École Nationale Supérieure des Télécommunications de Bretagne
École Supérieure d'Informatique-Electronique-Automatique

Introduction à perl

28 juin 2000

Olivier Aubert
Olivier.Aubert@enst-bretagne.fr
Ajouts par Emmanuel Pierre
epierre@e-nef.com

Table des matières

Chapitre 1

Introduction

1.1 Présentation

perl est un langage interprété (avec une phase interne de pré-compilation) optimisé pour traiter des fichiers texte, mais qui peut également être utilisé pour diverses tâches d'administration-système.

Sa syntaxe s'inspire très largement de celles de *C, awk, sed* et *sh*, ce qui rend son apprentissage très facile pour toute personne ayant un peu d'expérience avec ces outils.

Il n'y a aucune limitation sur la taille des données ou sur leur contenu (une chaîne peut contenir le caractère nul, et la gestion de la mémoire qu'elle utilise n'est pas à la charge du programmeur).

Une option sur les scripts en *setuid* permet de prévenir beaucoup de trous de sécurité (`PATH` incorrect, ...).

Une grande quantité de modules déjà disponibles permet de développer rapidement des applications touchant à des domaines divers (CGI, Tk, Gtk, Msql, POSIX, Curses, NNTP, ...).

Son utilisation touche divers domaines : traitement de fichiers texte, extraction d'informations, écriture de scripts d'administration-système, prototypage rapide d'applications, etc...

Un autre des ses avantages est qu'il permet d'écrire rapidement des applications puissantes qui peuvent tourner immédiatement sur plusieurs plates-formes différentes. Son utilisation va donc du prototypage rapide d'applications au développement complet de programmes divers.

1.2 Avertissement

Ce document traite uniquement de *perl5*, la dernière version du langage. Quelques incompatibilités existent avec la version précédente, mais la plus grande partie du document reste valable pour *perl4*. Ce dernier n'est de toute façon plus maintenu et son utilisation est déconseillée.

Pour savoir quelle est la version de *perl* que vous utilisez, lancez la commande `perl -v`. Si vous avez une version inférieure à 5.004, demandez à votre administrateur-système d'en installer une plus récente, ne serait-ce que parce que quelques trous de sécurités ont été corrigés (pour plus d'information, reportez-vous au site `http ://www.perl.com/`).

Il faut également noter que ce court document est loin de couvrir l'ensemble si vaste des fonctionnalités offertes par *perl*. Une liste de lectures est disponible plus bas, qui pourront apporter des précisions utiles sur des points laissés dans l'ombre.

1.3 Conseils de lecture

Pour essayer les exemples fournis dans ce document, vous pouvez les taper dans un fichier texte puis lancer l'interpréteur sur ce fichier. Ou bien, ce qui est plus simple, vous pouvez utiliser le débuggueur de *perl* pour taper des commandes interactive-ment.

Pour le lancer, exécutez la commande `perl -de 1`.

Vous pouvez alors taper diverses instructions, qui seront évaluées lors de l'appui sur la touche `RETURN`.

Certains chapitres sont facultatifs dans un premier temps. Ils permettent simplement d'aller un peu plus loin, une fois que l'on sait utiliser les commandes de base de *perl*. Ce sont en particulier les chapîtres 9 (sur les fonctions avancées). et 11 (sur les références).

1.4 Lectures

L'ouvrage de référence pour *perl4* était le livre *Programming perl* [4], connu aussi sous le nom de *Camel book*, coécrit par Larry Wall, l'auteur du langage. Il a été réécrit depuis la sortie de *perl5*, donc n'oubliez pas de vérifier que vous consultez la dernière version. Dans tous les cas, on se reportera au manuel [3] qui a été complètement réécrit pour *perl5* et constitue la référence la plus précise et la plus à jour.

Il existe également un ouvrage d'initiation : *Learning perl* [1], surnommé le *Llama book* à cause de l'animal qui en orne la couverture.

Enfin, pour une référence rapide, on se reportera à l'excellent *Perl Reference Guide* [2] qui se rend très rapidement indispensable. Les fonctions y sont groupées par type, ce qui rend sa consultation très aisée.

1.5 Où trouver de l'aide ?

La première chose à faire lorsque l'on cherche des erreurs dans un script, c'est de lancer l'interpréteur avec l'option `-w` qui affichera différents messages d'avertissement (variables non initialisées, ...) très informatifs. L'option `-c` permet de tester uniquement la validité du code sans l'exécuter. On peut donc les combiner en `-wc`, ce qui testera uniquement la validité syntaxique du code sans l'exécuter.

Pour effectuer des tests encore plus stricts, vous pouvez commencer vos scripts par le pragma (directive de compilation) `use strict`, qui apporte un certain nombre de restrictions visant à vous éviter de commettre des erreurs bêtes (notamment les fautes de frappe dans les noms de variables).

Si vous ne comprenez toujours pas l'erreur, il reste plusieurs solutions : d'abord, consulter la FAQ maintenue par Tom Christiansen[1] et disponible à l'adresse
`http ://language.perl.com/faq/`. À partir de la version 5.004_04 de *perl*, les FAQ sont distribuées avec *perl*. La commande `perldoc perlfaq` vous donnera l'index de ces FAQ. Prenez le réflexe de les consulter quasi-systématiquement, elles constituent une mine de savoir-faire.

Vous pouvez aussi envoyer un mail à
`Olivier.Aubert@enst-bretagne.fr` (j'apprécie également tout commentaire sur ce document, louange ou critique) ou bien poster un message dans le forum[2], continuum *perl*.

En désespoir de cause, vous pouvez toujours poster un message dans le newsgroup `comp.lang.perl.misc`, qui possède une large audience (qui comprend en particulier Larry Wall, le créateur du langage, et de nombreux gourous comme Tom Christiansen, Randal L. Schwartz ou Tim Bunce).

1.6 Où trouver les sources, documents, FAQs ?

Le réseau CPAN, abréviation de *Comprehensive Perl Archive Network*, a été mis en place dans le but de centraliser tous les documents et fichiers relatifs à *perl*. Le site principal est `ftp.funet.fi`, et il en existe deux miroirs en France :
`ftp ://ftp.jussieu.fr/pub/Perl/CPAN/` et
`ftp ://ftp.lip6.fr/pub/perl/CPAN/`.

C'est dans ces sites que vous pourrez trouver les dernières versions des sources, des documents et des modules relatifs à *perl*. Un des modules les plus intéressants est peut-être le module `CPAN`, qui automatise plus ou moins le rapatriement et l'installation des modules. Exécutez la commande `perldoc CPAN` pour en connaître le fonctionnement.

Il faut noter enfin l'existence de deux sites WWW : `http ://www.perl.com/`, site maintenu par Tom Christiansen, qui est une source d'informations très précieuse, ainsi que `http ://www.perl.org/` qui est maintenu par un groupe d'utilisateurs. Pour des informations vraiment techniques, préférez le premier.

[1] Grand gourou de *perl*
[2] Medium accessible uniquement aux étudiants de Télécom Bretagne

Chapitre 2

Structures de données

Les données en *perl* ne sont pas typées[1], mais il existe trois grands *types* de structure de données : les scalaires, les tableaux et les tableaux associatifs.

2.1 Les scalaires

2.1.1 Les données scalaires

Ce sont les chaînes de caractères, les nombres et les références. Voici les conventions utilisées par *perl* dans leur représentation :

Chaînes de caractères Elles sont encadrées par des `"` ou des `'`. La différence entre ces deux notations est similaire à celle utilisée par les shells : dans une chaîne délimitée par deux `"`, les variables seront interpolées. En revanche, dans une chaîne délimitée par deux `'`, aucune interprétation du contenu de la variable ne sera faite. Un exemple de ce comportement est donné un peu plus loin dans le document, lorsque les variables scalaires sont présentées.

Les chaînes de caractères ne sont limitées en taille que par la mémoire disponible, et elles peuvent contenir le caractère nul.

Nombres Plusieurs notations sont utilisées. Quelques exemŋples suffisent : `123`, `123.45`, `123.45e10`, `1_234_567` (les caractères `_` sont ignorés), `0xffff` (valeur hexadécimale), `0755` (valeur octale)[2], . . . Il faut noter que la conversion nombre ↔ chaîne se fait de manière automatique : `12345` représente la même chose que `"12345"`. La différence se fera lors de l'application d'une fonction : (`log()` implique un nombre, `substr()` implique une chaîne de caractères).

Références C'est une des nouveautés de *perl5*. Le chapitre 11 leur est consacré.

Une valeur scalaire est interprétée comme **FALSE** dans un contexte booléen si c'est une chaîne vide ou le nombre `0` (ou son équivalent en chaîne `"0"`).

Valeurs *defined* ou *undefined*

Il y a en fait deux types de scalaires nuls : *defined* et *undefined*. La fonction `defined()` peut être utilisée pour déterminer ce type. Elle renvoie 1 si la valeur est définie, 0 sinon. Une valeur est retournée comme *undefined* quand elle n'a pas d'existence réelle (erreur, fin de fichier, utilisation d'une variable non initialisée).

2.1.2 Les variables scalaires

Elles sont toujours précédées d'un signe `$`. Les noms de variables peuvent contenir des caractères numériques ou alphanumériques. Les noms qui commencent par un chiffre ne peuvent être composés que

[1] Il semble cependant que cette possibilité soit à l'étude chez les développeurs de *perl*, dans le cadre du compilateur de *perl*

[2] Attention, dans *perl* et plus généralement sous *UNIX*, un nombre commençant par `0` est considéré comme étant en octal

de chiffres. Les noms qui ne commencent ni par un chiffre, ni par une lettre, ni par le caractère _ sont limités à un caractère (la plupart sont des noms de variables pré-définies de *perl*, décrites dans le chapitre 13).

Notons tout de suite l'existence d'une variable scalaire qui a la particularité d'être l'argument par défaut de nombreuses fonctions : `$_`. Nous reviendrons souvent sur cette variable au caractère particulier.

Pour donner un exemple d'utilisation des variables, nous allons revenir sur la notion d'interpolation de chaînes de caractères exposée plus haut :

```
$a = 12;
$b = "Test";
# Ci-dessous : interpolation de la valeur.
$c = "Valeur de a : $a"; # ce qui donne la chaîne "Valeur de a : 12"
# Ci-dessous : pas d'interpolation de la valeur.
$d = 'Valeur de b : $b'; # ce qui donne la chaîne "Valeur de b : $b"
```

2.2 Les tableaux et les listes

2.2.1 Construction des listes

Une liste est un ensemble de valeurs scalaires[3], que l'on peut construire de diverses manières. La plus intuitive est l'énumération des éléments entre parenthèses, en les séparant par des virgules :

```
(1, "chaine", 0x44, $var)
```

La liste vide est représentée par `()`.

Il existe également une autre fonction de construction de tableaux, qui clarifie souvent l'écriture des scripts. C'est la fonction `qw`[4], qui permet de faire une énumération des éléments de la liste en les séparant uniquement par des espaces, d'où un gain de lisibilité. Par exemple :

```
@tableau = qw(Facile de construire une liste à 8 éléments.);
```

2.2.2 Les variables de tableaux

Les listes sont stockées dans des variables de type tableau. Ces tableaux, indexés par des entiers et commençant à l'index 0, permettent d'accéder directement à certaines valeurs de la liste.

Les variables sont précédées d'un signe `@` lorsqu'on manipule le tableau entier : `@tableau`[5]

Un tableau étant constitué uniquement de valeurs scalaires, lorsque l'on veut accéder à une des valeurs du tableau, en précisant son index entre crochets, celle-ci est donc scalaire, et il faut donc écrire `$tableau[2]`.

Il existe ce qu'on appelle des *tranches* de tableaux, qui sont notées ainsi :
`@tableau[2,5]`, `@tableau[2 .. 5]`.

Pour obtenir le nombre d'éléments d'un tableau, on peut utiliser la notation `$#tableau`, qui renvoie l'index de fin du tableau (attention, les index commencent à zéro), ou utiliser la fonction `scalar(@tableau)`, qui renvoie le nombre d'éléments contenus dans le tableau.

Certaines personnes vous diront qu'il suffit d'utiliser la variable de tableau dans un contexte scalaire (par exemple une addition), ce qui est syntaxiquement exact, mais mon expérience personnelle m'a conforté dans l'utilisation de `scalar`, qui a le mérite de rendre la conversion explicite. À vous de choisir... Comme l'exprime de manière appropriée un des slogans de *perl* : *Il y a plus d'une manière de faire*[6].

On peut sans problème effectuer des affectations de liste à liste, ce qui donne par exemple un moyen efficace d'échanger deux variables :

```
($a, $b) = ($b, $a);
@tableau1 = @tableau2;
```

De la même manière que pour les variables scalaires, la variable `@_` est prise comme argument par défaut pour de nombreuses fonctions.

---

[3]Si vous vous demandez déjà comment faire des listes de listes, vous êtes un peu en avance. Attendez d'arriver à la section 11 sur les références.

[4]documentée dans la page de manuel `perlop`

[5]Notez que chaque type de variable possède son propre espace de noms, donc `@var` n'a aucun rapport avec `$var`.

[6]*There's more than one way to do it.*

2.3 Les tableaux associatifs

2.3.1 Construction des tableaux associatifs

Ce sont des hash-tables (d'autres langages utilisent le terme de *dictionnaires*) gérées de manière transparente par *perl*. Ce sont donc des tableaux indexés par des chaînes de caractères.

Ce sont en fait des listes contenant des couples (clé, valeur). Leur construction s'effectue de la même manière que pour une liste :

```
( "cle1" => "valeur1",
  "cle2" => "valeur2" );
```

Il faut noter ici que l'opérateur `=>` n'est en fait qu'un synonyme de `,`. Il est cependant possible qu'il obtienne une signification particulière dans les prochaines versions de *perl*.

2.3.2 Les variables de tableaux associatifs

De manière analogue aux listes, on s'adresse à l'ensemble du tableau associatif en précédant son nom par le caractère `%` :

```
%hash = (
          "cle1" => "valeur1",
          "cle2" => "valeur2"
        );
%hash2 = %hash1;
```

On accède ensuite à chacun de ses éléments en précisant la chaîne d'indexation entre accolades. N'oubliez pas que comme pour les listes, les éléments d'un tableau associatif sont des valeurs scalaires, donc qu'elles doivent être précédées d'un dollar (`$`).

```
print $hash{'cle2'};
$hash{"cle3"} = 12;
```

Chapitre 3

Syntaxe et structures de contrôle

3.1 Syntaxe

Un script perl consiste en une suite de déclarations. Toutes les variables utilisateur non initialisées sont considérées comme nulles (*undefined*, la valeur `0` ou la chaîne `""`, suivant la fonction que vous appliquez).

Les commentaires sont introduits par le caractère `#`, et s'étendent jusqu'à la fin de la ligne. Il existe également un système de documentation plus évoluée, intégré au langage, connu sous le nom de *POD* (*Plain Old Documentation*), qui est traité dans le chapitre 7.

La première ligne du script doit contenir le chemin d'accès à l'interpréteur, soit (expression à modifier suivant la localisation de votre interpréteur) :

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

3.1.1 Commandes simples

Chaque commande doit être terminée par un point-virgule `;`. Elle peut être éventuellement suivie d'un *modifier*[1] juste avant le `;`. Les *modifiers* possibles sont :

```
if EXPR
unless EXPR
while EXPR
until EXPR
```

Par exemple :

```
print "Test reussi\n" if ($var == 1);
```

3.1.2 Blocs de commandes

Une séquence de commandes simples constitue un bloc. Un bloc peut être délimité par le fichier qui le contient, mais généralement il est délimité par des accolades `{}`.

3.2 Structures de contrôle

Les structures de contrôle sont :

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
[LABEL] while (EXPR) BLOCK
[LABEL] for (EXPR; EXPR; EXPR) BLOCK
[LABEL] foreach VAR (ARRAY) BLOCK [LABEL] BLOCK continue BLOCK
```

[1] Quelqu'un peut-il me proposer une traduction adéquate ?

Attention : contrairement au C, les accolades sont obligatoires même si les blocs ne sont composés que d'une seule commande. Ceci supprime par exemple les ambiguïtés résultant de l'imbrication de plusieurs `if`. Il est cependant possible d'utiliser la syntaxe vue plus haut ; les exemples suivants sont équivalents :

```
if (!open(FICHIER, $fichier))
   { die "Impossible d'ouvrir fichier: $!"; }

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

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

La dernière syntaxe est la plus utilisée dans ce cas précis, elle fait partie des idiomes de *perl*. N'oubliez jamais de tester la valeur de retour d'une fonction susceptible de ne pas se terminer correctement, et de toujours afficher un message d'erreur le plus précis possible (ici, l'utilisation de la variable spéciale `$!` permet d'afficher le message standard d'erreur UNIX indiquant la cause du problème). Ce conseil ne s'applique évidemment pas qu'à *perl*.

3.2.1 La commande `while`

La commande `while` exécute le bloc tant que l'expression est vraie. Le `LABEL` est optionnel. S'il est présent, il consiste en un identificateur, généralement en majuscules, suivi de :. Il identifie la boucle pour les commandes `next` (équivalent C : `continue`), `last` (équivalent C : `break`) et `redo`. Il permet ainsi de sortir simplement de plusieurs boucles imbriquées.

```
BOUCLE:
while ($var == 0)
{
  while ($var2 == 0)
  {
    ...;
    last BOUCLE if ($var3 == 1);
  }
}
```

3.2.2 La commande `for`

La commande `for` a la même syntaxe que son homonyme en C :

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

3.2.3 La commande `foreach`

La commande `foreach`[2] va affecter successivement à la variable VAR les éléments du tableau ARRAY. La variable VAR est implicitement locale à la boucle et retrouve sa valeur initiale – si elle existait – à la sortie de la boucle.

Si ARRAY est un tableau réel (et non une liste retournée par exemple par une fonction), il est possible de modifier chaque élément de ce tableau en affectant la variable VAR à l'intérieur de la boucle.

```
foreach $arg (@ARGV)
{
  print "Argument : ", $arg, "\n";
  # L'action suivante va mettre à zéro les éléments de @ARGV
  # C'est juste pour l'exemple, ne cherchez pas d'utilité à cette action.
  $arg = "";
}
```

[2] qui peut être remplacée par `for`

3.2.4 Comment émuler `switch` ?

Il n'y a pas de commande `switch` prédéfinie, car il existe plusieurs moyens d'en écrire l'équivalent :

```
SWITCH: {
  if (EXPR1) { ...; last SWITCH; }
  if (EXPR2) { ...; last SWITCH; }
  if (EXPR3) { ...; last SWITCH; }
  ...;
}
```

ou encore une suite de `if`/`elsif`.

3.3 Les opérateurs de test

Ils reprennent en grande partie la syntaxe du C, également en ce qui concerne les règles de précédence. On se référera à la section `perlop` du manuel de *perl5* [3] pour obtenir la liste complète des opérateurs ainsi que leur précédence.

3.3.1 Les opérateurs classiques

Les opérateurs de *perl* ne sont pas difficiles à apprendre pour quiconque connait un peu de C, *perl* s'en inspirant largement.

On peut citer quelques opérateurs :

```
!, ~, <, >, ||, or, &&, and
```

La différence entre `or` et `||`, de même qu'entre `and` et `&&` se situe au niveau de la priorité : `and` et `or` sont les éléments de plus faible priorité du langage.

La distinction la plus grande à faire par rapport au C est l'existence d'opérateurs particuliers s'appliquant aux chaînes de caractères. Le tableau 3.1 donne les équivalences entre les opérateurs s'appliquant aux nombres et ceux s'appliquant aux chaînes.

Nombres	Chaînes	Signification
`<`	`lt`	Inférieur à
`<=`	`le`	Inférieur ou égal à
`>`	`gt`	Supérieur à
`>=`	`ge`	Supérieur ou égal à
`==`	`eq`	Égal à
`!=`	`ne`	Différent de
`<=>`	`cmp`	Comparaison

TAB. 3.1 – Équivalence des tests nombres-chaînes

L'opérateur de comparaison (`<=>` pour les nombre et `cmp` pour les chaînes) renvoie -1, 0 ou 1 selon que le premier argument est inférieur, égal ou supérieur au second.

Une chose à noter sur les opérateurs `||` et `or` : ils n'évaluent que ce qui est nécessaire (c'est-à-dire qu'ils s'arrêtent à la première valeur évaluée à `TRUE`), et renvoient la dernière valeur évaluée. On peut donc écrire :

```
$fichier = $ARGV[0] || "defaut.txt";
```

De même, les opérateurs `&&` et `and` s'arrêtent à la première valeur évaluée à `FALSE` :

```
windows_is_running() && die "Please, execute me on a *REAL* OS...";
```

3.3.2 Les opérateurs de tests sur les fichiers

Reprenant une des caractéristiques des shells, *perl* permet d'effectuer simplement différents tests sur les fichiers, sans qu'il soit nécessaire d'invoquer directement la fonction `stat`.

Chacun de ces opérateurs s'applique soit à un nom de fichier (une chaîne de caractères), soit à un descripteur de fichier (*filehandle*).

Une liste complète de ces opérateurs de tests est disponible dans la section `perlfunc` du manuel [3]. Le tableau 3.2 donne les principaux.

`-r`	fichier accessible en lecture par la personne qui exécute le script
`-w`	fichier accessible en écriture par la personne qui exécute le script
`-x`	fichier exécutable
`-o`	fichier possédé par la personne qui exécute le script
`-e`	fichier existant
`-z`	fichier de taille nulle
...	
`-M`	âge du fichier en jours à partir de la date d'exécution du script
`-s`	taille du fichier

TAB. 3.2 – Opérateurs de tests sur les fichiers

La plupart de ces opérateurs renvoient un résultat booléen. Les deux derniers renvoient cependant des informations plus précises (l'âge ou la taille du fichier).

```
$fichier = "/vmunix";
$age = -M $fichier;
```

3.4 Les fonctions

3.4.1 Déclaration

Les fonctions sont déclarées par le mot-clé `sub`, suivi par le nom de la fonction, puis le bloc d'instructions correspondant, entre accolades

Le script étant pré-compilé avant d'être interprété, la définition de la fonction peut être effectuée indifféremment avant ou après son appel. Il peut être nécessaire de les déclarer avant (ou de faire une *forward declaration* à la C) si on utilise le pragma `use strict` ;

Les arguments sont passés dans le tableau `@_`. On peut définir des variables locales à la fonction grâce aux mots-clés `local()` ou `my()`.

Le mot-clé `return` existe, mais est optionnel. S'il n'est pas présent, la fonction retourne la dernière valeur évaluée.

Voici un exemple :

```
sub factorielle
{
  my $n = shift(@_);
  # ou bien encore
  # my $n = shift;
  # puisque @_ est dans ce cas pris par defaut.

  $n == 1 ? 1 : ( $n * &factorielle($n - 1) );
# équivalent à (notez le return implicite)
# if ($n == 1)
# {
#   1;
# }
# else
# {
#   $n * &factorielle($n - 1);
```

```
# }
}
```

my doit être utilisé de préférence à local, car my donne à la variable une étendue lexicale (*lexical scoping*), i.e. la variable ne sera pas visible des routines appelées ensuite. Ce n'est pas clair ? Un exemple aidera sûrement :

```
# On declare quelques fonctions...
sub f_local
{
  local($foo) = "Foo";
  &print_foo();
}
sub f_my
{
  my($foo) = "Bar";
  &print_foo();
}
# Affichage de la valeur de la variable $foo
sub print_foo
{
  print $foo, "\n";
}
# Début du script.
print "Appel avec local sans initialisation globale : ";
&f_local;
print "Appel avec my sans initialisation globale : ";
&f_my;
# Initialisation de la variable de manière globale
$foo = "Toto";
print "Appel avec local avec initialisation globale : ";
&f_local;
print "Appel avec my avec initialisation globale : ";
&f_my;

# Ce qui donne comme résultat a l'exécution :
Appel avec local sans initialisation globale : Foo
Appel avec my sans initialisation globale :
Appel avec local avec initialisation globale : Foo
Appel avec my avec initialisation globale : Toto
```

Ça va mieux ? Si la réponse est non, prenez une aspirine, faites-moi confiance et facilitez-vous la vie en utilisant systématiquement my.

3.4.2 Appel

Les noms de fonctions sont précédés du signe &. Ce signe est optionnel lorsqu'on fait appel à une fonction. Il est par contre nécessaire lorsqu'on veut passer une fonction comme paramètre par exemple.

Les parenthèses lors de l'appel ne sont pas obligatoires, même s'il n'y a aucun argument. On peut donc appeler une fonction de plusieurs manières :

```
&fonction(2, 4);
fonction(2, 4);
# Attention à la suite (man perlsub pour plus de détails):
fonction2();  # fonction2 est appellée avec une liste vide en argument.
&fonction2(); # Idem.
&fonction2;   # fonction2 est appellée avec la même liste d'arguments
              # que la fonction d'où l'appel est fait. Attention...
```

3.4.3 Deux fonctions particulières

Il existe deux fonctions particulières, héritées de la syntaxe de *awk*, qui permettent d'avoir une plus grande maîtrise sur le déroulement du script. Ce sont les fonctions `BEGIN` et `END`.

Une fonction `BEGIN` est exécutée aussitôt que possible, i.e. au moment où elle est complètement définie, avant même que le reste du script ne soit analysé. S'il y a plusieurs définitions de `BEGIN`, elles seront exécutées dans l'ordre de leur déclaration.

Cette fonction est utilisée en particulier pour modifier le chemin de recherche des fichiers à inclure (voir le chapitre 13 sur les variables spéciales) :

```
BEGIN
{
  push(@INC, "/home/aubert/lib/perl");
}
# que l'on écrit plutôt à partir de la version 5.001m
# use lib '/home/aubert/lib/perl';
```

La fonction `END` est exécutée le plus tard possible, généralement juste avant la sortie de l'interpréteur. Elle permet donc de faire un peu de ménage à la fin d'un programme.

3.5 Les paquetages

perl offre un moyen de protéger les variables d'un éventuel conflit de nom grâce au mécanisme des paquetages (ou encore *espaces de nommage*).

Un paquetage est déclaré par le mot-clé `package`, suivi du nom du paquetage, et s'étend jusqu'à la fin du bloc (ou du fichier, les paquetages étant généralement définis chacun dans leur propre fichier) ou à la prochaine déclaration de paquetage..

On accède ensuite depuis l'extérieur aux variables et aux fonctions du paquetage en les précédant du nom du paquetage suivi de `::`. Il est possible de subdiviser les paquetages en sous-paquetages, *ad nauseam*.

Le paquetage principal est appelé `main`.

Voici un exemple :

```
package Arb;
$a = 1;

package main;
$a = 2;
print $a, "\n";
# renverra 2
print $Arb::a, "\n";
# renverra 1
```

3.6 Les modules

3.6.1 Principe

Les modules sont une extension du concept de paquetage : ce sont des paquetages définis dans un fichier de même nom que le module, et qui sont destinés à être réutilisés.

On inclut un module grâce à la ligne suivante :

```
use Module;
```

ce qui va en fait être interprété comme

```
BEGIN {
require "Module.pm";
import Module;
}
```

`use` effectue un `import` en plus du `require`, ce qui a pour effet d'importer les définitions des fonctions dans l'espace du paquetage courant. Voici l'explication :

```
require Cwd;                  # make Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;                      # import names from Cwd::
$here = getcwd();

require Cwd;                  # make Cwd:: accessible
$here = getcwd();             # oops! no main::getcwd()
```

3.6.2 Modules existants

Une liste complète est régulièrement postée dans les news et archivée sur les sites *CPAN* (voir l'introduction). Un certain nombre est livré dans la distribution standard de *perl*, le reste se trouve également sur les sites *CPAN*.

Les plus importants sont ceux qui permettent de gérer différents formats de bases de données (`NDBM_File`, `GDBM_File`, ...), `CGI` qui fournit une interface très agréable à utiliser lors de l'écriture de scripts CGI,`GetCwd` qui permet de ne pas avoir à faire un appel à `/bin/pwd` pour obtenir le répertoire courant, `Fcntl` qui permet d'accéder aux constantes définies dans `fcntl.h`, `FileHandle` qui fournit des méthodes pour accéder aux *filehandles*, `Find` qui traverse une arborescence, `GetOptions`, POSIX qui permet d'accéder aux fonctions POSIX, `Tk` qui permet de construire des applications graphiques,...

Chapitre 4

Les entrées-sorties

4.1 Les bases

On va commencer avec le classique programme *hello world !*.

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

print "hello world !\n";

print "hello ", "world !\n";

print("hello ", "world !\n");

print "hello " . "world !\n";
```

On voit ici quatre lignes qui produisent exactement le même résultat. La première ligne est immédiatement compréhensible. La seconde illustre le fait que la fonction print prend une liste en argument. Les parenthèses autour de la liste sont optionnelles lorsqu'il n'y a pas d'ambiguïté. On peut cependant sans problème l'écrire avec des parenthèses, comme dans la troisième ligne.

La dernière ligne utilise un nouvel opérateur, ., qui effectue la concaténation de deux chaînes de caractères. Le résultat de cette concaténation est ensuite affiché par print.

4.2 Interaction

Voila un programme d'exemple qui demande d'entrer un nom et met le résultat de cette demande dans une variable :

```
print "Entrez votre nom : ";

$nom = <STDIN>;
chomp($nom);

print "Hello $nom\n";
```

La partie intéressante se situe au niveau des lignes 2 et 3. La ligne 2 affecte à la variable $nom le résultat de l'opérateur <> appliqué au descripteur de fichier STDIN.

L'opérateur de fichier <> est utilisé pour lire une ou plusieurs lignes d'un fichier. Dans un contexte scalaire comme ici, il lira les données jusqu'au prochain retour-chariot, qui sera d'ailleurs inclus dans le résultat.

Dans un contexte de tableau par contre, par exemple @lignes = <STDIN>, l'opérateur renvoie l'ensemble des lignes du fichier dans un tableau, ce qui peut produire des résultats indésirables, si par exemple on applique cette ligne à un fichier de plusieurs mégaoctets. Mais cette forme reste très utile pour lire rapidement en mémoire le contenu entier d'un fichier.

La fonction chomp[1] que l'on applique ensuite à la variable supprime le dernier caractère si c'est un *newline*. On peut d'ailleurs l'appliquer également à un tableau. Dans ce cas, elle enlèvera les *newline* à chaque élément du tableau.

On peut condenser ces deux lignes sous la forme

```
chomp($nom = <STDIN>);
```

4.3 L'ouverture

La commande open permet d'ouvrir un fichier. Sa syntaxe est la suivante :

```
open(FILEHANDLE, EXPR);
```

En cas de succès, cette fonction renvoie une valeur non nulle, ce qui explique que l'on rencontre tout le temps la ligne

```
open(FILE, "fichier") or die "Cannot open fichier: $!";
```

La fonction die affiche sur STDERR la chaîne (ou la liste) passéee en argument, puis termine le script en renvoyant un code d'erreur non nul.

Le nom du *filehandle* doit être en majuscules (en fait, il ne s'agit que d'une convention, mais tout le monde la respecte). Quelques filehandles par défaut existent : STDIN, STDOUT, STDERR.

L'expression EXPR est le nom du fichier à ouvrir, précédé éventuellement d'un caractère qui précise le mode d'ouverture. Ces caractères sont résumés dans le tableau 4.1

Caractère	Mode
Aucun	lecture
<	lecture
>	écriture
>>	ajout
+<	lecture/écriture
|	pipe

TAB. 4.1 – Modes d'ouverture

Un nom de fichier particulier est à signaler : "-". Ouvrir - est équivalent à ouvrir STDIN et ouvrir >- revient à ouvrir STDOUT.

L'utilisation du caractère *pipe* | permet d'envoyer du texte sur l'entrée standard d'une commande, ou bien de récupérer sa sortie standard (voir une illustration au chapitre 12). Dans ce cas, la valeur retournée par open est le numéro de processus (*pid*) du processus lancé.

4.4 La lecture

Pour lire sur un descripteur de fichier précédemment ouvert, on utilise principalement l'opérateur <>, comme par exemple dans $nom = <STDIN> ;, qui lit le filehandle précisé jusqu'au retour chariot suivant (qui est inclus dans le résultat).

Il existe également une commande read(FILEHANDLE,SCALAR,LENGTH) qui lit LENGTH octets de données dans la variable SCALAR depuis le fichier FILEHANDLE.

```
$len = read(FILE, $buffer, 512);
```

Cette commande est implémentée avec la fonction fread(). Pour effectuer un véritable appel-système à la fonction read(), il faut utiliser la commande sysread.

Pour lire un seul caractère, on peut utiliser la fonction getc(FILEHANDLE).

[1]En *perl4*, il n'existe que chop, qui supprime le dernier caractère sans tester si c'est un retour à la ligne

4.5 L'écriture

Le plus courant est l'utilisation de `print` auquel on fournit le descripteur de fichier en paramètre. Par exemple :

```
print FILE "hello world !\n";
```

Il faut noter qu'il n'y a pas de virgule entre le nom du descripteur de fichier et les éléments à écrire.

Il est possible d'accéder aux mêmes possibilités de formatage qu'en C en utilisant la fonction `printf`, à laquelle on passe les mêmes paramètres que son homologue en C :

```
printf STDOUT "Le nombre de %s est %3d.\n", $element, $nombre;
```

La fonction `printf` est cependant plus lente que la fonction `print`, que l'on utilisera donc de préférence.

De même que pour la lecture, il est possible d'effectuer un véritable appel-système à la fonction `write()` en utilisant `syswrite(FILEHANDLE,SCALAR,LENGTH)`.

```
$len = syswrite(FILE, $buffer, length($buffer));
```

4.6 La fermeture

Elle est effectuée par la fonction `close`, à laquelle on fournit en paramètre le descripteur de fichier à fermer :

```
close(FILE);
```

4.7 Le buffering

Les opérations d'entrée-sortie sont bufferisées par défaut. Il est possible de forcer *perl* à faire un *flush* après chaque opération de lecture ou d'écriture en fixant la variable spéciale `$|` à une valeur non nulle, après avoir sélectionné le descripteur courant comme descripteur par défaut grâce à la commande `select`.

On utilise communément cette syntaxe :

```
$oldfh = select(FILE);
$| = 1;
select($oldfh);
```

Note : la fonction `select` existe également sous la même forme qu'en C, pour effectuer l'appel-système `select`, ce qui peut induire une certaine confusion. Pour plus d'informations sur l'appel-système `select`, reportez-vous à la page de manuel de votre système :

```
$rin = $win = $ein = '';
vec($rin, fileno(STDIN), 1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;

($nfound, $timeleft) = select($rout = $rin,
                              $wout = $win,
                              $eout = $ein,
                              $timeout);
```

Si on désire plus de lisibilité pour fixer le buffering, on peut utiliser le module `FileHandle` :

```
use FileHandle;

autoflush FILE 1;
```

Chapitre 5

Les expressions régulières

Les expressions régulières[1] sont une des caractéristiques de *perl* qui rendent ce langage particulièrement adapté au traitement des fichiers texte.

Une expression régulière[2] est une suite de caractères suivant une certaine syntaxe qui permet de décrire le contenu d'une chaîne de caractères, afin de tester si cette dernière correspond à un motif[3], d'en extraire des informations ou bien d'y effectuer des substitutions.

5.1 La syntaxe

Elle est à la base identique à celle des expressions régulières de programmes connus comme *grep, sed*,... mais plusieurs nouvelles fonctionnalités ont été ajoutées. Nous allons voir ici les bases, ainsi que certaines améliorations apportées par *perl*.

Les opérations sur les expressions régulières sont effectuées par défaut sur la variable `$_`.

Pour les faire s'appliquer à une autre variable, il faut utiliser l'opérateur `=~` :

```
if ($variable =~ /regexp/) {...};
# est donc équivalent à
$_ = $variable;
if (/regexp/) {...};
```

5.1.1 Les métacaractères

Chaque caractère correspond à lui-même, exception faite des *métacaractères* qui ont une signification particulière. Pour traiter un métacaractère comme un caractère normal, il suffit de le précéder d'un \.

Voici quelques métacaractères, avec leur signification :

`^`	début de chaîne. Ce n'est pas un véritable caractère ;
`$`	fin de chaîne. Même remarque ;
`.`	n'importe quel caractère, excepté *newline* ;
`|`	alternative (à placer entre parenthèses) ;
`()`	groupement et mémorisation ;
`[]`	classe de caractères.

Quelques explications sont peut-être nécessaires. Une regexp du type `/a[bc]d/` correspondra aux chaînes `abd` et `acd`. `[]` permet d'énumérer une classe de caractères. L'interprétation de cette expression sera donc : un `a`, suivi d'un `b` ou d'un `c`, puis un `d`.

L'ensemble des caractères composant la classe peut être précisée par énumération (comme précédement) ou bien en précisant un intervalle comme par exemple `/[a-z]/` qui correspondra à tous les caractères compris entre `a` et `z`.

[1] Sujet à polémique : c'est ici une traduction littérale de l'anglais *regular expression*, qui serait plus correctement traduit en *expression rationnelle*. Mais l'usage semble avoir consacré la traduction littérale.

[2] On dit le plus souvent *regexp*

[3] Le mot utilisé en anglais est *matching*

On peut également prendre le complémentaire de cet ensemble, en le précédant d'un `^`[4]. Donc `/a[^bc]d/` correspondra à toutes les chaînes du type `a.d`, sauf `abd` et `acd`. On peut donc lire : un `a`, suivi d'un caractère qui n'est ni un `b` ni un `c`, puis un `d`.

L'alternative permet de préciser que l'on recherche l'une ou l'autre des expressions séparées par des `|`. Par exemple, `/arti(chaut|ste)/` correspondra aux chaînes `artichaut` et `artiste`. Il est bien sûr possible de mettre plus de deux alternatives.

Enfin, la mémorisation permet de mémoriser certaines des parties de la chaîne. Par exemple, appliquer l'expression `/b(.+)ars/` à la chaîne `beggars` mémorisera la partie qui correspond à ce qui se trouve entre parenthèses, i.e. `egg`, et fixera la variable `$1` à cette valeur. On peut donc lire : un `b`, suivi d'une série (`+`) de caractères quelconques (`.`), que l'on mémorisera (les parenthèses), puis la chaîne `ars`.

5.1.2 D'autres caractères spéciaux

Les notations suivantes sont également utilisables (pour la plupart inspirées de la syntaxe de la fonction C `printf`) :

`\t`	tabulation
`\n`	newline
`\r`	retour-chariot
`\e`	escape
`\cC`	contrôle-`C`, où `C` peu être n'importe caractère.
`\s`	espace
`\S`	non-espace
`\w`	lettre (caractères alphanumériques et _)
`\W`	non-lettre
`\d`	chiffre
`\D`	non-chiffre

5.1.3 Les quantificateurs

Différents quantificateurs s'appliquent aux caractères et métacaractères :

`*`	apparaît 0, 1 ou plusieurs fois
`+`	apparaît 1 ou plusieurs fois
`?`	apparaît 0 ou 1 fois
`{n}`	apparaît exactement n fois
`{n,}`	apparaît n fois ou plus
`{m,n}`	apparaît au moins m fois, au plus n fois

Il faut savoir que ces quantificateurs sont dits gourmands. Par exemple, appliquer `/a(.+)a/` à `abracadabra` fixera `$1` à `bracadabr`, la plus longue chaîne possible correspondant à l'expression, et non `br`.

5.1.4 Les quantificateurs non-gourmands

Une nouveauté intéressante introduite dans la version 5 de *perl* est la possibilité d'obtenir des quantificateurs non gourmands, i.e. qui ne matchent pas la plus grande chaîne possible, en mettant un `?` après le quantificateur.

Voici un exemple illustrant le caractère gourmand des expressions régulières classiques, et l'apport de *perl5* dans ce domaine :

```
$chaine = 'Voila un <A HREF="index.html">index</A> et
une autre <A HREF="reference.html">reference</A>.';

($greedy) = ($chaine =~ /(<.+>)/);
($nongreedy) = ($chaine =~ /(<.+?>)/);

print "1: ", $greedy, "\n2: ", $nongreedy, "\n";
```

[4] à ne pas confondre avec le métacaractère `^` que l'on trouve en dehors des classes et qui correspond à un début de ligne

qui donne le résultat suivant :

```
1: <A HREF="index.html">index</A> et une
          autre <A HREF="reference.html">reference</A>
2: <A HREF="index.html">
```

5.2 Utilisation : recherche

Une des premières utilisations des expressions régulières est le *matching* : on peut tester si une chaîne de caractères correspond à une expression régulière, i.e. à un motif particulier.

5.2.1 Syntaxe

Pour cela, on utilise la fonction `m/REGEXP/`, qui s'applique par défaut à la variable `$_`. Si on désire l'appliquer à une autre variable, la syntaxe est la suivante :

```
if ($var =~ m/REGEXP/) { ... }
```

`REGEXP` est l'expression régulière dont on cherche à savoir si elle correspond à la variable `$var`.

On peut faire suivre cette fonction de paramètres qui modifient le comportement de la fonction de matching. Ces paramètres sont formés d'au moins un caractère parmi `g, i, s, m, o, x`. Voici le détail de leurs actions :

`g`	recherche globale (recherche toutes les occurences, s'il y en a plusieurs dans la chaîne traitée)
`i`	ne pas tenir compte de la casse des caractères (case-insensitive)
`s`	traiter la chaîne comme une ligne simple (défaut)
`m`	traiter la chaîne comme une ligne multiple
`o`	ne compiler l'expression qu'une seule fois
`x`	utiliser les expressions régulières étendues

Pour tester si la variable `$var` contient la chaîne `foo` mais sans faire de distinction majuscules/minuscules, on écrira :

```
if ($var =~ m/foo/i) { ... }
```

Le premier `m` de l'expression `m/REGEXP/` peut être omis si le délimiteur de l'expression est un `/` (*slash*, ou barre oblique). L'utilisation du `m` permet d'utiliser n'importe quel caractère comme délimiteur, ce qui évite, lorsqu'il s'agit de construire une expression contenant des `/`, de précéder chaque `/` d'un `\`. Par exemple :

```
if ($var =~ m#^/etc#) { ... }
```

Les expressions régulières peuvent contenir des variables qui seront interpolées à chaque appel à l'expression. Ce comportement est utile, et intuitif, dans certaines situations mais est pénalisant du point de vue de la rapidité d'exécution. Si l'on est certain que la variable ne changera pas au cours du script, on peut ajouter le modificateur `o`. Mais attention, si on oublie que l'on a mis ce modificateur et que l'on modifie quand même la variable, ce changement ne sera pas pris en compte dans l'expression régulière.

5.2.2 Valeurs de retour

La valeur retournée par la fonction dépend du contexte dans lequel elle est appelée : dans un contexte scalaire, la fonction renvoie une valeur non nulle en cas de succès, et une valeur *undefined* dans le cas contraire.

```
$match = ($chaine =~ /regexp/);
```

Dans un contexte de liste, la fonction renvoie la liste des éléments qui ont matché les expressions entre parenthèses. Si l'expression ne correspondait pas, on obtient une liste nulle.

```
($href) = ($chaine =~ /<a\s+href="(.+?)">/i);
```

Dans tous les cas, la fonction fixera les variables `1,2, ...` avec les éléments qui ont matché les expressions entre parenthèses.

5.3 Utilisation : substitution

La syntaxe de la fonction de substitution est analogue à celle utilisée par *sed*, *vi*, ..., mises à part les quelques différences syntaxiques d'écriture des regexps. De manière analogue à la fonction de comparaison, la fonction de substitution s'applique par défaut à la variable `$_`. Pour la faire s'appliquer à une variable quelconque, il faut utiliser la notation :

```
$var =~ s/REGEXP/chaîne/egismox;
```

Les modificateurs qui suivent l'expression sont identiques à ceux applicables à la fonction de comparaison.

Un nouveau modificateur est disponible : `e`. Il précise que l'expression de remplacement est une expression à évaluer, ce qui permet d'utiliser le résultat d'une opération dans la chaîne de remplacement.

Par exemple, pour convertir des caractères du type `%xx` où `xx` est un code ASCII en hexadécimal dans le caractère correspondant, il suffit d'écrire :

```
$chaine =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
```

Une petite explication ? On veut effectuer une substitution (`s/.../.../`) sur la variable `$chaine`. On cherche un caractère `%` suivi de deux caractères valides en hexadécimal (`[a-fA-F0-9]`). On mémorise ces deux caractères, dans la variable `$1` puis on remplace les trois caractères (les deux mémorisés plus le `%` qui les précéde) par le résultat de la fonction `pack("C", hex($1))`, qui évalue le contenu de `$1` en hexadécimal, puis affiche le caractère correspondant à cette valeur. On effectue cette opération tant qu'il reste des possibilités (le modificateur `g`).

Prenez votre temps pour essayer de comprendre le fonctionnement de cette expression régulière. Ensuite, essayez de comprendre les transformations qui permettent d'arriver à l'expression équivalente :

```
$chaine =~ s/%([a-f0-9]{2})/pack("C", hex($1))/egi;
```

5.4 Utilisation : translation

Voilà une dernière fonction qui est généralement associée aux expressions régulières, même si elle ne les utilise pas. Son point commun avec les fonctions précédentes est qu'il faut utiliser `=~` pour préciser à quelle variable elle s'applique (`$_` par défaut).

Elle a le même comportement que l'utilitaire *UNIX* `tr`. Sa syntaxe est la suivante :

```
$variable =~ tr/liste1/liste2/cds;
```

Par défaut, elle transpose chaque caractère de `liste1` en le caractère correspondant de `liste2`. Trois modificateurs sont disponibles :

`c`	complémente la liste `liste1`
`d`	supprime les éléments de `liste1` qui n'ont pas de correspondance dans `liste2`
`s`	compresse les caractères dupliqués qui sont en double

Pour supprimer par exemple tous les caractères non alphanumériques, on écrit :

```
$chaine =~ tr/a-zA-Z0-9.//cd;
```

Chapitre 6

Les fonctions classées par catégorie

Ce chapitre est un résumé des fonctions les plus importantes classées par catégorie. Pour avoir une liste complète triée par catégories, on se référera au *Perl Reference Guide* [2] en utilisant notamment la commande `perldoc perlfunc`, ou, pour obtenir directement la syntaxe de la fonction `foo` : `perldoc -f foo`[1]

6.1 Les fonction de manipulation de listes

Pour manipuler les listes, *perl* propose plusieurs fonctions.

6.1.1 grep

Syntaxe : `grep(EXPR, LIST)`

Cette fonction évalue `EXPR` pour chaque élément de `LIST`, en fixant la variable `$_` à la valeur de cet élément. Une modification de `$_` à l'intérieur de `EXPR` modifiera la valeur de l'élément correspondant de `LIST`.

La valeur de retour est une liste contenant les éléments de `LIST` pour lesquels `EXPR` a retourné `TRUE`.

```
@elements = (1, 2, 3, 4, 5);
@selection = grep($_ < 3, @elements);
# -> (1, 2)
```

6.1.2 map

Syntaxe : `map(EXPR, LIST)`

Cette fonction évalue `EXPR` pour chaque élément de `LIST`, en fixant la variable `$_` à la valeur de cet élément. Une modification de `$_` à l'intérieur de `EXPR` modifiera la valeur de l'élément correspondant de `LIST`.

La valeur de retour est une liste contenant les résultats des évaluations de `EXPR` sur les éléments de `LIST`.

```
@elements = (1, 2, 3, 4, 5);
@doubles = map($_ * 2, @elements);
# -> (2, 4, 6, 8, 10)
```

6.1.3 pop, shift

Syntaxe : `pop @ARRAY, shift @ARRAY`

Ces fonctions extraient une valeur de la liste ou du tableau passé en paramètre. Dans le cas du tableau, elle le raccourcissent d'un élément et renvoient la valeur extraite.

Pour `pop`, c'est la dernière valeur de la liste qui est extraite, pour `shift`, la première valeur.

[1] Uniquement dans les versions récentes de *perl*

```
$valeur = pop(@elements);
# -> $valeur = 5
# et @elements = (1, 2, 3, 4);
```

6.1.4 push, unshift

Syntaxe : `push(@ARRAY, LIST)`, `unshift(@ARRAY,LIST)`

Ces fonctions effectuent l'opération inverse des précédentes : `push` va ajouter les éléments de `LIST` à `ARRAY`. `unshift` va insérer les éléments de `LIST` au début de `ARRAY`.

```
push(@elements, $valeur);
# -> @elements = (1, 2, 3, 4, 5)
```

6.1.5 reverse

Syntaxe : `reverse LIST`

Dans un contexte de liste, cette fonction renvoie `LIST` dans l'ordre inverse sans modifier `LIST`.

Dans un contexte scalaire, elle renvoie le premier élément de `LIST` en ayant inversé ses caractères.

```
@elements = reverse(@elements);
# -> @elements = (5, 4, 3, 2, 1)
print "" . reverse(123);
# -> 321
```

6.1.6 sort

Syntaxe : `sort [SUBROUTINE] LIST`

Trie `LIST` et retourne la liste triée. `SUBROUTINE` peut être spécifiée pour changer la fonction de comparaison. C'est soit un nom de fonction utilisateur, soit un bloc, qui retourne une valeur négative, nulle ou positive, et qui s'applique aux variables `$a` et `$b` (pour des raisons d'optimisation).

Si aucune routine n'est spécifiée, le tri sera alphanumérique.

```
print sort { $a <=> $b } @elements;
# -> (1, 2, 3, 4, 5)
```

6.1.7 split

Syntaxe : `split PATTERN, EXPR [, LIMIT]`

`split` va diviser la chaîne de caractères `EXPR` suivant le séparateur `PATTERN`, qui est une expression régulière.

Le paramètre optionnel `LIMIT` permet de fixer la taille maximale de la liste retournée.

Par exemple, la commande suivante va retourner les champs délimités par `:` :

```
@elements = split(/:/, $chaine);
```

6.2 Les fonctions sur les tableaux associatifs

6.2.1 keys

Syntaxe : `keys %HASH`

`keys` retourne une liste contenant les clés du tableau associatif `%HASH`.

6.2.2 values

Syntaxe : `values %HASH`

`values` retourne une liste contenant les valeurs du tableau associatif `%HASH`.

6.2.3 `each`

Syntaxe : `each %HASH`

`each` retourne une liste à deux éléments, contenant la clé et la valeur pour l'élément suivant de `%HASH`. Quand le tableau associatif a été entièrement parcouru, un tableau nul est retourné (ou la valeur `undef` dans un contexte scalaire).

Cette fonction est surtout utilisée dans le cas de très gros tableaux associatifs, où la place utilisée par la liste des clés serait trop importante.

6.2.4 `delete`

Syntaxe : `delete $HASH{KEY}`

Efface la valeur spécifié du tableau `%HASH` spécifié. Retourne la valeur supprimée.

6.3 Les fonctions de manipulation de chaînes de caractères

On retrouve les mêmes qu'en C, et qui respectent la même syntaxe :

`substr` Retourne la sous-chaîne délimitée par les index fournis en paramètre. *Note :* elle peut être utilisé comme *lvalue* (ie affectée).

```
$chaine = "Hello";
print substr($chaine, 2, 2);
# -> "ll"
substr($chaine, 2, 2) = "toto";
# $chaine -> "Hetotoo"
```

`index`, `rindex` Retourne la position de la première (resp. dernière) occurrence d'une sous-chaîne dans une chaîne de caractères.

```
print index($chaine, "l");
# -> 2
```

D'autres fonctions sont également définies :

`length`
Donne la longueur de la chaîne passée en paramètre.

l'opérateur `.` (point)
Il effectue la concaténation de deux chaînes.

```
$var = $var . ".bak";
```

`crypt(PLAINTEXT, SALT)` Encrypte la chaîne passée en argument.

```
$clair = <STDIN>;
# On récupère le deuxième champ (passwd) de la ligne correspondant à
# l'uid ($<) de la personne exécutant le script
$crypte = (getpwuid($<))[1];
$passwd = crypt($clair, $crypte);
```

`lc`, `lcfirst`, `uc`, `ucfirst` Opérations sur la casse de la chaîne. `lc` passe la chaîne en minuscules (*lowercase*). `lcfirst` passe la première lettre de la chaîne en minuscule. `uc` et `ucfirst` effectuent le passage en majuscules.

```
print lc($chaine);
# -> "hello"
```

6.4 Les autres fonctions

On retrouve généralement les mêmes qu'en C, en particulier pour toutes les fonctions plus avancées, décrites dans le chapitre 9.

Chapitre 7

Le format de documentation *POD*

7.1 Présentation

Le format *POD*[1] a été conçu par Larry Wall afin de faciliter la documentation des modules et programmes *perl*. Un de ses énormes avantages est qu'il peut être inséré à l'intérieur même du code *perl*, ce qui fournit une documentation automatique - et souvent plus à jour - des modules en particulier. Le but de ce langage est de fournir aux programmeurs un moyen simple d'écrire de la documentation.

Un ensemble de convertisseurs est disponible pour convertir ce format *POD* en d'autres formats : *nroff* avec `pod2man` pour l'affichage dans les terminaux, texte avec `pod2txt` pour la lecture simple, LaTeX avec `pod2tex` pour obtenir une impression de qualité, HTML avec `pod2html`, etc. Le document au format *POD* lui-même ne nécessite pas d'éditeur particulier. Comme pour les programmes *perl*, un simple éditeur de texte suffit.

Le programme `perldoc` cherche dans les modules de *perl* celui qui correspond à l'argument qui a été passé, et en extrait la documentation.

La documentation du format est disponible, au format *POD* bien sûr, en exécutant la commande `perldoc perlpod`.

7.2 Syntaxe

La syntaxe de *POD* est simple : elle consiste en trois types de paragraphes. Tous les paragraphes doivent être séparés par une ligne vide.

7.2.1 Exemples de code

Le premier est un paragraphe indenté par au moins un espace ou une tabulation. Le texte de ce paragraphe sera alors reproduit tel-quel (`verbatim`).

7.2.2 Commandes

Le second type est une commande. Une commande commence par un signe *égal* (=), suivi d'un identificateur, puis un texte pouvant être utilisé par l'identificateur. Le tableau 7.1 récapitule les différents identificateurs reconnus.

Lors de la construction d'une liste, il est demandé de rester cohérent dans l'utilisation du paramètre de l'identificateur `=item`. On utilise couramment `=item *` pour construire une liste avec de simples boutons, `=item 1`, `=item 2`, etc pour construire une liste numérotée, et enfin `=item titi` pour construire une énumération d'éléments.

Le premier élément de la liste est utilisé par la plupart des convertisseurs pour choisir le type de liste. Par exemple, une liste commençant par `=item *` sera convertie en HTML en liste non-ordonnée (`<ul> <li> ... </ul>`), et en LaTeX également (environnement *list*), tandis qu'une liste commençant par `=item toto` sera convertie en HTML en liste de définitions (`<dl> <dt> ... <dd> ... </dl>`) et en LaTeX en énumération (environnement *enumerate*).

[1] *Plain Old Documentation*

`=pod`	Début de code *POD*.
`=cut`	Fin de code *POD*.
`=head1 Texte`	Construit un titre de niveau 1.
`=head2 Texte`	Construit un titre de niveau 2.
`=over N`	Début de liste.
`=item Text`	Nouvel élément de liste.
`=back`	Fin de liste.
`=begin X`	Début de section spécifique au format X (`html` par exemple).
`=end X`	Fin de section spécifique au format X.

TAB. 7.1 – Identificateurs du format *POD*

Le paramètre de l'identificateur `=over N` est la taille de l'indentation utilisée. La valeur par défaut est de 4.

7.2.3 Formatage de texte

Le dernier type de paragraphe est un bloc de texte ordinaire, qui peut de plus contenir des commandes de mise en forme, résumées dans le tableau 7.2.

I<texte>	Italique. Utilisé pour les variables et les mises en valeur
B<texte>	Gras (*Bold*) utilisé pour les options de ligne de commande et les programmes.
S<texte>	Le texte contient des espaces insécables.
C<code>	Du code littéral.
L<nom>	Un lien (référence croisée) vers une page de manuel.
L<nom/ident>	Idem, vers un élément de la page de manuel.
L<nom/"sec">	Idem, vers une section de la page de manuel.
L</"section">	Lien vers une section de cette page de manuel.
F<fichier>	Nom de fichier.
X<index>	Entrée d'index.
Z<>	Caractère de taille nulle.
E<entité>	Un caractère accentué suivant le tableau des entités HTML (par exemple E<Agrave> pour À).
E<lt>	Le caractère <.
E<gt>	Le caractère >.

TAB. 7.2 – Liste des séquences de mise en forme de *POD*

7.3 Intégration dans du code

Il suffit à l'intérieur d'un programme *perl* de commencer la documentation par la commande **`=head1 texte`**, et de la terminer par la commande `=cut`. L'interpréteur ignorera le texte compris entre ces deux éléments.

7.4 Exemple

Un exemple est beaucoup plus parlant. En voilà quelques lignes, tirées d'un programme existant :

```
=head1 NAME

clean - clean your account...
```

```
=head1 SYNOPSIS

B<clean> [options] [directory]

with options in:

=over 4

[B<-a>] [B<--all>]
[B<-c>] [B<--check-only>]
[B<-d>] [B<--default>]
[B<-e>] [B<--delete-empty-dir>]
[B<-f>I<[filename]>] [B<--output>I<[=filename]>]
[...]
[B<-V>] [B<--version>]

=back

=head1 OPTIONS

[...]
```

On voit ici apparaître diverses sections standard dans les pages de manuel UNIX : `NAME`, `SYNOPSIS`, `OPTIONS`, `DESCRIPTION`, `NOTES`, `ENVIRONMENT`, `FILES`, `AUTHOR`, `BUGS`.

De nombreux exemples sont disponibles également dans toute distribution de *perl* : la documentation est au format *POD*. Elle se trouve généralement dans le répertoire `/usr/local/lib/perl5/pod/`, ou `/usr/lib/perl5/pod/`.

Chapitre 8

Les formats

perl a été à l'origine conçu pour manipuler du texte et produire des rapports. Il offre donc un outil permettant de formatter ces rapports facilement : les formats.

Les formats sont déclarés à n'importe quel endroit du fichier. Étant donné que *perl* fait une précompilation du script avant de l'exécuter, il n'est pas indispensable de les déclarer avant de les utiliser.

8.1 Syntaxe

On les déclare de la manière suivante (en les terminant par un point seul sur une ligne) :

```
format NAME =
FORMALIST
.
```

`NAME` est le nom associé au format. Par défaut, si un *filehandle* est défini avec le même nom, ce sera ce format qui sera utilisé.

Le format `NAME_TOP` sera affiché à chaque début de page s'il est défini.

`FORMLIST` est une séquence de lignes qui peuvent être de trois types :
- un commentaire, indiqué par un `#` en début de ligne
- une ligne *image*, donnant le format pour une ligne de sortie
- un ligne *arguments* fournissant les valeurs à insérer dans la ligne *image* précédente.

Les lignes images sont imprimées telles quelles, après substitution des champs d'interpolation.

Le tableau 8.1 résume les différents types de champs d'interpolation.

<table>
<tr><td>@<<<<</td><td>champ aligné à gauche</td></tr>
<tr><td>@||||</td><td>champ centré</td></tr>
<tr><td>@>>>></td><td>champ aligné à droite</td></tr>
<tr><td>@####</td><td>champ numérique (aligné à droite)</td></tr>
<tr><td>@#.##</td><td>champ numérique avec décimale</td></tr>
<tr><td>@*</td><td>champ multiligne</td></tr>
</table>

TAB. 8.1 – Champs d'interpolation

8.2 Utilisation

La fonction `write` envoie le format approprié sur le *filehandle* spécifié (`STDOUT` par défaut).

Voilà un exemple d'utilisation :

```
format STDOUT =
Nom : @<<<<<<<<<<<<    Prenom : @<<<<<<<<<<<<<
$nom, $prenom
Age : @###
```

```
$age
.

format STDOUT_TOP =
       Fiche de renseignement

.

while (&next_record)
# next_record affecte les variables $nom, $prenom, $age
{
   write ;
}
```

Pour préciser un nom de format différent de celui du filehandle, il faut fixer la variable `$~`, ou bien utiliser le module `FileHandle` :

```
use FileHandle;

format TABLE =
Nom : @<<<<<<<<< Prenom : @>>>>>>>>>>>
.

format_name STDOUT TABLE;
# équivalent (mais plus lisible) à
# select (STDOUT);
# $~ = TABLE;
```

Chapitre 9

Fonctions avancées

perl était à la base conçu pour effectuer différentes transformations sur des fichiers texte, mais il ne se réduit pas à cela. De nombreuses fonctions avancées en font un langage idéal pour diverses tâches d'administration-système.

Note : ce chapitre est censé être plus volumineux. J'intégrerai volontiers toute contribution sur l'utilisation de fonctions telles que `pack`, `unpack`, `tie`, etc.

9.1 Accès à la base NIS

perl permet d'accéder à la base NIS sans avoir à se préoccuper de la méthode d'accès.

Pour cela, diverses fonctions existent, qui sont les homonymes des fonctions équivalentes en C (`getpwnam, getpwuid, getpwent, ...`). Il suffit de se reporter à la page de manuel [3] pour en connaître les détails.

Par exemple, pour obtenir le *home directory* ainsi que le nom complet de la personne qui exécute le script, il suffit d'une ligne :

```
# $< renvoie l'uid de la personne qui exécute le script
($fullname, $home) = (getpwuid($<))[6,8];
```

9.2 Utilisation des fonctions réseau

Les différentes fonctions permettant d'utiliser les sockets sont présentes dans *perl*. Les noms sont identiques à ceux des fonctions C équivalentes. Cependant, les arguments diffèrent, pour deux raisons : les descripteurs de fichiers fonctionnent différemment en *perl* et en C, et de plus *perl* connait toujours la longueur de ses chaînes de caractères, donc cette information est inutile.

La section `perlipc` du manuel [3] donne un exemple de script client/serveur, illustrant l'utilisation des fonctionnalités réseau.

Une particularité est à souligner : l'utilisation des fonctions `pack` et `unpack` pour créer les structures qui sont passées en paramètres aux diverses fonctions.

9.3 Gestion des sémaphores et de la mémoire partagée

Comme pour les fonctions réseau, les différentes fonctions portent le même nom que leurs équivalents en C. Un exemple est proposé dans la section `perlipc` du manuel [3].

Chapitre 10

Perl WIN32/NT

10.1 Présentation

La machine virtuelle de *perl*, plus que celle de Java, est portée et supportée sur de nombreuses plateformes.

De plus, *perl* intègre des librairies d'interface au système sur lequel il s'exécute, ce qui facilite son intégration.

Dans ce chapître, nous verrons comment utiliser les alternatives et extensions de *perl* pour WIN32

10.2 Installation

La meilleure façon d'avoir un *perl* à jour est de prendre la dernière version, qui porte le nom de `perl5.xxxx-bindistxx-bc.zip` disponible sur tous les miroirs CPAN dans ports/WIN32/. Le chemin le plus plausible pour installer *perl* est dans la racine, i.e. `c :\perl`.

Comme rares sont les NT fournis avec un compilateur C (à part les compilateurs commerciaux ou gratuits e.g. le portage de GCC par cygnus), à l'installation est demandé si vous voulez disposer des sources.

Pour cette raison, beaucoup de modules nécessitant la compilation sont livrés pré-compilés, les fonctions réseau libnet et LWP, l'interface Tk, Database Interface, HTML, HTTP, MIME, Graphic Device (GD), MD5 et bien-sûr Win32 (OLE, ODBC, ...).

De plus il vous est proposé d'avoir la documentation au format HTML, ce qui est plus pratique hors d'un environnement facilitant l'utilisation de la ligne de commande.

N.B. Les versions distribuées par ActiveWare semblent toujours en retard d'une version, et peu fournies en modules précompilés.

10.3 Utiliser la base de registre

10.3.1 Installer une variable d'environnement via la base de registre

Pour ajouter le chemin vers l'interpréteur *perl* dans les variables de NT, il faut accéder au fichier de registres via `use WIN32;`.

La valeur à ajouter se trouve dans SYSTEM\CurrentControlSet\Control\Session Manager\Environment , on ouvre donc cette clef :

```
$p = "SYSTEM\\CurrentControlSet\\Control\\Session Manager\\Environment";
$main::HKEY_LOCAL_MACHINE->Open($p, $srv) or die "open: $!";
```

et on lit la table de hachage des valeurs :

```
$srv->GetValues(\%vals) or die "QueryValue: $!";
```

On peut vérifier si le chemin est déjà présent, sinon on l'ajoute :

```
if (!($vals{'Path'}[2] =~ /perl/i)) {
        $value=$vals{'Path'}[2];
        $srv->SetValueEx("Path", 0,REG_SZ, "c:\\perl\\bin;$value");
}
```

10.3.2 Installer un Script *perl* en tant que service (NT seulement)

Un service NT répond à un interfaçage de message pour le démarrage, la mise en pause et l'arrêt. Un script seul ne peut donc être installé.

Le NT Ressource Kit 3 fournit `INSTSRV` et `ANYSRV`, le premier installant le second qui est l'interface service.

Comme exemple on crée un service appelé PerlService :

```
`c:\\peliproc\\instsrv.exe PerlService c:\\peliproc\\srvany.exe`;
```

Cela installe une entrée dans la base de registres. Vous pouvez alors finir l'installation à la main comme spécifié dans la documentation de ces outils ou modifier via Win32 : :Registry les entrées :

```
$p = "SYSTEM\\CurrentControlSet\\Services\\PerlService";
$main::HKEY_LOCAL_MACHINE->Open($p,$srv)||die "open: $!";
```

et maintenant créer la clef Parameters contenant

Application le chemin complet vers le script

AppParameters les paramètres d'exécution

AppDirectory le chemin d'où il sera lancé

ce que l'on fait ainsi :

```
$srv->Create("Parameters", $param) or die "Create: $!";
$param->SetValueEx("Application", 0, REG_SZ, "c:\\perlproc\\scan.cmd");
$param->SetValueEx("AppParameters", 0, REG_SZ, "2>c:\\temp\\toto.out");
$param->SetValueEx("AppDirectory", 0, REG_SZ, "c:\\perlproc\\");
```

On lance donc `scan.cmd` dans le répertoire `c:\perlproc` (N.B. : : il faut répéter le \ séparateur de répertoire sous NT, qui sert à 'échapper' i.e. utiliser les caractères spéciaux sous perl, donc pour avoir \ on le double : \\).

L'argument est `2>c:\temp\toto.out`, on redirige la sortie erreur standard (STDERR sous perl, descripteur de fichier numéro 2 sous tout les O.S.) vers un fichier.

Maintenant il faut pouvoir démarrer le service.

```
`d:\\winnt\\system32\\net.exe start PerlService`;
```

La seule chose que cet exemple passe sous silence est que le service est démarré avec le compte SYSTEM qui sous NT a plus de priorités que les autres comptes, même Administrator. Il est possible en modifiant la base de Registres de spécifier un autre utilisateur, mais ... il faut rentrer un mot de passe, ce que je ne sais pas faire via la base de registres ...

10.4 Émuler une *crontab* sous NT

La commande `at` et son interface `winat` ne permettent que d'exécuter une tâche par jour, et ce sous le compte SYSTEM. Pour certaines tâches, cela est trop contraignant, e.g. scanner un répertoire toutes les 10 mn.

Pour cela, l'idéal est de créer un Service NT, et dans le script *perl* ajouter une boucle sans fin, et un `sleep(600)` pour 10 mn i.e. 10*60 secondes.

10.5 Accès disque et Serveurs de fichiers

10.5.1 Lire sur une ressource UNC sous NT

Cela se fait simplement en appelant la commande externe `NET`.

```
`d:\\winnt\\system32\\net.exe use \\serveur\ressource "mot_de_passe" /USER:"nom_d'utilisateur"'
```

L'idéal pour connaître la commande de retour est de rediriger avec `2>` vers un fichier et le scanner après coup.

À partir de là, l'environnement courant a mémorisé l'autorisation d'accès à cette ressource, et pour y accéder ensuite, il suffit de spécifier le chemin comme un chemin normal :

```
open(FIC,"\\serveur\ressource\vol1\DD-XF-FF.080") or warn "$0 $!";
```

10.5.2 Mapper un lecteur sous NT

Mapper un lecteur c'est associer à une ressource sur un réseau, un device, i.e. lecteur, représenté de `a :` à `z :`.

Cela se fait de la même façon sous NT, mais l'utiliser en tant que service pose le problème que l'utilisateur peut déjà utiliser le nom de device (seulement 23 disponibles au minimum). Un lecteur mappé ne l'est que dans le profil de l'utilisateur courant ; hors connexion ou sous un autre login, il ne le sera plus, à moins que ce nouveau profil ne le mappe lui-même.

Chapitre 11

Les références

Une des limitations de *perl4* venait du fait qu'il était impossible de créer et de manipuler des structures de données plus évoluées que de simples tableaux sans passer par des astuces plus ou moins efficaces.

Une des grandes innovations de *perl5* a été l'introduction des références, qui permettent de travailler sur des structures de données plus complexes.

11.1 Principe

Une référence est un scalaire qui pointe sur une structure de données (scalaire, tableau, tableau associatif, fonction).

Du fait de sa nature scalaire, il est possible de construire des tableaux de références. Et si ces références pointent sur des tableaux par exemple, on obtient ainsi des tableaux de tableaux, i.e. un tableau à deux dimensions.

perl garde trace du nombre de références sur un objet. Une fois que celui-ci atteint 0, l'objet est détruit et la mémoire libérée.

11.2 Déclaration - Syntaxe

Soit `@liste` un tableau. Alors on peut obtenir sa référence avec l'expression suivante : `$ref = \@liste`

Si on exécute la ligne `print $ref ;`, on obtient quelque chose du type :
`ARRAY(0x91a9c)`.

Il y a deux manières d'utiliser ensuite cette référence. On peut la déréférencer pour la manipuler comme un tableau normal :

```
@new = @$ref;
print $new[0];
```

ou bien on peut accéder directement aux données en remplaçant le nom du tableau par la référence :

```
print $$ref[0];
# ou bien
print $ref->[0];
```

On peut donc ainsi construire facilement des tableaux à plusieurs dimensions :

```
print $table[0]->[2]->[4];
```

Les flèches étant optionnelles entre les crochets ou les accolades, on peut également l'écrire :

```
print $table[0][2][4];
```

Ces exemples s'appliquent également aux références sur des tableaux associatifs ou des scalaires.

11.3 Création de références anonymes

Nous avons vu jusqu'ici comment créer une référence à partir d'une variable existante. Mais il est également possible de créer des références sur des structures anonymes en vue d'obtenir des structures de données plus complexes.

Voilà par exemple comment créer un tableau de tableaux :

```
@liste = ();

for $i ('A'..'z')
{
    push(@liste, [ $i, ord($i) ]);
}

print @{$liste[0]};
# -> affichage de ('A', 65) sous la forme A65
print ${$liste[0]}[0];
# -> affichage de 'A' sous la forme A
# ou bien
@paire = @{$liste[0]};
print $paire[0];
# -> affichage de 'A' sous la forme A
```

Ceci va créer un tableau à deux dimensions (ou plutôt un tableau de tableaux).

La notation `[]` renvoie une référence sur un tableau composé des éléments que les crochets renferment.

De la même manière, la notation `{}` renverra une référence sur un tableau associatif composé des éléments encadrés par les accolades :

```
@liste = ();

for $i ('A'..'z')
{
    push(@liste, { "caractere" => $i,
                   "valeur"    => ord($i) } );
}

print %{$liste[0]};
# -> affichage du tableau associatif ("caractere" => "A",
#                                    "valeur" => 65)
#    sous la forme caractereAvaleur65
print ${$liste[0]}{"caractere"};
# -> "A"
# ou bien
%paire = %{$liste[0]};
print $paire{"caractere"};
# -> "A"
```

On peut ainsi imbriquer plusieurs niveaux de références pour créer des structures de données complexes (mais attention à la mémoire utilisée !).

11.4 Les références sur les fonctions

On peut de la même manière que pour les tableaux créer des références sur des fonctions de deux façons différentes.

À partir d'une fonction déjà existante :

```
sub fonc {  ...; }
$ref = \&fonc;
```

ou bien en créant une référence sur une fonction anonyme :

```
$ref = sub { print "hello world !\n";};
```

On peut ensuite appeler directement :

```
&$ref;
```

11.5 Les objets

L'introduction des références a permis d'introduire également les concepts de la programmation orientée objets. Pour obtenir une documentation plus complète, se référer aux pages de manuel [3] `perlobj` et `perlbot`. Un tutorial est également accessible par la commande `perldoc perltoot`.

Attention : cette section ne constitue pas une introduction à la programmation objet. Il est considéré que le lecteur aura déjà été sensibilisé à certains de ses concepts.

11.5.1 Les bases

Voici les trois grands principes d'implémentation des objets en *perl5*.

- Un objet est simplement une référence qui sait à quelle classe elle appartient (voir la commande `bless`).
- Une classe est un paquetage qui fournit des méthodes pour travailler sur ces références.
- Une méthode est une fonction qui prend comme premier argument une référence sur un objet (ou bien un nom de paquetage).

En général, on utilise une référence sur un tableau associatif, qui contient les noms et les valeurs des variables d'instance.

11.5.2 Un exemple

Supposons que la classe `Window` soit définie. Alors on crée un objet appartenant à cette classe en appelant son constructeur :

```
$fenetre = new Window "Une fenetre";
# ou dans un autre style
$fenetre = Window->new("Une fenetre");
```

Si `Window` a une méthode `expose` définie, on peut l'appeler ainsi :

```
expose $fenetre;
# ou bien
$fenetre->expose;
```

Voici un exemple de déclaration de la classe `Window` :

```
package Window;

# La methode de creation. Elle est appelee avec le nom de la
# classe (ie du paquetage) comme premier parametre. On peut passer
# d'autres parametres par la suite
sub new
{
  # On recupere les arguments
  my($classe, $parametre) = @_;

  # L'objet qui sera retourne est ici (et c'est generalement le cas)
  # une reference sur un tableau associatif. Les variables
  # d'instance de l'objet seront donc les valeurs de ce tableau.
  my $self = {};

  # On signale a $self qu'il depend du paquetage Window
  # (ie que sa classe est Window)
  bless $self;

  # Diverses initalisations
  $self->initialize($parametre);
```

```
  # On retourne $self
  return $self;
}

# Methode d'initalisation.
# Le premier parametre est l'objet lui-meme.
sub initialize
{
  my($self, $parametre) = @_;

  $self->{'nom'} = $parametre || "Anonyme";
}

# Autre methode.
sub expose
{
  my $self = shift;

  print "La fenetre ‘‘, $self->{'parametre'},
        " a recu un evenement expose.\n";
}
```

Chapitre 12

Tips and tricks

Ce chapitre est destiné à répondre aux problèmes les plus fréquemment posés lors de l'écriture d'un script *perl*. Si vous ne trouvez pas la réponse à la question que vous vous posez, consultez la FAQ que vous trouverez sur `http ://www.perl.com/perl/`.

12.1 Comment récupérer la sortie d'un programme

Il existe deux manières pour effectuer cela : on peut d'abord utiliser des *backquotes* comme en shell. Par exemple :

```
$pwd = `/bin/pwd`;
chop($pwd);
```

La deuxième manière, qui offre plus de possibilités, est la suivante :

```
($pid = open(PIPE, "/bin/ps -a |")) or die "Error: $!\n";
(kill 0, $pid) or die "ps invocation failed";
while (defined($line = <PIPE>))
{
  ...;
}
close(PIPE);
```

Cette manière d'opérer se retrouve également dans l'opération inverse, qui consiste à envoyer des données sur l'entrée standard d'un programme :

```
($pid = open(PIPE, "| /usr/ucb/ftp")) or die "Error: $!\n";
(kill 0, $pid) or die "ftp invocation failed";
print PIPE "open ftp.enst-bretagne.fr\n";
...;
close(PIPE);
```

Dans l'exemple présenté ci-dessus, il serait plus intéressant de pouvoir envoyer des données sur l'entrée standard en même temps que de récupérer des données sur la sortie standard. Ceci est possible grâce à un module fourni dans la distribution de *perl* : `IPC : :Open2`.

12.2 Comment effacer ou copier un fichier ?

Pour effacer un fichier, il est inutile de faire un appel au programme `/bin/rm`, comme beaucoup de personnes le font. Il existe une instruction `unlink` qui appelle la fonction C du même nom et supprime le fichier dont le nom est passé en paramètre.

En revanche, il n'existe pas d'instruction pour copier un fichier. Le méthode conseillé est d'utiliser le module `File : :Copy`, ou bien on peut écrire soi-même les quelques lignes pour copier le fichier :

```
open(SOURCE, "<$source") || die "Error: $!\n";
open(DEST, ">$dest") || die "Error: $!\n";
while ($len = sysread(SOURCE, $buffer, 512))
{
  syswrite(DEST, $buffer, $len);
}
close(DEST);
close(SOURCE);
```

12.3 Les *here-documents*

Il existe en *perl* la possibilité d'utiliser les *here-documents* de manière analogue au shell, afin de faciliter l'écriture de longues chaînes de caractères. On utilise une chaîne de caractères particulière pour délimiter un bloc de texte, et on peut ainsi insérer de manière lisible des variables multilignes, ou encore utiliser des guillemets sans avoir à les précéder d'un backslash.

C'est très pratique par exemple pour écrire des scripts CGI :

```
print <<"FIN";
Content-type: text/html

<HTML><HEAD>
<TITLE>Erreur</TITLE>
</HEAD><BODY>
<H1>Erreur</H1>
$err_msg
</BODY></HTML>
FIN
```

L'utilisation de guillemets simples permet d'éviter que les variables ne soient interpolées à l'intérieur de la chaîne.

Il existe deux pièges à éviter lors de l'utilisation de ce mécanisme : il ne faut surtout pas oublier le point-virgule après le délimiteur de bloc, ce qui constitue une erreur de syntaxe.

De plus, l'identificateur de fin de bloc doit être impérativement au début de la ligne.

12.4 Comment savoir si une valeur est dans une liste ?

Ceci constitue une action fréquemment rencontrée lors de l'écriture de scripts. La solution dépendra de la taille de la liste.

Une première solution est de parcourir le tableau et de tester chaque valeur successivement :

```
$in = 0;
for $val (@liste)
{
  if ($item eq $val)
  {
    $in = 1;
    last;
  }
}
```

On peut également utiliser la fonction `grep` qui effectue un test sur tout un tableau, et renvoie l'ensemble des éléments du tableau qui ont validé le test :

```
@elements = grep($_ eq $item, @liste);
```

Enfin, et c'est la méthode la plus rapide, mais à n'utiliser que sur des listes de taille raisonnable, on peut utiliser un tableau associatif et tester si la valeur recherchée est une clé du tableau :

```
for $cle (@liste)
{
  $hash{$cle} = 1;
```

```
}
# ce qui peut s'ecrire de maniere moins lisible (mais plus
# courte a taper...) :
# %hash = map( ($_, 1), @liste )

if (defined($hash{$item}))
{
  $in = 1;
}
```

12.5 Utilisation des options de ligne de commande

Il existe de nombreuses options de ligne de commande qui permettent entre autre d'effectuer des actions simples.

Une liste complète de ces options se trouve dans la section `perlrun` du manuel.

12.5.1 Recherche/remplacement dans plusieurs fichiers

Pour effectuer ceci, nous utilisons deux options de ligne de commande. L'option `-p` lit chaque fichier dont le nom est passé en argument au script, effectue une action sur chaque ligne puis l'affiche sur la sortie standard.

Par exemple, le script suivant affichera le fichier passé en paramètre après avoir converti toutes les lignes en minuscules.

```
#!/usr/local/bin/perl -p
lc($_);
```

Il peut s'écrire également en ligne de commande de la manière suivante :

```
perl -p -e 'lc($_)' fichier.txt
```

L'option `-e` permet d'indiquer une expression à évaluer. Plusieurs options `-e` peuvent être fournies au script, auquel cas toutes les expressions passées seront concaténées, et le script résultant exécuté.

L'utilisation de ces options est particulièrement pratique lorsqu'elles sont combinées à l'option `-i` (qui signifie *in-place*).

Cette option modifie le comportement de l'option `-p` : chaque ligne, au lieu d'être réecrite sur la sortie standard, est écrite dans un fichier, qui sera ensuite recopié à la place du fichier original.

Si l'on fournit un paramètre à cette option (`-i.bak`), une copie du fichier original sera effectuée, en ajoutant l'extension précisée au nom du fichier, avant de procéder à la suite des opérations.

En pratique, si l'on veut substituer un mot pour un autre dans plusieurs fichiers, il suffit d'écrire :

```
perl -pi.bak -e 's/oldstr/newstr/g' *
```

12.5.2 Extraction d'informations

Soit un fichier au format HTML dont on veut extraire toutes les références. Ceci peut être effectué en une seule ligne, grâce à l'option `-n` qui lit chaque ligne des fichiers précisés, mais sans la réécrire ensuite. Attention, on fait ici l'hypothèse que toutes les URLs tiennent sur une seule ligne, ce qui n'est pas forcément le cas. Pour des gestions plus complètes, utilisez les modules adéquats (`HTML : :*`).

```
perl -ne 'print $1, "\n" while
         (/\<a\s+href\s*=\s*"(.+?)"\>/ig)' *.html
```

Un deuxième exemple :

```
ypcat passwd | perl -ne 'print $2, "\n"
                         if (/(.+?:){4}(.+?):.+\/bin\/sh/)'
```

12.6 Accès aux fichiers dbm

perl permet un accès très facile aux fichiers de base de données au format dbm. Il est possible d'établir une correspondance entre un tel fichier et un tableau associatif. Par la suite, les opérations effectuées sur le tableau associatif seront répercutées sur la base de données.

```
use NDBM_File;
tie(%HIST, NDBM_File, '/usr/lib/news/history', 1, 0);
print keys(%HIST), "\n";
delete $HIST{"cle"};
$HIST{"cle2"} = "element";
untie(%HIST);
```

12.7 Utilisation du débuggueur

Une caractéristique intéressante de *perl* est son débuggueur intégré, qui est d'ailleurs lui-même écrit en *perl*.

Il est invoqué par l'option `-d` :

```
perl -d script.pl
```

On peut également l'utiliser pour faire exécuter interactivement des commandes par l'interpréteur :

```
perl -d -e 1
```

Lors du débuggage d'un script, deux autres options sont également intéressantes.

`-c` vérifie uniquement la syntaxe du script, sans chercher à l'exécuter.

`-w` est beaucoup plus intéressant : il affiche divers messages concernant les variables non initialisées, ou bien utilisées une seule fois, etc.

Donc, pour tester la validité syntaxique d'un script, avant de le lancer, il suffit de le tester avec la commande

```
perl -wc script.pl
```

Chapitre 13

Les variables spéciales

Un certain nombre de variables spéciales sont prédéfinies par *perl*. Je présenterai ici les plus importantes. Pour une liste complète, se reporter au *Perl Reference Guide* [2] ou à la section `perlvar` du manuel [3].

13.1 Les variables spéciales scalaires

- `$_` La variable par défaut pour de nombreuses fonctions
- `$?` Le statut retourné par la dernière commande `system` ou pipe
- `$]` Le numéro de version de *perl*
- `$!` Dans un contexte numérique, renvoie la valeur de `errno`
  Dans un contexte de chaîne, renvoie le message d'erreur correspondant
- `$@` Le message d'erreur de la dernière commande `eval` ou `do`
- `$0` Le nom du fichier contenant le script exécuté. Cette variable peut être assignée
- `$$` Le pid du script. Altéré par `fork` dans le processus fils
- `$<, $>` Les uid (*real* et *effective*) du script
- `$(, $)` Les gid (*real* et *effective*) du script
- `$|` Si non nulle, force un *flush* après chaque opération de lecture ou d'écriture sur le filehandle courant
- `$1, $2, ...` Voir le chapitre 5 sur les expressions régulières

13.2 Les variables spéciales de tableaux

Elles sont beaucoup moins nombreuses que les scalaires :

- `@ARGV` Arguments de la ligne de commande
- `@INC` Chemins de recherche des fichiers requis par `require` ou `use`
- `@_` Tableau contenant les paramètres des routines

Attention : le premier élément du tableau `@ARGV` (donc la valeur `$ARGV[0]`) correspond au premier argument passé, et non au nom du script comme pourrait s'y attendre n'importe quel programmeur C. Le nom du script est disponible dans la variable `$0`.

13.3 Les variables spéciales de tableaux associatifs

- `%ENV` Liste des variables d'environnement
- `%SIG` Utilisé pour préciser les *signal-handlers*
- `%INC` Liste des fichiers qui ont été appelés par `require`

Index

Bibliographie

[1] Schwartz (Randal L.). – *Learning Perl.* – O'Reilly & Associates, November 1993. Excellent ouvrage d'initiation.

[2] Vromans (Johan). – *Perl Reference Guide.* – `jv@nl.net`. Carnet de référence contenant toutes les commandes classées par sujet. Le source LaTeX est disponible par ftp sur les sites CPAN.

[3] Wall (Larry). – *Perl reference pages.* – `lwall@netlabs.com`. Les pages de manuel. Constituent une référence très bien structurée et facile à consulter. La dernière version se trouve sur les sites CPAN.

[4] Wall (Larry) et Schwartz (Randal L.). – *Programming perl.* – O'Reilly & Associates, 1996, 2nd édition. L'ouvrage de référence (attention, la première édition ne couvre par les particularités de *perl5*. Disponible à la bibliothèque.