utilisé la commande `cat`.

En fait, toute la magie de ce script réside dans l'expression <>. Lorsqu'on l'utilise, Perl considère tous les arguments du script comme des noms de fichiers et <> renvoie successivement chaque ligne du premier fichier, puis du deuxième, etc. C'est un moyen très pratique pour traiter les fichiers. Si aucun argument n'a été spécifié sur la ligne de commande, <> renvoie chaque ligne de l'entrée standard.

On pourrait même écrire ce script plus simplement :

```
1 while ( <> )
2 {
3    print ;
4 }
```

On utilise ici une particularité de Perl, la variable implicite `$_`. De nombreuses instructions et fonctions utilisent cette variable si on ne leur en fournit pas une, ce qui est le cas dans cet exemple. Celui-ci est équivalent à :

```
while ( $_ = <> )
{
   print "$_" ;
}
```

mais la forme utilisant la variable implicite est plus simple et, avec un peu d'habitude, plus lisible, pour peu que la séquence d'instructions l'utilisant soit courte.

Notez qu'on ne déclare jamais la variable implicite au moyen de l'instruction **my**.

10.2 Lecture

```
open ( FICHIER , 'fichier' ) ;

while ( my $ligne = <FICHIER> )
{
   # faire quelque chose
}

close ( FICHIER ) ;
```

Avant de pouvoir faire quoi que ce soit avec un fichier, il faut tout d'abord l'ouvrir. C'est le rôle de la fonction **open**. Son premier argument est un *descripteur de fichier*, qui est initialisé par **open** et qui permettra de lire le fichier par la suite. Son second argument est le nom du fichier à ouvrir.

Ensuite, il est très simple de lire le fichier ligne par ligne, grâce à l'expression `<FICHIER>`, qui renvoie successivement chaque ligne du fichier.

Enfin, le fichier est refermé par la fonction **close**, qui prend en argument le descripteur de fichier initialisé par **open**.

En ouvrant le fichier, on peut ajouter un test permettant de quitter le script (ou de faire quelque chose de plus utile que de planter) si le fichier ne peut pas être ouvert.

```
my $fichier = 'fichier' ;

open ( FICHIER , $fichier )
   or die ( "Impossible d'ouvrir $fichier : $!\n" ) ;

while ( my $ligne = <FICHIER> )
{
   # faire quelque chose
}

```

```
11 close ( FICHIER ) ;
```

Si le fichier ne peut pas être ouvert, la suite de la ligne (après `or`) est exécutée. La fonction **`die`** sort du script en imprimant le message d'erreur qui la suit (`$!` est substitué par le message d'erreur qui correspond à la situation).

10.3 Écriture

```
1  my $fichier = 'fichier' ;
2
3  open ( FICHIER , ">$fichier" )
4     or die ( "Impossible d'ouvrir $fichier : $!\n" ) ;
5
6  # ...
7
8  print FICHIER "Des choses utiles.\n" ;
9
10 # ...
11
12 close ( FICHIER ) ;
```

Comme pour la lecture, on commence par ouvrir le fichier grâce à **`open`**, sauf qu'on ajoute le caractère > avant le nom du fichier pour indiquer qu'on souhaite l'ouvrir en écriture (c'est analogue à l'interpréteur de commande, où > indique la redirection de la sortie standard).

L'écriture dans le fichier se fait au moyen de **`print`**, en indiquant le descripteur de fichier avant la chaîne à écrire.

Enfin, le fichier est refermé par la fonction **`close`**.

11 Les expressions rationnelles

Les expressions rationnelles permettent de vérifier le format d'une chaîne de caractères et d'en extraire des parties selon certains critères. Elles sont abondamment utilisées en Perl.

```
1  open ( FICHIER , 'fichier' ) ;
2
3  while ( <FICHIER> )
4  {
5     if ( /^total/ )
6     {
7        print ;
8     }
9  }
10
11 close ( FICHIER ) ;
```

Chaque ligne du fichier est testée par rapport à l'expression rationnelle `^total` (qui teste si elle commence par `total`, `^` correspondant au début de la ligne). Si oui, la ligne est affichée. Vous remarquerez que ce script utilise beaucoup la variable implicite `$_`.

Étudions un exemple un peu plus complexe :

```
open ( FICHIER , 'fichier' ) ;

while ( <FICHIER> )
{
   if ( /^(...) (\d\d):(\d\d):(\d\d)$/ )
   {
      my ( $jour , $heure , $minute , $seconde )
         = ( $1 , $2 , $3 , $4 ) ;

      # ...
   }
}

close ( FICHIER ) ;
```

Les parenthèses servent à mémoriser certaines parties de la chaîne pour un traitement ultérieur. Cette expression rationnelle correspond aux chaînes commençant (^) par trois caractères quelconques (un point correspond à n'importe quel caractère), suivis d'un espace, de deux chiffres (`\d` correspond à un chiffre), de :, de deux chiffres, de : et de deux chiffres. Le `$` final correspond à une fin de la ligne et indique que la chaîne doit se terminer par les deux derniers chiffres sans rien d'autre après. La première sous-chaîne mémorisée est stockée dans `$1`, la deuxième dans `$2` et ainsi de suite. Comme ces variables ont une durée de vie très limitée, il faut tout de suite les recopier dans d'autres variables pour pouvoir ensuite en faire quelque chose.

On peut se passer des variables intermédiaires de cette façon :

```
open ( FICHIER , 'fichier' ) ;

while ( <FICHIER> )
{
   if ( my ( $jour , $heure , $minute , $seconde )
           = /^(...) (\d\d):(\d\d):(\d\d)$/ )
   {
      # ...
   }
}

close ( FICHIER ) ;
```

De nombreuses séquences permettent d'identifier différentes informations dans des chaînes de caractères :

`^`	début de chaîne
`$`	fin de chaîne
`.`	un caractère quelconque
`\w`	un caractère alphanumérique
`\W`	un caractère non-alphanumérique
`\s`	un blanc (espace ou tabulation)
`\S`	un caractère qui n'est pas un blanc
`\d`	un chiffre
`\D`	un caractère qui n'est pas un chiffre

On peut également être plus précis en précisant de manière exacte ce que l'on recherche :

```
1 if ( my ( $nucleotide ) = /^([atgc])$/ )
2 {
3    # ...
4 }
```

Ici l'on recherche les lignes ne contenant qu'un caractère parmi les lettres a, t, g ou c. Les crochets permettent de spécifier l'espace de recherche.

On peut également élargir la recherche :

```
1 if ( my ( $nucleotide ) = /^([atgc]+)$/ )
2 {
3    # ...
4 }
```

Le + ajouté au motif de recherche s'applique à ce qui le précède immédiatement et indique que ceci peut être répété plusieurs fois. Ainsi on recherche ici les lignes contenant une combinaison quelconque des lettres a, t, g et c. Mais il en faut au moins une. On utilise $*$ si le motif peut être inexistant :

```
1 if ( my ( $nucleotide ) = /^([atgc]*)$/ )
2 {
3    # ...
4 }
```

Ici on détecte également les lignes vides.

12 Les formats

Les formats permettent de présenter facilement des informations sous une forme synthétique.

```
1 my ( $jour , $total ) ;
2
3 format FICHIER_TOP =
4 Jour     | Total
```

```
5  ---------+---------
6  .
7
8  format FICHIER =
9  @<<<<<<< | @####.##
10 $jour ,    $total
11 .
12
13 open ( FICHIER , '>fichier' ) ;
14
15 ( $jour , $total ) = ( 'Lundi'    , 1234.45 ) ;
16 write FICHIER ;
17 ( $jour , $total ) = ( 'Mardi'    , 6789.15 ) ;
18 write FICHIER ;
19 ( $jour , $total ) = ( 'Mercredi' ,   49.99 ) ;
20 write FICHIER ;
21
22 close ( FICHIER ) ;
```

Étudions tout d'abord la définition du format `FICHIER`. Elle comporte deux lignes. La première indique comment présenter le contenu des variables spécifiées dans la seconde. Un `@` commence une indication de présentation et est suivi d'un certain nombre de caractères, tous identiques, indiquant le type de présentation souhaité. Le nombre de ces caractères (y compris le `@` initial) indique la largeur du champ. Ces caractères peuvent être :

`>`	pour aligner à droite
`<`	pour aligner à gauche
`|`	pour centrer
`#`	pour aligner sur le point décimal

La définition du format se termine par un point, seul sur une ligne.

Une fois le fichier `FICHIER` ouvert en écriture, la fonction **`write`**, suivie du descripteur de fichier, permet d'y inscrire les valeurs de `$jour` et de `$total` conformément au format `FICHIER`. Il est nécessaire que le format et le descripteur de fichier aient le même nom.

Le script définit aussi le format `FICHIER\_TOP`, dont le contenu sera affiché avant le premier appel à **`write`**.

Après exécution du script, le fichier `fichier` contient

```
Jour     | Total
---------+---------
Lundi    |  1234.45
Mardi    |  6789.15
Mercredi |    49.99
```

13 Les sous-programmes

Comme tout langage de programmation digne de ce nom, Perl permet de définir des sous-programmes.

```
print "Début du programme\n" ;
sousprog ( ) ;
print "Fin du programme\n" ;

sub sousprog
{
   print "Appel au sous-programme.\n" ;
}
```

La définition du sous-programme `sousprog` commence par le mot-clé **`sub`**, suivi par le nom du sous-programme, puis par son corps, un bloc d'instructions.

Le sous-programme est appelé dans le programme principal comme une fonction.

Les définitions de sous-programmes peuvent apparaître à n'importe quel endroit dans un script, mais on préfère généralement les regrouper à la fin, après le programme principal.

Un sous-programme récupère les paramètres qui lui sont passés dans le tableau implicite `@_`.

```
addition ( 4 , 5 ) ;

sub addition
{
   my ( $a , $b ) = @_ ;
   my $c = $a + $b ;

   print "$a + $b = $c\n" ;
}
```

Enfin, le sous-programme peut renvoyer des valeurs (cela peut être une variable scalaire, un tableau, etc.) au programme appelant grâce à l'instruction **`return`**.

```
my ( $x , $y ) = ( 4 , 5 ) ;
my $z = addition ( $x , $y ) ;
print "$x + $y = $z\n" ;

sub addition
{
   my ( $a , $b ) = @_ ;

   return $a + $b ;
}
```

14 L'opérateur <<

Il est souvent utile d'insérer directement dans un programme Perl une chaîne de caractères contenant plusieurs lignes.

Dans ce cas, une chaîne traditionnelle est peu lisible :

```
1 my $html = "<p>\nCeci est un paragraphe HTML.\n</p>\n" ;
```

Il est alors très utile d'utiliser l'opérateur << :

```
1 my $html = << 'FIN' ;
2 <p>
3 Ceci est un paragraphe HTML.
4 </p>
5 FIN
```

Ici, la variable `$html` est initialisée avec les lignes qui suivent, jusqu'à la chaîne `FIN` (non comprise). C'est tout de même beaucoup plus lisible.

À la suite de l'opérateur <<, la chaîne `FIN` est entre apostrophes, ce qui signifie qu'aucune substitution ne sera faite dans les lignes suivantes. Si des substitutions sont nécessaires, il faut l'entourer de guillemets :

```
1 my $html = << "FIN" ;
2 <p>
3 Bienvenue $prenom $nom !
4 </p>
5 FIN
```

Et l'opérateur << peut s'utiliser partout ou l'on attend une chaîne de caractères :

```
1 execute_sql ( << "FIN" ) ;
2 SELECT nom , prenom
3 FROM utilisateurs
4 WHERE nom = $nom
5 FIN
```

Cet opérateur est évidemment à utiliser sans parcimonie.

15 Les références

Les références sont à Perl ce que les pointeurs sont au C. Ils permettent en particulier de construire des structures de données complexes.

En effet, les listes sont nécessairement plates. Une liste ne contient que des scalaires, elle ne peut pas contenir d'autres listes.

Ainsi :

```
1 my @liste = ( 1 , ( 2 , 3 , ( 4 , 5 ) ) ) ;
```

est équivalent à :

```
my @liste = ( 1 , 2 , 3 , 4 , 5  ) ;
```

Dans ces conditions, il est impossible de construire des listes imbriquées.

En revanche, on peut construire des listes contenant des références vers d'autres listes. Une référence est simplement une variable scalaire permettant d'accéder à une autre variable (scalaire, tableau ou tableau associatif).

Voyons un exemple simple de référence :

```
my $scalaire = 5 ;
my $reference = \$scalaire ;

print "$$reference\n" ;
```

Ici, `$reference` est une référence sur la variable `$variable`. On l'obtient en mettant un `\` devant le nom de la variable à référencer. Ensuite on utilise `$reference` en préfixant son nom d'un `$` supplémentaire.

On peut également référencer une liste :

```
my @liste = ( 1 , 2 , 3 ) ;
my $reference = \@liste ;

foreach my $element ( @$reference )
{
   print "$element\n" ;
}
```

Ou encore :

```
my @liste = ( 1 , 2 , 3 ) ;
my $reference = \@liste ;

foreach my $i (  0 .. $#$reference )
{
   print "$reference->[$i]\n" ;
}
```

Il est ainsi possible de construire des listes contenant non pas d'autres listes mais des références vers ces listes :

```
my @liste1 = ( 1 , 2 , 3 ) ;
my @liste2 = ( 4 , 5 , 6 ) ;
my @liste = ( \@liste1 , \@liste2 ) ;

foreach my $reference (  @liste )
{
   foreach my $element ( @$reference )
```

```
    {
        print "$element\n" ;
    }
}
```

Néanmoins, cette façon de faire nécessite la création de listes intermédiaires afin d'en créer des références. On peut créer directement des références anonymes en entourant les listes de crochets au lieu des parenthèses :

```
my $ref1 = [ 1 , 2 , 3 ] ;
my $ref2 = [ 4 , 5 , 6 ] ;
my @liste = ( $ref1 , $ref2 ) ;
```

Voire directement :

```
my @liste = ( [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ) ;
```

ou encore :

```
my $ref = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ] ;
```

Et l'on accède très simplement à un élément particulier :

```
my $ref = [ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ] ;

print "$ref->[0][1]\n" ;   # affiche 2
```

On peut également créer des référence sur des tableaux associatifs :

```
my %assoc = ( 'nom'    => 'Lampion' ,
              'prénom' => 'Séraphin' ) ;

my $reference = \%assoc ;

print "Nom    : $reference->{'nom'}\n" ;
print "Prénom : $reference->{'prénom'}\n" ;
```

Et, avec une référence anonyme :

```
my $reference = { 'nom'    => 'Lampion' ,
                  'prénom' => 'Séraphin' } ;

print "Nom    : $reference->{'nom'}\n" ;
print "Prénom : $reference->{'prénom'}\n" ;
```

16 Tableaux multidimensionnels

Il est très simple de définir des tableaux multidimensionnels directement, élément par élément, en utilisant des références et les facultés de gestion automatique de la mémoire de Perl. On peut même mélanger tableaux et tableaux associatifs :

```
1  my $tab ;
2
3  $tab->{'Jan'}[1] = 45 ;
4  $tab->{'Jan'}[2] = 34 ;
5
6  # ...
7
8  $tab->{'Dec'}[31] = 67 ;
9
10 foreach my $mois ( 'Jan' , 'Fev' , 'Mar' , 'Avr' ,
11                    'Mai' , 'Jun' , 'Jul' , 'Aou' ,
12                    'Sep' , 'Oct' , 'Nov' , 'Dec' )
13 {
14    foreach my $jour ( 1 .. 31 )
15    {
16       if ( defined ( $tab->{$mois}[$jour] ) )
17       {
18          print "$tab->{$mois}[$jour]\n" ;
19       }
20    }
21 }
```

17 Les modules

Les modules permettent d'étendre les capacités de Perl grâce à de nouveaux types de données et aux sous-programmes permettant de les manipuler.

Les types de données définis dans les modules sont toujours des références, auxquelles il est possible d'appliquer des méthodes selon les principes de la programmation orientée objet.

17.1 Utilisation

Par exemple, le module `Math::Complex` permet de manipuler des nombres complexes :

```
1  #! /usr/pkg/bin/perl
2
3  use strict ;
4  use warnings ;
5
```

```
6  use Math::Complex ;
7
8  my $z = Math::Complex->make ( 3 , 4 ) ;
9  my $a = $z->Re ( ) ;
10 my $b = $z->Im ( ) ;
11
12 print "Partie réelle     : $a\n" ;
13 print "Partie imaginaire : $b\n" ;
```

Pour utiliser un module, il est nécessaire de le charger de la même façon qu'une directive :

```
1  use Math::Complex ;
```

On peut ensuite créer une variable représentant le nombre complexe $3+4i$, variable que l'on appelle *objet* :

```
1  my $z = Math::Complex->make ( 3 , 4 ) ;
```

La façon de créer un objet dépend du module qu'on utilise.

Un objet n'est autre qu'une référence et s'utilise comme tel. On n'a généralement pas accès directement à la variable référencée et l'on doit utiliser des fonctions définies dans le module pour la manipuler. Par exemple, la fonction `Re`, appliquée directement à un objet, en renvoie la partie réelle :

```
1  my $a = $z->Re ( ) ;
```

Il n'est pas possible de décrire les modules de manière plus universelle parce que chacun d'eux a ses particularités. En revanche, ils sont généralement très bien documentés. Pour obtenir la documentation du module `Math::Complex`, il suffit de taper la commande suivante dans un interpréteur de commandes :

```
perldoc Math::Complex
```

17.2 Un autre exemple : les fichiers

La façon de lire et d'écrire dans les fichiers vue au paragraphe 10 est la façon historique de faire. Il existe maintenant un module permettant de faire la même chose de manière plus élégante.

17.2.1 Lecture

```
1  #! /usr/pkg/bin/perl
2
3  use strict ;
4  use warnings ;
5
```

```
6  use IO::File ;
7
8  my $fichier = 'fichier' ;
9
10 my $f = IO::File->new ( ) ;
11
12 $f->open ( $fichier )
13    or die ( "Impossible d'ouvrir $fichier : $!\n" ) ;
14
15 while ( my $ligne = <$f> )
16 {
17    # faire quelque chose
18 }
19
20 $f->close ( ) ;
```

17.2.2 Écriture

```
1  #! /usr/pkg/bin/perl
2
3  use strict ;
4  use warnings ;
5
6  use IO::File ;
7
8  my $fichier = 'fichier' ;
9
10 my $f = IO::File->new ( ) ;
11
12 $f->open ( ">$fichier" )
13    or die ( "Impossible d'ouvrir $fichier : $!\n" ) ;
14
15 # ...
16
17 print $f "Des choses utiles.\n" ;
18
19 # ...
20
21 $f->close ( ) ;
```

17.3 Programmation

Créer un module Perl est en fait assez simple. Il faut d'abord lui choisir un nom, si possible en suivant les règles de nommage du CPAN (voir le paragraphe 18). Nous allons donc créer notre propre module de traitement des nombres complexes et l'appeler `Amoi::Math::Complex`.

Ce module sera donc contenu dans un fichier `Amoi/Math/Complex.pm` (les noms placés avant les `::` sont transformés en répertoires et le dernier nom correspond au fichier contenant le module).

Un module commence toujours par une directive **`package`** suivie du nom du module :

```
1  package Amoi::Math::Complex ;
```

On peut ensuite inclure les directives classiques :

```
3  use strict ;
4  use warnings ;
```

Une méthode (habituellement appelée `new`) permet de créer un nouvel objet. On l'utilise ainsi dans le programme principal :

```
9  my $z = Amoi::Math::Complex->new ( 3 , 4 ) ;
```

et on la définit ainsi dans le module :

```
6  sub new
7  {
8     my ( $classe , $re , $im ) = @_ ;
9
10    my $complexe = [ $re , $im ] ;
11
12    bless ( $complexe , $classe ) ;
13
14    return $complexe ;
15 }
```

J'ai choisi ici de représenter un nombre complexe sous la forme d'une référence (puisqu'un objet est toujours une référence) sur une liste de deux éléments : sa partie réelle et sa partie imaginaire :

```
10    my $complexe = [ $re , $im ] ;
```

La fonction **`bless`** permet d'associer la référence `$complexe` au module qui a permis de la créer (et, par la suite, d'appeler les bonnes méthodes en cas d'utilisation simultanée dans un programme de modules ayant des méthodes homonymes). Le nom du module (utilisé comme second argument de la fonction **`bless`**) est récupéré en premier paramètre de la méthode `new`. Comment cela se fait-il alors que, dans le programme principal, on n'a appelé la méthode `new` qu'avec deux paramètres ? C'est parce que, dans le cas de l'appel d'une méthode de classe (non liée à un objet particulier, ce qui est le cas de `new`), Perl ajoute systématiquement avant les paramètres d'appel de la méthode le nom du module.

On termine la méthode new en renvoyant la référence sur l'objet nouvellement créé.

On peut également faire une méthode permettant d'afficher le contenu du nombre complexe :

```
17 sub affiche
18 {
19    my ( $complexe ) = @_ ;
20
21    print "$complexe->[0] + $complexe->[1]i" ;
22 }
```

Qui s'utilise ainsi :

```
11 $z->affiche ( ) ;
```

Bien qu'aucun paramètre ne soit passé à la méthode lors de son appel, Perl ajoute systématiquement en premier la référence sur l'objet auquel elle s'applique. La méthode peut ainsi accéder au contenu de la référence.

Enfin (c'est un vieil héritage), un module doit toujours se terminer par une valeur vraie. La plus simple, c'est 1 :

```
24 1 ;
```

Ce qui donne au total :

```
1  package Amoi::Math::Complex ;
2
3  use strict ;
4  use warnings ;
5
6  sub new
7  {
8     my ( $classe , $re , $im ) = @_ ;
9
10    my $complexe = [ $re , $im ] ;
11
12    bless ( $complexe , $classe ) ;
13
14    return $complexe ;
15 }
16
17 sub affiche
18 {
19    my ( $complexe ) = @_ ;
20
21    print "$complexe->[0] + $complexe->[1]i" ;
22 }
23
```

```
24 1 ;
```

Et le programme principal :

```
1  #! /usr/bin/perl
2
3  use strict ;
4  use warnings ;
5
6  use lib '/machin/chose' ;
7  use Amoi::Math::Complex ;
8
9  my $z = Amoi::Math::Complex->new ( 3 , 4 ) ;
10
11 $z->affiche ( ) ;
12
13 print "\n" ;
```

La directive **use** `lib` permet d'indiquer le chemin d'accès absolu du répertoire contenant `Amoi/Math/Complex.pm`. Dans notre exemple, celui-ci est donc `/machin/chose/Amoi/Math/Complex.pm`.

On peut améliorer la méthode `new` pour pouvoir l'utiliser comme méthode de classe ou d'instance :

```
9  my $z1 = Amoi::Math::Complex->new ( 3 , 4 ) ;
10 my $z2 = $z1->new ( ) ;   # $z2 vaut également 3 + 4i
```

Pour ceci, on modifie `new` de cette façon :

```
6  sub new
7  {
8     my ( $premier , $re , $im ) = @_ ;
9     my $complexe ;
10    my $classe ;
11
12    if ( ref ( $premier ) )
13       # $premier est une référence, méthode d'instance
14    {
15       $classe = ref ( $premier ) ;
16       $complexe = [ @$premier ] ;
17    }
18    else   # méthode de classe
19    {
20       $classe = $premier ;
21       $complexe = [ $re , $im ] ;
22    }
```

```
23
24      bless ( $complexe , $classe ) ;
25
26      return $complexe ;
27  }
```

La fonction **ref** permet de déterminer si le premier argument de new est une référence (auquel cas new a été appelée comme méthode d'instance) ou pas (auquel cas new a été appelée comme méthode de classe). Si `$premier` est un objet, **ref** `( $premier )` renvoie le nom du module correspondant.

Puisqu'il s'agit d'un module de traitement des nombres complexes, il serait pratique de pouvoir surcharger les opérateurs arithmétiques, par exemple l'addition :

```
29  use overload
30      '+' => \&addition ;
31
32  sub addition
33  {
34      my ( $z1 , $z2 ) = @_ ;
35
36      return Amoi::Math::Complex->new ( $z1->[0] + $z2->[0] ,
37                                        $z1->[1] + $z2->[1] ) ;
38  }
```

Et on peut alors écrire dans le programme principal :

```
9   my $z1 = Amoi::Math::Complex->new ( 3 , 4 ) ;
10  my $z2 = Amoi::Math::Complex->new ( 5 , 6 ) ;
11  my $z3 = $z1 + $z2 ;
```

On peut également surcharger l'opérateur d'affichage :

```
40  use overload
41      '""' => \&affiche ;
42
43  sub affiche
44  {
45      my ( $complexe ) = @_ ;
46
47      return "$complexe->[0] + $complexe->[1]i" ;
48  }
```

Dans le programme principal, on pourra donc écrire simplement :

```
13  my $z = Amoi::Math::Complex->new ( 3 , 4 ) ;
14
15  print "$z\n" ;
```

18 Comprehensive Perl Archive Network (CPAN)

L'une des grandes forces de Perl est de disposer d'une impressionnante bibliothèque de modules. Quoiqu'on veuille faire, il y a de fortes chances qu'un module réalisant 95 % du travail existe déjà. Ces modules sont regroupés dans le *Comprehensive Perl Archive Network*.

Le serveur CPAN principal est : `ftp://ftp.cpan.org/CPAN/`. Il existe de nombreux miroirs de ce serveur à travers le monde.

Le CPAN dispose également d'un serveur Web doté d'un moteur de recherche : `http://www.cpan.org/`.

19 Perl sur l'Internet

Plusieurs serveurs Web sont consacrés à Perl :

- `http://www.perl.org/`
- `http://www.perl.com/`

Des groupes USENET sont dédiés à Perl :

`comp.lang.perl.announce` est un groupe modéré dans lequel sont annoncées les nouveautés du monde Perl (nouvelles versions de Perl, nouveaux modules...).

`comp.lang.perl.misc` permet de discuter de Perl ;

`comp.lang.perl.modules` est spécialisé dans les discussions concernant les modules Perl ;

`fr.comp.lang.perl` est le groupe francophone dédié à Perl.

Références

[1] Sébastien APERGHIS-TRAMONI, Philippe BRUHAT, Damien KROTKINE et Jérôme QUELIN.
Perl moderne. L'essentiel des pratiques actuelles.
Le guide de survie.
Pearson, octobre 2010.
URL : http://perlmoderne.fr/.

[2] Tom CHRISTIANSEN, brian d FOY, Larry WALL et Jon ORWANT.
Programming Perl.
4e édition.
O'Reilly Media, février 2012.
URL : http://shop.oreilly.com/product/9780596004927.do.

[3] Tom CHRISTIANSEN et Nathan TORKINGTON.
Perl Cookbook.
2e édition.
O'Reilly Media, août 2003.
URL : http://shop.oreilly.com/product/9780596003135.do.

[4] CHROMATIC.
Modern Perl.
Onyx Neon Press, janvier 2012.
URL : http://www.onyxneon.com/books/modern_perl/.

[5] Damian CONWAY.
Perl Best Practices.
O'Reilly Media, juillet 2005.
URL : http://shop.oreilly.com/product/9780596001735.do.

[6] Simon COZENS.
Advanced Perl Programming.
2e édition.
O'Reilly Media, juin 2005.
URL : http://shop.oreilly.com/product/9780596004569.do.

[7] Mark Jason DOMINUS.
Higher-Order Perl.
Elsevier, 2005.
URL : http://hop.perl.plover.com/.

[8] brian d FOY.
Mastering Perl.
O'Reilly Media, juillet 2007.
URL : http://shop.oreilly.com/product/9780596527242.do.

[9] Joseph N. HALL, Joshua A. MCADAMS et brian d FOY.
Effective Perl Programming. Ways to Write Better, More Idiomatic Perl.
2e édition.
Addison-Wesley Professional, avril 2010.
URL : http://www.informit.com/store/product.aspx?isbn=0321496949.

[10] James LEE.
Beginning Perl.
3e édition.
Apress, avril 2010.
URL : http://www.apress.com/9781430227939.

[11] Randal L. SCHWARTZ, brian d FOY et Tom PHOENIX.
Intermediate Perl.
O'Reilly Media, mars 2006.
URL : http://shop.oreilly.com/product/9780596102067.do.

[12] Randal L. SCHWARTZ, brian d FOY et Tom PHOENIX.
Learning Perl.
6e édition.
O'Reilly Media, juin 2011.
URL : http://shop.oreilly.com/product/0636920018452.do.

[13] Johan VROMANS.
Perl Pocket Reference.
4e édition.
O'Reilly Media, juillet 2002.
URL : http://shop.oreilly.com/product/9780596003746.do.

Table des matières