P.E.R.L.

PRACTICAL EXTRACTING AND REPORT LANGUAGE

Table des matières

Chapitre 1 1

1 INTRODUCTION 1

1.1 Présentation 1
1.2 Usages 1
1.3 Lectures 2
1.4 Sources, documents, FAQs 2

Chapitre 2 3

2 COMMENT CRÉER UN PROGRAMME PERL 3

2.1 Définition d'un programme Perl 3
2.1.1 Edition 3
2.1.2 Sauvegarde 3
2.1.3 L'interpréteur Perl 3
2.1.4 Lancement d'un programme 4

Chapitre 3 5

3 LES STRUCTURES DE DONNÉES 5

3.1 Les scalaires 5
3.1.1 Les données scalaires 5
3.1.2 Les variables scalaires 6
3.2 Les tableaux et les listes 6
3.2.1 Construction des listes 6
3.2.2 Les variables de tableaux 6
3.3 Le "hash" ou tableau associatif 7
3.3.1 Construction des tableaux associatifs 7
3.3.2 Les variables de tableaux associatifs 7

3.4 Les "pointeurs" 7
3.5 Les références 7
3.6 Récupération des données: déréférence. 8
3.7 Les objets 8
3.7.1 Les bases 8
3.7.2 Un exemple 8
3.8 Variables prédéfinies 9

Chapitre 4 11

4 SYNTAXE ET STRUCTURES DE CONTRÔLE 11
4.1 Syntaxe 11
4.2 Les délimiteurs 11
4.2.1 Commandes simples 11
4.2.2 Blocs de commandes 12
4.3 Assignation de valeur 12
4.3.1 Format numérique 12
4.3.2 Contexte alphanumérique 12
4.3.3 Assignation aux variables 12
4.4 Structures de contrôle 12
4.4.1 La commande while 13
4.4.2 La commande for 13
4.4.3 La commande foreach 13
4.4.4 Comment émuler switch? 13

Chapitre 5 15

5 LES OPÉRATEURS PERL 15
5.1 Règles de priorité 15
5.2 Les fonctions et les listes 16
5.3 L'opérateur flèche 16
5.4 Opérateur d'incrémentation et de décrémentation 16
5.5 Exponentielle 16
5.6 Symbole unaire 16
5.7 Opérateur de lien 16
5.8 Opérateur de multiplication 16
5.9 Opérateur additif 16
5.10 Opérateur de décalage 16
5.11 Fonction unaire 17
5.12 Opérateurs de comparaison 17
5.13 Opérateur d'égalité 17
5.14 Opérateur logique bit à bit 17
5.15 Opérateur logique 17
5.16 Opérateur d'étendue 17
5.17 Guillemets 17
5.18 Les opérateurs de tests sur les fichiers 18

Chapitre 6 19

6 STRUCTURE DES PROGRAMMES 19
6.1 Structure d'un programme 19
6.2 Les procédures 19
6.3 Transmission des arguments 20
6.3.1 Transmission des arguments par valeur 20
6.3.2 Transmission des arguments par adresse 20
6.4 Les variables locales 20
6.5 Fonction ou procédure 20
6.6 Appel 21
6.7 Deux fonctions particulières 21
6.8 Les paquetages 21
6.9 Les modules 22
6.9.1 Principe 22
6.9.2 Modules existants 22

Chapitre 7 23

7 LES ENTRÉES-SORTIES 23
7.1 Les bases 23
7.2 Interaction 23
7.3 L'ouverture 24
7.4 La lecture 24
7.5 L'écriture 24
7.6 La fermeture 25
7.7 Le buffering 25

Chapitre 8 27

8 LES EXPRESSIONS RÉGULIÈRES 27
- *8.1 La syntaxe 27*
  - 8.1.1 Les métacaractères 27
  - 8.1.2 D'autres caractères spéciaux 28
  - 8.1.3 Les quantificateurs 28
  - 8.1.4 Les quantificateurs non-gourmands 28
- *8.2 Utilisation : recherche 29*
  - 8.2.1 Syntaxe 29
  - 8.2.2 Valeurs de retour 29
- *8.3 Utilisation: substitution 30*
- *8.4 Utilisation: translation 30*
- *8.5 Un exemple : Inversion de deux mots 30*

Chapitre 9 33

9 LES RÉPERTOIRES ET LES FICHIERS 33
- *9.1 Accès aux répertoires 33*
  - 9.1.1 Parcourir l'arborescence 33
  - 9.1.2 Globaliser 34
  - 9.1.3 Handle de répertoires 34
  - 9.1.4 Ouvrir et fermer un handle de répertoire 34
  - 9.1.5 Lire un handle de répertoire 34
- *9.2 Manipulation des réperoires et fichiers 35*
  - 9.2.1 Supprimer un fichier 35
  - 9.2.2 Renommer un fichier 35
  - 9.2.3 Les liens 35
    - 9.2.3.1 Liens solides et liens symboliques 35
    - 9.2.3.2 Créer des liens solides et symboliques avec Perl 35
  - 9.2.4 Créer et supprimer des répertoires 36
  - 9.2.5 Modifier les autorisations 36
  - 9.2.6 Modifier la propriété 36
  - 9.2.7 Modifier les repères de temps 36

Chapitre 10 37

10 GESTION DE PROCESSUS 37
- *10.1 Utilisez system et exec 37*
- *10.2 Apostrophes inverses 38*
- *10.3 Utiliser les processus en tant que handles de fichiers 38*
- *10.4 Utiliser fork 39*
- *10.5 Envoyer et recevoir des signaux 39*

Chapitre 11 41

11 NOTIONS SUPPLÉMENTAIRES 41
- *11.1 Les formats 41*
  - 11.1.1 Syntaxe 41
  - 11.1.2 Utilisation 42
- *11.2 Débuggage 42*
  - 11.2.1 Fonctionnement 42
  - 11.2.2 Commandes 42
- *11.3 Comment récupérer la sortie d'un programme 43*
- *11.4 Comment effacer ou copier un fichier 43*
- *11.5 Comment savoir si une valeur est dans une liste 43*

Chapitre 12 45

12 LES FONCTIONS PERL PAR CATÉGORIE 45
- *12.1 Les fonction de manipulation de listes 45*
- *12.2 Les fonctions sur les tableaux associatifs 46*
- *12.3 Les fonctions de manipulation de chaînes 47*
- *12.4 Les fonctions d'Entrée-Sortie 47*
- *12.5 Les fonctions sur les répertoires et processus 48*
- *12.6 Les fonctions de calcul mathématique 48*
- *12.7 Les autres fonctions 48*

Petit référentiel Perl 49

/PATTERN/ 49
?PATTERN? 49
abs (VALEUR) (Perl 5) 49
accept(NEWSOCKET,GENERICSOCKET) (UNIX) 49
alarm(SECONDS) (UNIX) 50
atan2(Y,X) 50
bind(SOCKET,NAME) (UNIX) 50
binmode(FILEHANDLE) 50
bless (REF, [PACKAGE]) (Perl 5) 50
caller([EXPR]) 50
chdir(EXPR) 50
chmod(acces, LIST) (Unix) 50
chomp([VARIABLE|LISTE]) (Perl 5) 50
chop([LIST]) 50
chown(uid, gid, LIST)(Unix) 50
chr NOMBRE (Perl 5) 50
chroot(FILENAME)(Unix) 50
close(FILEHANDLE) 50
closedir(DIRHANDLE) 51
connect(SOCKET,NAME) (Unix) 51
cos(EXPR) 51
crypt (TEXTE,CLE) (Perl 5) 51
defined(EXPR) 51
delete (EXPR) (Perl 5) 51
die(LIST) 51
do BLOCK 51
do SUBROUTINE (LIST) 51
do EXPR 51
dump LABEL(Unix) 51
each(ASSOC_ARRAY) 51
eof(FILEHANDLE) 51
eval(EXPR) 51
exec(LIST) 51
exists (EXPR) (Perl 5) 51
exit(EXPR) 52
exp(EXPR) 52
fcntl(FILEHANDLE,FUNCTION,SCALAR) (Unix) 52
fileno(FILEHANDLE) 52
flock(FILEHANDLE,OPERATION) (Unix) 52
fork (Unix) 52
formline (PICTURE, LIST) (Perl 5) 52
getc(FILEHANDLE) 52
getlogin (Unix) 52
getpeername(SOCKET) (Unix) 52
getpgrp(PID) (Unix) 52
getppid (Unix) 52
getpriority(WHICH,WHO) (Unix) 52
getpwnam NAME, getgrnam NAME, gethostbyname NAME, getnetbyname NAME, getprotobyname NAME, getpwuid UID , getgrgid GID , getservbyname NAME,PROTO , gethostbyaddr ADDR,ADDRTYPE , getnetbyaddr ADDR,ADDRTYPE , getprotobynumber NUMBER , getservbyport PORT,PROTO getpwent , getgrent , gethostent , getnetent , getprotoent , getservent setpwent , setgrent , sethostent STAYOPEN , setnetent STAYOPEN , setprotoent STAYOPEN , setservent STAYOPEN , endpwent , endgrent , endhostent , endnetent , endprotoent , endservent (Perl 5) 52
getsockname SOCKET (Perl 5) 53
getsockopt SOCKET,LEVEL,OPTNAME (Perl 5) 53
glob EXPR (Perl 5) 53
gmtime(EXPR) 53
goto LABEL 53
grep(EXPR,LIST) 53
hex(EXPR) 53
index(STR,SUBSTR,POSITION) 53
import : (Perl 5) 53
int(EXPR) 53
ioctl(FILEHANDLE,FUNCTION,SCALAR) (Unix) 53
join(EXPR,LIST) 53
keys(ASSOC_ARRAY) 53
kill(numero,LIST) (Unix) 53
last LABEL 54
lc EXPR : (Perl 5) 54
lcfirst EXPR : (Perl 5) 54
length(EXPR) 54
link(FILE1,LINK)(Unix) 54
listen(SOCKET,QUEUESIZE)(Unix) 54
local(LIST) 54
localtime(EXPR) 54
log(EXPR) 54

lstat(FILEHANDLE)(Unix) 54
m/PATTERN/gio 54
map BLOCK|EXPR LIST : (Perl 5) 54
mkdir(REP,MODE) 55
msgctl(ID,CMD,ARG), msgget(KEY,FLAGS)
msgsnd(ID,MSG,FLAGS), msgrcv(ID,VAR,SIZE,TYPE,FLAGS)... 55
my EXPR : (Perl 5) 55
next LABEL 55
no Module LIST : (Perl 5) 55
oct(EXPR) 55
open(FILEHANDLE,EXPR) 55
opendir(DIRHANDLE,EXPR) 55
ord(EXPR) 55
pack(TEMPLATE,LIST) 55
pipe(READHANDLE,WRITEHANDLE) (Unix) 56
pop(ARRAY) 56
pos SCALAR : (Perl 5) 56
print(FILEHANDLE LIST) 56
printf(FILEHANDLE LIST) 56
push(ARRAY,LIST) 56
q/STRING/, q/STRING/, qq/STRING/ et qx/STRING/ 56
quotemeta EXPR : (Perl 5) 57
rand(EXPR) 57
read(FILEHANDLE,SCALAR,LENGTH,OFFSET) 57
readdir(DIRHANDLE) 57
readlink(EXPR) (Unix) 57
recv(SOCKET,SCALAR,LEN,FLAGS) (Unix) 57
redo LABEL 57
ref EXPR : (Perl 5) 57
rename(OLDNAME,NEWNAME) 57
require(EXPR) 57
reset(EXPR) 57
return LIST 57
reverse(LIST) 57
rewinddir(DIRHANDLE) 57
rindex(STR,SUBSTR,POSITION) 57
rmdir(FILENAME) 57
s/PATTERN/REPLACEMENT/gieo 58
scalar(EXPR) 58
seek(FILEHANDLE,POSITION,WHENCE) 58
seekdir(DIRHANDLE,POS) 58
select(FILEHANDLE) 58
semctl(ID,SEMNUM,CMD,ARG) (Unix) 58
semget(KEY,NSEMS,SIZE,FLAGS) (Unix) 58
semop(KEY,OPSTRING) (Unix) 58
send(SOCKET,MSG,FLAGS,TO) (Unix) 58
setpgrp(PID,PGRP) (Unix) 58
setpriority(WHICH,WHO,PRIORITY) (Unix) 58
setsockopt(SOCKET,LEVEL,OPTNAME,OPTVAL) (Unix) 58
shift(ARRAY) 58
shmctl(ID,CMD,ARG) shmget(KEY,SIZE,FLAGS)
shmread(ID,VAR,POS,SIZE) shmwrite (ID,STRING,POS,SIZE)
(Unix) 58
shutdown(SOCKET,HOW) (Unix) 58
sin(EXPR) 59
sleep(EXPR) 59
socket(SOCKET,DOMAIN,TYPE,PROTOCOL) (Unix) 59
socketpair(SOCKET1,SOCKET2, DOMAIN,TYPE,PROTOCOL)
(Unix) 59
sort(SUBROUTINE LIST) 59
splice(ARRAY,OFFSET,LENGTH,LIST) 59
split(/PATTERN/,EXPR,LIMIT) 59
sprintf(FORMAT,LIST) 59
sqrt(EXPR) 59
srand(EXPR) 59
stat(FILEHANDLE) 59
study SCALAR|$_ : (Perl 5) 59
substr(EXPR,OFFSET,LEN) 59
symlink(OLDFILE,NEWFILE) (Unix) 60
syscall(LIST) (Unix) 60
sysread(FILEHANDLE,SCALAR,LENGTH,OFFSET) (Unix) 60
system(LIST) (Unix) 60
syswrite(FILEHANDLE,SCALAR,LENGTH,OFFSET) 60
tell(FILEHANDLE) 60
telldir(DIRHANDLE) 60
tie VARIABLE,PACKAGENAME,LIST : (Perl 5) 60
Time 60
times : (Perl 5)(Unix) 60
tr/SEARCHLIST/REPLACEMENTLIST/cds 60
truncate(FILEHANDLE,LENGTH) (Unix) 60
uc EXPR : (Perl 5) 61
ucfirst EXPR : (Perl 5) 61

umask(EXPR) ()Unix *61*
undef(EXPR) *61*
unlink(LIST) *61*
unpack(TEMPLATE,EXPR) *61*
unshift(ARRAY,LIST) *61*
untie VARIABLE : (Perl 5) *61*
utime(LIST) *61*
use MODULE [LIST] : (Perl 5) *61*
values(ASSOC_ARRAY) *61*
vec(EXPR,OFFSET,BITS) *61*
wait (Unix) *61*
waitpid(PID,FLAGS) (Unix) *61*
wantarray *61*
warn(LIST) *61*
write(FILEHANDLE) *61*
y/// : (Perl 5) *61*

Bibliographie **63**

Chapitre 1

1 Introduction

1.1 Présentation

Perl est un langage interprété (avec une phase interne de précompilation) optimisé pour traiter des fichiers texte, mais qui peut également être utilisé pour diverses tâches d'administration système. Perl est un langage de programmation simple d'approche et de compréhension, pour qui connaît les langages structurés tels que le C ou le PASCAL. Perl est un langage interprété, comme le CShell, optimisé pour parcourir des fichiers textes, extraire des informations de ces fichiers et écrire des rapports basés sur cette information (statistiques d'accès à un serveur, convertir du texte en format HTML…).

Le Perl est développé pour être pratique, facile à utiliser, efficace, complet plutôt que beau!!! En résumant vite, Perl reprend le meilleur du C, sed, awk et du sh. Sa syntaxe de programmation est quasiment identique à celle du C, mises à part certaines exceptions. 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).

1.2 Usages

On peut utiliser Perl pour récupérer et traiter des champs de données provenant d'un navigateur WEB via un serveur et créer dynamiquement des pages HTML. Mais Perl n'est pas limité à cet aspect traitement de rapport. Il est capable de gérer votre système aussi bien que le sed, awk ou sh. Nous verrons comment utiliser du code écrit en Perl, comment créer un programme Perl, comment débuger et enfin comment utiliser Perl sur le réseau Internet.

Une grande quantité de modules déjà disponibles permet de développer rapidement des applications touchant à des domaines divers (CGI, Tk, 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.

1.3 Lectures

L'ouvrage de référence pour Perl est le livre Programming Perl [1], connu aussi sous le nom de Camel book, co-écrit par Larry Wall, l'auteur du langage. Dans tous les cas, on se reportera au manuel [2] qui constitue la référence la plus précise et la plus à jour.

Il existe également un ouvrage d'initiation : Learning Perl [3], surnommée 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 [4] qui se rend très rapidement indispensable. Les fonctions y sont groupées par type, ce qui rend sa consultation très aisée.

1.4 Sources, documents, FAQs

Le réseau CPAN, abréviation de Comprehensive Perl Archive Network, a été mis en place pour 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/
- ftp://ftp.ibp.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.
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,
- http://www.Perl.org/ qui est maintenu par un groupe d'utilisateurs.

Pour des informations vraiment techniques, préférez le premier.

Pour conclure, j'ai essayé d'argumenter au maximum par des exemples l'ensemble des développements de ce document, mais souvenez-vous que la pratique vaut tous les cours du monde.

Chapitre 2

2 Comment créer un programme Perl

On aborde ici tout ce qui touche à l'aspect pratique du Perl, c'est à dire la réponse à l'interrogation: "Par o_ u et comment commence-t-on?"

2.1 Définition d'un programme Perl

2.1.1 Edition

Vous devez utiliser votre éditeur de texte préféré pour taper votre programme Perl.

2.1.2 Sauvegarde

Lorsque vous aurez tapé votre programme Perl, sauvez-le. Sachez qu'en général les modules Perl ont les extensions suivantes par convention :

- pl :Programme principal(contient l'appel à l'interpréteur de commande),
- pm: Fichier contenant les fonctions (modules),
- ph: Fichier contenant les variables (données).

2.1.3 L'interpréteur Perl

Le programme principal en Perl débute toujours par une déclaration destinée à l'interpréteur de commande de votre système (sh, csh, bash, …). Au début du programme principal, il vous faudra toujours faire figurer la commande du type:

```
#!<PATH_TO_PERL/perl> -options de compilation.
```

Exemple: `#!/usr/local/bin/perl -w`

Le symbole #! fait comprendre à l'interpréteur de votre système qu'il doit interpréter votre fichier à l'aide du programme localisé dans <PATH_TO_PERL>.

Les options de "compilation" de Perl. Les options de compilation peuvent être très utiles. On a entre autres:

- w permet l'affichage de tous les warnings lors de la compilation,
- d mode débbugage,
- c permet de vérifier la syntaxe du programme sans l'exécuter.

2.1.4 Lancement d'un programme

Sous UNIX, dans le répertoire contenant le fichier que vous venez de créer, tapez le nom du fichier contenant votre programme principal

Voilà le programme est lancé. En cas de problème à la compilation, lisez attentivement les messages où figurent les lignes problématiques. Si les problèmes persistent, essayez l'option de compilation -w. Si vous arrivez à compiler, mais que votre programme ne fait pas ce que vous voulez, utilisez le débuggeur. Enfin, il faut savoir qu'un programme Perl est interprété à chaque fois qu'il est lancé. Il n'y a pas d'exécutable Perl en langage machine de créé. C'est peut être là un point faible du Perl.

Chapitre 3

3 Les structures de données

On attaque ici les fondements du Perl. On verra au cours des chapitres suivants la syntaxe et la structure du Perl. Une des particularités du Perl réside dans le fait qu'il n'est pas obligatoire de déclarer une variable tant en type qu'en valeur. En effet lors de sa première utilisation le compilateur Perl décidera quel est le type de la variable en fonction de la valeur qui lui est affectée. On a trois structures de données de base en Perl :

- les scalaires
- les tableaux de scalaire (array)
- les tableaux d'association de scalaires (hashes)

N.B. Un scalaire est aussi bien un nombre ou une chaîne de caractère (il n'y a pas de type défini en Perl).

3.1 Les scalaires

3.1.1 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 exemples suffisent : 123, 123.45, 123.45e10, 1 234 567 (les caractères sont ignorés),

0xffff (valeur hexadécimale), 0755 (valeur octale), … 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).

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").

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).

3.1.2 Les variables scalaires

Il s'agit d'une variable au sens usuel qui contient un et un seul élément. Un "scalaire" peut, la première fois qu'il est appelé, être aussi bien utilisé ou déclaré. La variable peut référencer un élément d'un tableau. Un scalaire est déclaré ou appelé par un nom précédé du symbole $

Exemple:
```
$toto est une variable.
$toto = 10 #la variable $toto est déclarée
if ($toto == 10) …
#la variable $toto est utilisée dans un contexte numérique.

$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"
```

3.2 Les tableaux et les listes

3.2.1 Construction des listes

Une liste est un ensemble de valeurs scalaires, 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, "chaîne", 0x44, $var)
```

La liste vide est représentée par ().
Il existe également d'autres fonctions de construction de tableaux, qui clarifient souvent l'écriture des scripts. Ce sont les fonctions qw, q, qq, qx. Elles permettent 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.);
```

3.2.2 Les variables de tableaux

Un tableau est déclaré par un nom précédé du symbole @. Les éléments d'un tableau sont appelés comme suit :
Exemple : $tableau[i] est le $i^{ème}$ élément du tableau @tableau.
@tableau=(1,2,3,4); création d'un vecteur.
$tab=[[1,2,3],[4,5,6],[7,8,9]]; création d'une matrice @$tab
$$tab[2][1] contient: 8

Les tableaux commence à l'indice 0 par défaut. On peut y remédier par exemple par :

```
@tab1[1..10] est la déclaration d'un tableau qui s'étend de 1 à 10.
```

Vous n'avez pas à vous préoccuper de la dimension du tableau, Perl la gère pour vous. L'allocation est dynamique lors de l'interprétation du programme.

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), mais l'expérience montre que l'utilisation de scalar a le mérite d'expliciter la conversion. A vous de choisir...
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 même que pour les variables scalaires, la variable @ est prise par défaut pour de nombreuses fonctions.

3.3 Le "hash" ou tableau associatif

3.3.1 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.

3.3.2 Les variables de tableaux associatifs

Un `hash" est une liste associative de binôme. Il est appelé ou déclaré par son nom précédé de %. Il s'agit donc d'une structure qui associe par pair, un nom à une valeur. On les déclare ainsi:

Exemple : en utilisant une liste:

```
%map=('red',0x00f,'blue',0x0ef);
```

ou, par affectation:

```
$map{'red'}=0x00f;
```

ou, en utilisant les pointeurs

```
$map->{'red'}=0x00f
print `$$map{'red'} #écrit: 0x00f
```

ou,

```
$map=(
        'red'=> 0x00f;
        'blue'=> 0x0ef);
```

%map sera un tableau associatif.

3.4 Les "pointeurs"

Il existe en Perl deux sortes de références (pointeur). On a la référence symbolique et la référence "hard".
La référence symbolique utilise le symbole *. Une telle référence contient le nom de la variable pointée, tout comme un lien symbolique sous Unix contient le nom d'un fichier. Nous ne nous étendrons pas d'avantage sur ce type de pointeur. L'autre référence est appelée une référence "hard". Cette référence est appelée par l'opérateur \. En Perl version 5, les références "hard" permettent à un scalaire d'être une référence sur un tableau, un hash, … Il est donc facile de réaliser des tableaux de tableaux, tableaux de fonctions, …

3.5 Les références

La référence "hard" est appelée par l'opérateur \. Voici quelques exemples :
Exemple :

```
$pointeur=\$variable  # pointeur de la variable $variable.
                      # $pointeur contient l'adresse de $variable.
$pointeur=\@tableau #pointeur sur un tableau
$pointeur=\&fonction #pointeur sur une fonction ou procédure
$pointeur=%hash #pointeur sur un tableau associatif
```

Voici un exemple de référence sur un tableau.
Exemple : $tableau=[1,2,['a','b','c']]
$tableau est un pointeur sur un tableau que l'on dit anonyme (grâce au crochet).

Voici un exemple de référence sur un "hash".
Exemple : $hash_ref={'adam'=>'eve','bonnie'=>'clyde'};

Le principe ici est d'affecter un tableau associatif anonyme (grâce aux accolades) a un scalaire. $hash_ref est un pointeur.

3.6 Récupération des données: déréférence.

Pour récupérer les données contenus dans un pointeur, vous devez déréférencer par le symbole définissant le type du contenu.
Ainsi:

Exemple : $bar=\"Salut"; # On pointe sur un scalaire (chaîne de caractères)
print $bar"; #On récupère la valeur pointée en l'interprétant comme un scalaire à l'aide du symbole $.

$tableau=[1,2,['a','b','c']]
$$tableau[2][1] # contient b. (On récupère un scalaire)
@$pointeur #équivaut à un type tableau.

$pointeur=\%hash On pointe sur un tableau associatif
%$pointeur , %hash (On récupère un tableau associatif)
$pointeur=\&subroutine
&$pointeur pour appeler la fonction

On a donc le choix pour référencer et déréférencer des variables. Notez que l'on a vu qu'une partie des possibilités syntaxiques pour les pointeurs. On se méfiera devant cette débauche d'artifice pour référencer ou déréférencer.

3.7 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 perlobj et perlbot.

3.7.1 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.

3.7.2 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";
}
```

3.8 Variables prédéfinies

Certaines variables ont une signification particulière en Perl. Il existe, pour la plupart de ces variables deux définitions: une version abrégée et une version longue. Pour utiliser la version longue indiquez au début du programme use English;. Ceci créera un alias entre les noms courts et les noms longs. Passons en revue une liste non exhaustive, mais utile de ces variables. En gras figure le nom court et entre parenthèses le nom long.

- **$_** ($ARG): Variable par défaut (renvoyée par certains opérateurs)
  Exemple : while (<>){…}
  while ($_ =<>){…} sont équivalents

- **$&** ($MATCH):La chaîne comparée par le dernier opérateur de comparaison (/ /).

- **$`** ($PREMATCHING):La chaîne précédant ce qui a été comparée avec succès.

- **$'** ($POSTMATCHING): La chaîne suivant ce qui a été comparée avec succès
  Exemple : $ ='abcdefghi';
  /def/;
  print "$`:$&:$' "; #écrit abc:def:ghi !

- **$]** ($Perl VERSION): Contient la version du Perl.

- **$?** : Le statut retourné par la dernière commande system ou pipe,

- **$!** : 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, ...** Mémorisations de chaînes dans les expressions régulières

- @**ARGV**: contient la liste des arguments passés lors de l'appel du script Perl. $#ARGV est généralement le nombre d'argument moins un passé au script.

- **@INC**: contient la liste des chemins où Perl doit aller pour trouver les librairies. Généralement il contient \/usr/lib/perl5" et \." pour le répertoire courant. Pour y ajouter vos propres chemins menant à vos librairies tapez: @INC= (@INC,PATH TO LIBRARY);

- @_ : 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.

- **%ENV** : contient les variables de votre environnement.

- **%INC** : Liste des fichiers qui ont été appelés par require.

Chapitre 4

4 Syntaxe et structures de contrôle

Tout comme le C, Perl est assez souple au point de vue de la syntaxe. Méfiance donc! Attention, les minuscules et majuscules sont distinctes en Perl.

4.1 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 utilisez dessus).
Les commentaires sont introduits par le caractèere #, et s'étendent jusqu'à la fin de la ligne.
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
```

4.2 Les délimiteurs

- **;** : Marque la fin d'une ligne de code.
- **( )** : Délimite une liste.
- **[ ]** : Délimite un indice de tableau, mais est aussi une classe de caractères.
- **#** : Masque les caractères à droite jusqu'au retour à la ligne.
- **{ }** : Délimite un bloc d'instruction

4.2.1 Commandes simples

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

```
if EXPR
```

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

Par exemple :

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

4.2.2 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 {}.

4.3 Assignation de valeur

Comme vous l'avez remarqué précédemment, le type d'une variable dépend du contexte dans lequel est utilisé cette variable.

4.3.1 Format numérique

Le format numérique est le format habituel.

Exemple :
```
12.34 #format décimal
1234E-2 #idem
0x0ef#format hexadécimal
0377#format octal
```

4.3.2 Contexte alphanumérique

Les chaînes de caractères peuvent être délimitées par une apostrophe ou un guillemet. Les chaînes entre guillemets sont interprétées, c'est à dire que les variables ou certains symboles tels que \n (retour à la ligne) sont remplacés par leur signification. Par contre, les chaînes entre apostrophes ne sont pas interprétées.

Exemple :
```
print '$toto'; #écrit $toto
print "$toto"; #écrit 12.34
```

4.3.3 Assignation aux variables

Les scalaires Pour donner au scalaire toto la valeur 12.34, on opère ainsi

Exemple :
```
$toto=12.34;
$name="Bonjour\n";
```

Les tableaux On procède de la façon suivante pour affecter une valeur à un tableau :

Exemple :
```
@tableau=(1,2,3); #$tab[0] vaut 1…
$tableau[0]="Bonjour" #On affecte une chaîne au premier élément du tableau
$pointeur=[1,2,3];
$pointeur->[0]="Bonjour"; #idem;
```

Les listes associatives On a entre autres deux façons pour déclarer une liste (hash): La première nous l'avons vu. L'autre façon est d'affecter la valeur de chaque paire.

Exemple :
```
$map{'red'}=0x00f;
$pointeur=%map; #idem
$pointeur->{'red'}=0x00f;
```

4.4 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
```

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.

4.4.1 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);
        }
}
```

4.4.2 La commande for

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

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

4.4.3 La commande foreach

La commande foreach 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 un exemple, ne cherchez pas d'utilité à cette action.
    $arg = "";
}
```

4.4.4 Comment émuler switch?

Il n'y a pas de commande switch prédéfini, 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.

Chapitre 5

5 Les Opérateurs Perl

On passe ici en revue les opérateurs Perl. En général, ils fonctionnent comme en C.

5.1 Règles de priorité

On peut définir un ordre décroissant de priorité dans les opérateurs. C'est à dire que l'on commencera toujours par évaluer l'opération de plus haute priorité.

Relation à	Opérateur
gauche	Fonction et liste
gauche	->
non associatif	++, --
droite	**,
droite	!, ~, \, les unaires + et -
gauchegauche	=~, !~, ++, /, %, x
gauche	+, -, .
gauche	<<, >>
non associatif	fonction à argument
non associatif	<, >, <=, >=, lt, gt, ge
non associatif	==, !=, ⇔, eq, ne, cmp
gauche	&
gauche	|^
gauche	&&
gauche	||
non associatif	..
droite	?:
droite	=, +=, -=, *=, etc.
gauche	, =>
gauche	not
gauche	and
gauche	or, xor

5.2 Les fonctions et les listes

Elles ont la plus haute priorité. Cela sous entend les variables, les guillemets, les apostrophes, les opérations entre parenthèses et toutes les fonctions dont les arguments sont entre parenthèses. En l'absence de parenthèses, la priorité de fonctions telles que "print", "sort" est soit la plus haute soit la plus basse. Cela dépend si vous considérez la partie à gauche ou la partie à droite. Par exemple :

```
@ary (1, 2, sort 4 ,3)
print @ary; #_ecrit 1234
# Les expressions suivantes exécutent la sortie avant le print.
print ($foo, exit);
print $foo, exit;
# Les expressions exécutent d'abord le print.
(print $foo), exit;
print ($foo), exit;
```

5.3 L'opérateur flèche

Comme en C ou C++, -> est un opérateur de déréférence ou de référence. Si la partie à droite est soit […] ou {…} alors la partie à gauche doit être une référence à un tableau ou un hash.

5.4 Opérateur d'incrémentation et de décrémentation

"++" et "--" fonctionnent comme en C. S'ils sont placés avant la variable, l'incrémentation ou la décrémentation à lieu avant le renvoie de la variable. S'ils sont placés après la variable alors l'incrémentation ou la décrémentation a lieu après le renvoie de la valeur.

5.5 Exponentielle

L'opérateur binaire ** est l'exponentiel. Remarquez que -2**4 vaut -(2**4) et non pas (-2)**4, conformément aux règles de priorité des opérateurs.

5.6 Symbole unaire

! réalise la négation logique.
\ crée une référence sur ce qui le suit (scalaire, tableau, hash ou fonction).

5.7 Opérateur de lien

=~ Cet opérateur binaire lie une expression à un opérateur de comparaison. Certaines opération de recherche, de modification ou de translation, lorsqu'aucune variable n'est spécifiée via l'opérateur =~, recherchent et modifient par défaut la variable $_. Cet opérateur permet donc de faire en sorte que la variable recherchée soit celle à gauche de l'opérateur.

Exemple :
```
$a='moi';
$a=~s/m/t/g; # <=>$a=toi;
# Equivaut à
$_='moi';
~s/m/t/g; #équivalant à ce qui précède.
```

!~ est similaire à l'opérateur précédent mais retourne la valeur négative dans un sens logique.

5.8 Opérateur de multiplication

***** binaire, multiplie deux nombres.
/ binaire, divise deux nombres.
% binaire, calcule le modulo de deux nombres.
x opérateur binaire de répétition.
Exemple :
```
print -' x 80 # écrit 80 fois -
@tab=(1) x @tab  # équivaut à @tab=(1,1,: : : )
```

5.9 Opérateur additif

+ binaire, renvoie la somme de deux nombres.
- binaire, renvoie la différence de deux nombres.
. binaire, renvoie la concaténation de deux nombres.

5.10 Opérateur de décalage

<< opérateur binaire renvoie la valeur de son argument de gauche décalé à gauche du nombre de bits spécifié par l'argument de droite.
>> opérateur opposé au précédent.

5.11 Fonction unaire

Si un opérateur LIST (ex: print(),etc) ou un opérateur unaire (chdir(),etc) est suivi par une parenthèse, l'opérateur et les arguments entre parenthèses ont la plus grande priorité.
Exemple : chdir $foo || die; # (chdir $foo) || die

par contre en raison de la priorité de * sur les fonctions unaires, on a :
chdir $foo * 20; # chdir ($foo*20);

5.12 Opérateurs de comparaison

Il y a deux catégories d'opérateurs de comparaison. Ceux utilisés pour le contexte numérique et ceux pour le contexte alphanumérique.

Sens	Numérique	Alphanumérique
Strictement supérieur	>	gt
Strictement inférieur	<	lt
Inférieur ou égal	<=	le
Supérieur ou égal	>=	ge

5.13 Opérateur d'égalité

Sens	Numérique	Alphanumérique
Egalité	==	eq
Différence	!=	ne
Comparaison renvoie -1 si < renvoie 0 si = renvoie +1 si >	<=>	cmp

5.14 Opérateur logique bit à bit

& binaire, renvoie le résultat d'un ET logique opéré bit à bit.
| binaire, renvoie le résultat d'un OR logique opéré bit à bit.
^ binaire, renvoie le résultat d'un XOR logique…

5.15 Opérateur logique

&& binaire, réalise le ET logique entre deux opérandes évaluées Vraie ou Fausse à l'aide des opérateurs de comparaisons ou d'égalité.
|| binaire, OU logique: : :

5.16 Opérateur d'étendue

.. binaire. Dans un contexte de liste, il retourne un tableau de valeurs contenant les valeurs en partant de l'argument de gauche vers l'argument de droite.
Exemple : for (1..10){…} #fait une boucle de 1 à 10
@alphabet=('A'..'Z','a'..'z'); #définit un tableau contenant l'alphabet.
?: Ternaire, Si l'argument précédent ? est vrai alors l'argument avant : est retourné, sinon c'est l'argument suivant : qui est renvoyé.

```
$c =( $a_or_b ? $a : $b);
        équivaut à:
si ($a_or_b est vrai) alors $c=$a sinon $c=$b;
```

5.17 Guillemets

Les délimiteurs en Perl fonctionnent comme des opérateurs qui réalisent des interprétations ou des comparaisons de chaîne(pattern matching). Ces opérateurs permettent entre autres de délimiter des chaînes de caractères ou des commandes

Délimiteur			
Abréviation	Format Normal	Chose délimitée	Interprétation
' '	q{}	Littéral	Non
" "	qq{}	Littéral	Oui
	qx{}	Commande	Oui
	qw{}	Liste de mot	Oui
/ /	m{}	Comparaison de chaîne	Oui
	s{}{}	Substitution	Oui
	tr{}{}	Translation	Non

Exemple :

```
$foo=q!Ce que je dis!;
# équivaut à:
$foo='Ce que je dis';
$foo=qq(Bouh!); # <=> $foo="Bouh!";
$foo=qxfdateg; # <=> $foo=`date`;
$foo=qw("Bonjour Vous");
# équivaut à découper, à chaque espace, la chaîne de caractères
en liste de mots
```

$,@ sont interprétés pour les constructions qui le font, ainsi que :

Séquence	Signifiaction
\t	Tabulation
\n	Nouvelle ligne
\r	Retour
\l	Mettre en minuscule le caractère suivant
\u	Mettre en majuscule le caractère suivant
\L … \E	Bloc à mettre en minuscule
\U … \E	Bloc à mettre en majuscule
\b	Backspace
\a	Alarme
…	…

Exemple :

```
print "$toto \n";
# écrit le contenu de la variable $toto avec retour à la ligne
print '$toto \n'; # écrit $toto \n
tr /$toto/$tata/ # les variables ne sont pas interprétées
```

5.18 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 filehandle.
Une liste complète de ces opérateurs de tests est disponible dans la section perlfunc du manuel [2]. Le tableau suivant 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	Age du fichier en jours à partir de la date d'exécution du script
-s	Taille du fichier

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;
```

Chapitre 6

6 Structure des programmes

6.1 Structure d'un programme

Tout d'abord vous devez savoir qu'en règle générale, un programme Perl à la structure suivante:

- Appel d'un interpréteur Perl.
- Appel des différents modules nécessaires.
- Définition, éventuellement, de fonctions.
- Définition des variables globales. Leur place importe peu, mais attention à la redondance.
- Corps du programme principal.

6.2 Les procédures

On appellera par procédure aussi bien les fonctions que les procédures, qui sont des fonctions qui ne renvoient aucune valeur. Les procédures ont en général, la structure suivante :

```
sub NAME {
    Récupération des valeurs ou adresse transmise à la procédure.
    Définition des variables locales.
    Corps de la procédure.
    Renvoie de valeurs ou adresses (si besoin).
}
```

La déclaration, d'une procédure peut se faire n'importe où à l'intérieur de votre programme et a priori dans un module.

6.3 Transmission des arguments

On récupère au sein d'une procédure les valeurs passées grâce au symbole @ qui est celui du tableau par défaut. Il contient la liste des scalaires passés à la procédure.

6.3.1 Transmission des arguments par valeur

Le passage par valeur est le plus simple à réaliser. Il suffit pour cela de faire suivre l'appel de la fonction par une liste de variable.
Exemple :

```
&NAME ($a,$b,@c);Exploitable.
&NAME ($a,@tab1,@tab2);Inexploitable dans la variable de transmission @ où commencera tab1 ou tab2???
```

En effet @_=($a,$b,$tab1[0],..,$tab2[0]).
Le deuxième exemple est inexploitable. Les arguments passés lors de l'appel sont récupérés dans la variable @_ comme une liste de scalaires. On doit donc faire autrement: la transmission des arguments par adresse.

6.3.2 Transmission des arguments par adresse

La transmission d'argument par adresse se fait par le passage d'un pointeur sur la variable.
Exemple :

```
$tab1=\@tableau1;
$tab2=\@tableau2;
&NAME ($a,$tab1,$tab2);
```

6.4 Les variables locales

On peut définir des variables locales à la fonction grâce aux mots clés local() ou my(). my doit être utilisé de préférence à local, car my donne à la variable une étendue lexicale, par exemple, la variable ne sera pas visible des routines appelées ensuite. Ce n'est pas clair ? Un exemple aidera sûrement :

```
# On déclare 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
```

6.5 Fonction ou procédure

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) );
        # _equivalent _ a (notez le return implicite)
        # if ($n == 1)
        # {
        #       1;
        # }
        # else
        # {
        #       $n * &factorielle($n - 1);
        # }
```

6.6 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 appelée avec une liste vide en argument.
&fonction2(); # Idem.
&fonction2;
# fonction2 est appelée avec la même liste d'arguments
# que la fonction d'où l'appel est fait. Attention...
```

6.7 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, par exemple 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 était utilisée en particulier pour modifier le chemin de recherche des fichiers à inclure :

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

6.8 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).

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, etc.
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
```

6.9 Les modules

6.9.1 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;                # importnames from Cwd::
$here = getcwd();

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

6.9.2 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 7

7 Les entrées-sorties

7.1 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.

7.2 Interaction

Voilà 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 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>);
```

7.3 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) passée 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 suivant :

Caractère	Mode
Aucun	Lecture
<	Lecture
>	Ecriture
>>	Ajout
+<	Lecture/écriture
|	Pipe

Un nom de fichier particulier est à signaler : "-". Ouvrir - est équivalent à ouvrir STDIN et ouvrir >- revient 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. Dans ce cas, la valeur retournée par open est le pid du processus lancé.

7.4 La lecture

Pour lire sur un filehandle 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);
```

Pour lire un seul caractère, on peut utiliser la fonction :

```
getc(FILEHANDLE);
```

7.5 L'écriture

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

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

Il faut noter qu'il n'y a pas de virgule entre le nom du filehandle 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;
```

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));
```

7.6 La fermeture

Elle est effectuée par la fonction close, à laquelle on fournit en paramètre le filehandle à fermer :

```
close(FILE);
```

7.7 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 filehandle comme descripteur courant grâce à la commande select. On utilise communément cette syntaxe :

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

Chapitre 8

8 Les expressions régulières

Les expressions régulières sont une des caractéristiques de Perl qui rendent ce langage particulièrement adapté au traitement des fichiers texte. Une expression régulière 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, d'en extraire des informations ou bien d'y effectuer des substitutions.

8.1 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 amélioration 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 =~ :

```
$variable =~ /regexp/;
```

8.1.1 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 (expression régulière) du type /a[bc]d/ correspondra aux chaînes abd et acd. [] permet d'énumérer une classe de caractères.

L'ensemble des caractères composant la classe peut être précisée par énumération (comme précédemment) ou bien en précisant un intervalle comme par exemple :

```
/[a-z]/
```

correspondra à tous les caractères compris entre a et z.

On peut également prendre le complémentaire de cet ensemble, en le précédant d'un ^. Donc /a[^bc]d/ correspondra à toutes les chaînes du type a.d, sauf abd et acd.

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, egg dans l'exemple, et fixera la variable $1 à cette valeur.

8.1.2 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 peut-être n'importe quel caractère
\s	Espace
\S	Non-espace
\w	Lettre (caractères alphanumériques et _)
\W	Non-lettre
\d	Chiffre
\D	Non-chiffre

Il existe aussi une structure conditionnelle : ?=. En voici un exemple :

```
/nw+(?=nt)/
```

Cette expression matche les mots (\w+) si (?=) ils sont suivis par une tabulation \t.

8.1.3 Les quantificateurs

Différents quantificateurs s'appliquent au caractères et métacaractères :

*	Matche 0 ou plusieurs fois
+	Matche 1 ou plusieurs fois
?	Matche 0 ou 1 fois
{n}	Matche exactement n fois
{n,}	Matche n fois ou plus
{m,n}	Matche 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.

8.1.4 Les quantificateurs non-gourmands

Une nouveauté intéressante introduite dans la version 5 de Perl est la possibilité d'obtenir des quantificateurs non gourmands, par exemple 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";
```

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">
```

8.2 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, par exemple à un motif particulier.

8.2.1 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 (matche toutes les occurrences, 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). 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 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.

8.2.2 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 une 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.

8.3 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'effectuer un remplacement par le résultat d'une opération.
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;
```

D'autres exemples :

```
$string='\l\U\T\2';  #Chaîne non interprétée
($tot=$string)=~s/\\/\_/g;
# On vient de substituer les backslash par des underscore.
print $tot #écrit : _l_U_T_2

# On remplace les voyelles par une étoile
($tot=$string)=~ s/(a|e|i|o|u|y)/*/g;

$_='abc123def456';      # Initialise la variable par défaut.
# On substitue le premier ensemble de chiffre (option d+) en
# l'évaluant (option e) et en multipliant chacun des chiffres par 2
s/\d+/$&*2/e;           # Fournit abc246def456,
# Ici on n'évalue pas le résultat de la recherche. On recherche les
# chaînes de chiffres dans toutes la chaîne de caractères.
s/\d+/$&*2/g;           # Fournit abc123*2def456*2
# Recherche globale avec évaluation
s/\d+/$&*2/ge;          # Fournit abc246def912
```

8.4 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;
```

Autres exemples :

```
# les a sont remplacés par z, b par y !!
tr /ab/zy/;

# compte les étoiles dans $sky.
$cnt= $sky =_tr/*/*/;

 #Remplace les caractèeres non alphas par des espaces.
tr/a-zA-Z/ /c;
```

8.5 Un exemple : Inversion de deux mots

Pour montrer la puissance du Perl, mais aussi sa complexité, voici un petit exemple d'inversion des deux derniers mots d'une chaînes de caractères contenue dans une variable et séparer par un double espace.

Le principe est de repérer la fin de chaîne par le symbole $, de repérer le double espace, qui ici sépare les 2 derniers mots de la chaînes.

On a pour solution:

- \s{2,2}

On cherche un espace (\s) au moins 2 fois et au plus 2 fois ({2,2}).

- \s\s

On cherche 2 espaces.

On cherche aussi à isoler les chaînes de caractères. On a :

- \w+

On cherche un caractère une fois ou plus. On cherche donc un mot car les espaces ne sont pas considérés comme des caractères.

- [^]*

On cherche tous les caractères autres que les espaces.

```
$_ ='aa bb  cc';
# Instructions qui fonctionnent.
s/([^ ]*)\s{2,2}([^ ]*)$/$2 $1/;
s/(\w+)\s\s([^ ]*)$/$2 $1/;
s/([^ ]*)  ([^ ]*)$/$2 $1/;
# Instructions qui ne fonctionnent pas
# Manque le double espace.
s/([^ ]*)([^ ]*)$/$2 $1/;
# La première recherche ne s'applique qu'à un caractère.
s/([^ ])  *([^ ]*)$/$2 $1/;
```

Chapitre 9

9 Les répertoires et les fichiers

9.1 Accès aux répertoires

9.1.1 Parcourir l'arborescence

En programmation système, la fonction système chdir change le répertoire courant; Perl utilise également ce nom.

La fonction chdir (change directory) de Perl n'accepte qu'un seul argument - une expression dont l'évaluation produit un nom de répertoire. Comme la plupart des autres appels système, chdir renvoie true lorsque vous avez réussi à passer dans le répertoire désiré, et false en cas d'échec. Voici un exemple :

```
Chdir("/etc") || die "Impossible d'aller dans /etc ($!)";
```

Les parenthèses sont optionnelles, vous pouvez par conséquent procéder comme ceci :

```
If (chdir $ouvatons) {
    # changement de repertoire
} else {
    # le changement à échoué
}
```

Il est difficile de savoir où l'on est sans lancer une commande pwd (print working directory). Nous verrons dans le chapitre suivant, Gestion des processus, comment lancer des commandes.

Chaque processus possède son propre répertoire courant. Au lancement du processus, celui-ci hérite du répertoire en cours de son père, mais la liaison s'arrête là.

La fonction chdir sans paramètre conduit au répertoire "HOME".

9.1.2 Globaliser

Le shell prend comme argument de ligne de commande un seul astérisque (*) et le transforme en liste de tous les noms de fichiers du répertoire courant. Ainsi, en indiquant rm *, vous supprimer tous les fichiers du répertoire courant. De même, en indiquant [a-m]*.c, vous obtenez une liste de tous les noms de fichiers du répertoire courant qui commencent par l'une des lettres de la première moitié de l'alphabet et se terminent par .c.

L'extension d'arguments tels que * ou /etc/host* dans la liste des noms de fichiers à retrouver s'appelle la globalisation. Perl supporte la globalisation au travers d'un mécanisme très simple : placez simplement le motif de globalisation entre chevrons, ou utilisez la fonction glob, plus mnémonique.

```
@a = </etc/host*>;
@a = glob("/etc/host*");
```

Dans un contexte de liste, comme ici, glob renvoie une liste de tous les noms correspondants au motif. Dans un contexte scalaire, le prochain nom correspondant à l'expression est renvoyé, ou undef s'il n'y a plus de correspondance.

Les motifs multiples sont autorisés dans l'argument de globalisation de fichier; les listes sont établies individuellement puis concaténées :

```
@multi = <truc* bidule*>
```

L'argument de glob est interpolé avant extension. Vous pouvez donc vous servir des variables Perl pour sélectionner des fichiers en fonction d'une chaîne construite en cours d'exécution :

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

Il existe cependant une exception à cette règle : le motif <$var> doit être écrit sous la forme <${var}> car <$var> lit une ligne du handle de fichier dont le nom se trouve dans la variable $var.

9.1.3 Handle de répertoires

Un handle de répertoires est un nom provenant encore d'un autre espace de noms, et auquel s'appliquent toutes les précautions et les recommandations qui concernent les handles de fichiers. Le handle de fichier TRUC et le handle de répertoire TRUC n'ont aucun rapport.

Le handle de répertoire représente une connexion à un répertoire particulier. Au lieu de lire des données, il permet de lire une liste de noms de fichiers. Il est toujours ouvert en lecture seule; on ne peut pas l'utiliser pour modifier le nom ou supprimer un fichier ou un répertoire.

9.1.4 Ouvrir et fermer un handle de répertoire

La fonction opendir fonctionne comme l'appel du même nom de la bibliothèque C et C++. On lui fournit le nom d'un nouvel handel de répertoire et une valeur de type chaîne indiquant le nom du répertoire à ouvrir. La valeur de retour est true si l'ouverture a fonctionné et false sinon. Voici un exemple :

```
opendir(ETC, "/etc") || die "Impossible d'ouvrir /etc : $!";
```

Pour fermer un handle de répertoire, il suffit d'utiliser la commande closedir, de la même façon que close, comme ceci :

```
closedir(ETC);
```

Comme close, closedir n'est pas indispensable, car tous les handles de répertoires sont automatiquement fermés avant d'être réouverts, ou à la fin du programme.

9.1.5 Lire un handle de répertoire

Dès que nous disposons d'un handle de répertoire ouvert, nous pouvons lire la liste des noms avec readdir, qui ne prend qu'un seul paramètre : le handle du répertoire. Chaque appel à readdir dans un contexte scalaire renvoie le nom de fichier suivant. S'il n'y a plus de fichier, readdir renvoie undef. L'appel à readdir dans un contexte de liste renvoie une liste de tous les noms restants. Voici un exemple de listage de tous les noms du répertoire /etc :

```
opendir(ETC,"/etc") || die "pas d'etc ??? : $!";
while($nom = readdir(ETC)) { #contexte scalaire, un par boucle
print "$nom\n";}
```

9.2 Manipulation des réperoires et fichiers

Nous allons voir maintenant comment manipuler les fichiers eux-mêmes et non les données qu'ils contiennent.

9.2.1 Supprimer un fichier

Vous avez appris comment créer un fichier Perl en l'ouvrant en sortie par un handle de fichier. A présent, voyons comment supprimer un fichier.

La fonction Perl unlink supprime un nom d'un fichier (qui pourrait posséder d'autres noms). Lorsque le dernier nom d'un fichier est supprimé, et qu'aucun processus ne l'a ouvert, le fichier lui-même est supprimé. Cela correspond exactement à ce que fait la commande UNIX rm.

Comme dans la plupart des cas, un fichier ne possède qu'un seul nom, vous pouvez considérez qu'en supprimant un nom, vous supprimer le fichier. Par conséquent, voici comment effacer un fichier nommé truc puis un fichier indiqué durant l'exécution du programme :

```
unlink("truc"); #dit au revoir à truc !!!
print "Quel fichier doit-on supprimer ? ";
chomp ($nom = <STDIN>);
unlink ($nom);
```

La fonction unlink peut également prendre une liste de noms à supprimer :

```
unlink ("truc", "bidule");
unlink <*.o>;
```

La valeur de retour de unlink correspond au nombre de fichier supprimés avec succès.

Si aucun argument n'est spécifié à unlink, la variable $_ est là encore utilisée par défaut.

9.2.2 Renommer un fichier

En Perl, la command rename permet permet de changer le nom d'un fichier :

```
rename ("truc", "bidule")
    || die "Impossible de renomer truc en bidule : $!";
```

Comme la plupart des autres fonctions, rename renvoie une valeur true en cas de réussite et false sinon.

9.2.3 Les liens

Parfois, il est utile d'avoir deux, trois, ou dix noms pour un fichier. Cette action qui consiste à créer des noms de remplacement pour un fichier s'appelle la liaison. Les deux formes principales de liaison sont les liens solides et les liens symboliques.

9.2.3.1 Liens solides et liens symboliques

Un lien solide vers un fichier est indiscernable de ce dernier. Le système d'exploitation conserve le nombre de liens solides qui font référence au fichier à un moment donné. A sa création, un fichier est créer avec un seul lien. Chaque nouveau lien solide incrémente le compteur alors que chaque suppression le décrémente. Lorsque le dernier lien disparaît, et que le fichier est fermé, celui-ci disparaît.

Un lien symbolique (symlink) est une sorte particulière de fichier qui contient un nom de chemin. A l'ouverture d'un tel fichier, le système d'exploitation considère son contenu comme caractères de remplacement du nom de fichier qui seront passés au noyau pour une nouvelle tentative d'ouverture.

Un lien solide évite la perte du contenu d'un fichier (car il est considéré comme étant l'un des noms du fichier). Un symlink n'assure pas ce service; il reste indépendant du fichier qu'il référence. Si ce dernier disparaît, le symlink "pendouille", c'est-à-dire pointe sur un nom invalide. Encore un détail : on peut créer un symlink vers un répertoire, mais pas un lien solide.

9.2.3.2 Créer des liens solides et symboliques avec Perl

Pour créer un lien solide en Perl, on utilise la commande link :

```
link ("truc", "bidule") || die "Ne peut lier bidule à truc : $!";
```

crée un lien solide du fichier truc (qui doit exister) vers bidule. La commande renvoie true si la liaison réussi.

Pour créer un lien symbolique, on utilise la commande symlink :

```
symlink ("truc", "bidule") ||
    die "Ne peut lier symboliquement bidule à truc : $!";
```

Cette exemple permet de créer un lien symbolique nommé "bidule" vers un fichier nommé "truc". Remarquez que le fichier truc n'a pas besoin d'exister, que ce soit à présent ou à l'avenir. Dans ce cas, une référence à "bidule" renverra quelque chose ressemblant vaguement à "No such file or directory".

9.2.4 Créer et supprimer des répertoires

En Perl, on utilise la commande mkdir pour créer des répertoire. Cette commande prend un nom pour un nouveau répertoire et un mode définissant ses autorisations. Le mode est indiqué sous forme de nombre interprété par rapport à un format d'autorisations interne. Voici un exemple de la façon de créer un répertoire nommé machin :

```
mkdir ("machin", 0777) || die "Impossible de créer machin : $!";
```

Pour supprimer un répertoire, on utilise la commande rmdir. Voici comment supprimer machin :

```
rmdir ("machin") || die "Impossible de supprimer machin : $!";
```

9.2.5 Modifier les autorisations

Les autorisations sur un fichier ou un répertoire définissent qui peut faire quoi de ce fichier ou répertoire. Comme sous UNIX, Perl modifie les autorisations grâce à la commande chmod. Cet commande prend un mode sous forme numérique et une liste de noms de fichiers, et essaie d'attribuer ce mode à tous ces fichiers. Par exemple, pour permettre à tout le monde la lecture et l'écriture des fichiers truc et bidule, faites ceci :

```
chmod (0666, "truc", "bidule");
```

La valeur de retour de chmod est le nombre de fichiers traités avec succès; elle est donc sembleble à celle d'unlink.

9.2.6 Modifier la propriété

Chaque fichier du système de fichiers possède un propriétaire et un groupe qui sont établis lors de la création du fichier.

Pour modifier ses propriétés, on utilise la commande chmod. Cette commande prend un numéro ID d'utilisateur (UID) et un numéro ID de groupe (GID), ainsi qu'une liste de noms de fichiers. Une valeur de retour différente de zéros est donc égale au nombre de fichiers modifiés avec succès.

Remarquez que l'on change simultanément le propriétaire et le groupe. Si vous ne voulez modifier qu'un seul des deux, utilisez un ID égal à -1.

Remarquez aussi qu'il faut utilisez les UID et GID numériques, et non les noms symboliques correspondants. Par exemple, si truc possède l'UID 1234 et si le groupe de truc possède le GID 35, la commande suivante fait appartenir les fichier bidule et machin à truc et son groupe :

```
chown (1234, 35, "bidule", "machin");
```

9.2.7 Modifier les repères de temps

A chaque fichier, sont associés trois repère de temps : la date de dernier accès, la date de dernière modification, et la date de dernière modification de l'inode. Les deux premiers repères peuvent être initialisés à une valeur quelconque par la fonction utime. Le fait d'initialiser ces deux valeurs initialise automatiquement la troisième à l'heure courante; il n'existe donc pas de moyen d'initialiser soi-même la troisième valeur.

Les valeurs sont mesurés en temps interne, c'est-à-dire un nombre entier de secondes écoulées depuis le premier janvier 1970 à minuit GMT. Cela est représenté par entier non signé de 32 bits.

La fonction utime se comporte comme chmod et unlink. Elle prend une liste de noms de fichiers et renvoie le nombre de fichiers modifiés. Voici comment faire croire que les fichiers truc et bidule ont été recemment modifiés :

```
$deracces = $dermodif = 700_000_000;
utime($deracces, $dermodif, "truc", "bidule");
```

ou même, dans le futur :

```
$quand = time() + 20*60;        #20 minutes à partir de maintenant
utime($quand, $quand, "fic_futur");
```

Chapitre 10

10 Gestion de processus

10.1 Utilisez system et exec

Lorsque vous demandez au shell d'exécuter une ligne de commande, celui-ci crée en général un nouveau processus, qui devient un fils du shell, s'exécutant de manière indépendante, bien que toujours en coordination avec lui.

De la même façon, un programme Perl peut lancer un nouveau processus de diverses manières.

La plus simple consiste à utiliser la fonction system. Sous sa forme la plus élémentaire, cette fonction passe à un shell /bin/sh flambant neuf une chaîne devant être exécutée comme une commande. Une fois cette commande terminée, la fonction system renvoie la valeur de sortie (en principe 0 si tout s'est bien passé). Voici un exemple :

```
system("date");
```

Vers quel endroit est dirigée la sortie de la commande ? D'où vient l'entrée, dans le cas où la commande a besoin d'une entrée ? Voilà donc ce qui distingue les diverses formes de création de processus.

Dans le cas de la fonction system, les trois fichiers standard sont hérités du processus Perl. Puisque vous lancez un shell, vous avez la possibilité de modifier la sortie standard en utilisant les redirections d'E/S. Voici un exemple :

```
system("date >maintenant") && die "Impossible de créer
maintenant";
```

Un autre exemple :

```
#Détermine le nom du fichier
$qui_quand = "qui_quand_". ++$i.".txt";
system "(date; who) >^qui_quand &";
```

Dans cet exemple, la chaîne entre apostrophe est interpolée; $qui_quand est donc remplacée par sa valeur (par Perl et non par le shell). Pour faire référence à une variable shell nommée $qui_quand, il aurait fallu faire précéder le signe $ par un antislash (\), ou utiliser une chaîne entre apostrophes simples. On remarque aussi que cette commande UNIX est exécutée en arrière plan puisque la ligne de commande se termine par le signe &.

Outre les handles de fichiers standard, un processus fils hérite de nombreuses choses de son père. Cela comprend l'umask en cours, le répertoire courant, et naturellement, l'ID utilisateur. De plus, le fils hérite de toutes les variables d'environnement.

Enfin, il existe plusieurs manière de transmettre des paramètres à une commande shell :

```
system "grep ´truc bidule´ machin";       # Utilise le shell
system "grep", "truc bidule","machin";    # Evite le shell
```

Ces deux commandes sont identiques. Le fait de fournir une liste à system plutôt qu'une simple chaîne permet également d'économiser un processus shell; procédez par conséquent ainsi à chaque fois que vous le pouvez.

10.2 Apostrophes inverses

Il est également possible de lancer un processus en mettant une ligne de commande entre apostrophes inverses. Comme le shell, cela déclenche une commande et attend son achèvement, en interceptant la sortie standard :

```
# Obtient les sorties texte et date
$maintenant = "Nous sommes le ".`date`;
```

La variable $maintenant contient à présent le texte "Nous sommes le " suivit du résultat de la commande date, y compris le "nouvelle ligne" de terminaison.

En utilisant la commande entre apostrophes inverses dans un contexte de liste plutôt que dans un contexte scalaire, vous obtenez une liste de chaînes, chacune étant une ligne provenant de la sortie de la commande.

L'entrée standard et la sortie d'erreur standard de la commande entre apostrophes inverses sont hérités du processus Perl, ce qui signifie que vous n'obtenez comme valeur de la chaîne entre apostrophes inverses, que la sortie standard des commandes.

Il est courant d'insérer la sortie d'erreur standard dans la sortie standard, en utilsant la construction shell 2>&1, ainsi :

```
die "rm a parlé !!!" if `rm truc 2>&1`;
```

affiche le message si rm dit quelque chose, que ce soit sur la sortie standard ou la sortie d'erreur standard.

10.3 Utiliser les processus en tant que handles de fichiers

On peut également créer un processus qui ressemble à un handle de fichier, lequel capture la sortie d'un processus ou lui fournit une entrée. Voici un exemple :

```
open (WHOPROC, "who|");       # ouvre who en lecture
```

Remarquez le pipe à droite de who. Il indique à Perl que ce open s'effectue sur une commande et non un fichier. De plus, comme il se situe à droite, il indique à Perl qu'il s'agit d'une ouverture en lecture, donc la sortie standard de la commande est capturée.

Voici une manière de lire les données de la commande who et de les écrire dans un tableau :

```
@who_a_dit = <WHOPROC>;
```

De même, pour invoquer une commande attendant une entrée, on peut utiliser un handle de fichier-commande en écriture en plaçant le pipe à gauche de la commande, comme ceci :

```
open (LPR, "|lpr -Plp01");
print LPR @a_imprimer;
close (LPR);
```

Dans ce cas, après avoir ouvert un handle nommé LPR, nous y écrivons des données puis le fermons. Le fait d'ouvrir un processus avec un handle de fichier-processus permet à la commande de s'exécuter parallèlement au programme Perl. En indiquant close pour un handle de processus, nous obligeons le programme Perl à attendre que le processus se termine. Si nous ne fermons pas le handle, le processus peut se poursuivre même au-delà de l'exécution du programme Perl.

Le fait d'ouvrir un processus en écriture entraîne que l'entrée standard de la commande provient du handle, et le processus partage la sortie standard et l'erreur standard avec Perl. Il est possible d'utiliser la redirection d'E/S :

```
open (LPR, "|lpr -Plp01 >/dev/null 2>&1);
```

Il n'est pas nécessaire de n'ouvrir qu'une commande à la fois. Il est possible d'ouvrir un pipeline entier. La ligne suivante en est un exemple :

```
open (WHOPR, "ls -r | tail |");
```

On démarre un processus ls, lequel dirige sa sortie dans un processus du tube tail, lequel envoie enfin sa sortie vers le handle WHOPR.

10.4 Utiliser fork

Une autre manière de créer un processus supplémentaire consiste à cloner le processus Perl en cours à l'aide de la primitive UNIX fork. La fonction fork crée un clone du processus en cours, qui partage le même code exécutable, les variables, et même les fichiers ouverts; Pour faire le distinguo entre les deux processus, la valeur de retour de fork est nulle pour le fils, et non-nulle pour le père (ou undef en cas d'échec). La valeur reçue par le père se trouve être l'ID du processus fils. Il est possible de tester la valeur de retour et d'agir en conséquence :

```
if (!defined($PID_fils = fork()) {
    die "fork a échoué : $!";
} elsif ($PID_fils) {
    # Je suis le père
} else {
    # Je suis le fils
}
```

Afin de mieux utiliser ce clone, nous allons voir les fonctions wait, exit, et exec.

La fonction exec, qui ressemble à la fonction system, sauf qu'au lieu de lancé un nouveau processus, elle remplace le processus en cours. Par exemple :

```
exec "date";
```

remplace le programme Perl en cours par la commande date, la sortie de date étant redirigée sur la sortie standard du programme Perl.

En fait, on peut considérer que la fonction system correspond à un fork suivi d'un exec :

```
# METHODE 1
system("date");

#METHODE 2
unless(fork) {
    # fork a renvoyé zéro, je suis donc le fils, et j'exec
    exec("date");      # le processus fils est remplacé date
}
```

Il n'est cependant pas aussi facile d'utiliser fork et exec de cette façon, car la commande date et le processus père progresse en même temps, mélangeant le cas échéant leurs sorties. Nous avons besoin d'une possibilité de demander au père d'attendre la fin du processus fils. C'est précisément le rôle de la fonction wait; elle attend qu'un fils se termine. La fonction waitpid est plus discriminatoire : elle attend qu'un processus fils particulier se termine.

Enfin, la fonction exit entraîne la sortie immédiate du processus Perl en cours.

10.5 Envoyer et recevoir des signaux

On peut établir une communication entre processus en envoyant et recevant des signaux. Les signaux sont numérotés et certains possèdent

une signification préétablie, et sont envoyés automatiquement à un processus sous certaines conditions.

La réponse à un signal s'appelle action du signal. Les signaux prédéfinis possèdent des actions par défaut. Presque tous ces signaux peuvent être écrasés, ignorés ou interceptés.

Jusqu'ici, rien de bien différent. Voyons maintenant les spécificité à Perl. Avec Perl, il est nécessaire de connaître les noms des signaux pour créer un gestionnaire de signaux.

Les noms des signaux sont définis dans la page de manuel signal, et en général également dans le fichier C d'inclusion /usr/include/sys/signal.h. Ils commencent habituellement par SIG, comme SIGINT, SIGQUIT, et SIGKILL.

Pour déclarer le sous-programme mon_capteur_de_sigint() comme étant le gestionnaire de signal s'occupant de SIGINT, nous introduisons une valeur dans le hachage %SIG. Nous initialisons également la valeur de la clé INT avec le nom du sous-programme qui interceptera le signal SIGINT, comme ceci :

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

Mais nous avons aussi besoin d'une définition de ce sous-programme. En voici une :

```
Sub mon_capteur_de_sigint {
    $vu_sigint = 1;    # lève le drapeau
}
```

Cet interpréteur de signal initialise une valeur globale puis sort immédiatement. La sortie de ce sous-programme fait reprendre l'exécution à l'endroit où elle avait été interrompue. Voilà une utilisation habituelle :

```
$vu_sigint = 0;                          # efface le drapeau
$SIG{'INT'} = 'mon_capteur_de_sigint';# enregistre l'intercepteur
foreach (@grand_tableau) {
    ……                                   # fait quelque chose
    ……                                   # en fait d'autres
    ……                                   # en fait encore d'autres
    if ($vu_sigint) {                    # interruption demandée ?
        ……                               # ici, un peu de ménage
        last;
    }
}
$SIG{'INT'} = 'DEFAULT';       # restaure l'exécution par défaut
```

L'astuce consiste ici à vérifier le drapeau à des endroits appropriés, ainsi que pour sortir prématurément de la boucle, et effectuer un peu de nettoyage.

Dans la dernière instruction de ce code, le fait d'initialiser l'action à DEFAULT restaure l'action par défaut pour le signal en question. IGNORE est une autre valeur particulière bien utile car elle indique d'ignorer le signal.

Chapitre 11

11 Notions supplémentaires

11.1 Les formats

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

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

11.1.1 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 suivant résume les différents types de champs d'interpolation.

@<<<<<	Champ aligné à gauche
@|||||	Champ centré
@>>>>>	Champ aligné à droite
@#####	Champ numérique (aligné à droite)
@#.###	Champ numérique avec décimale
@*	Champ multiligne

11.1.2 Utilisation

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

Voilà a 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_name STDOUT "autre_format"
```

11.2 Débuggage

Un jour ou l'autre vous aurez des problème avec vos programmes. Il sera temps pour vous de procéder au débuggage de votre application. En Perl, il y a deux façons de procéder.

- L'affreuse : des "print expression" partout comme en BASIC (parfois plus rapide),
- L'élégante : utiliser le débuggeur Perl.(waouh!!)

11.2.1 Fonctionnement

Voyons comment fonctionne le débuggeur Il est appelé grâce à l'option de compilation -d lors de l'appel de l'interpréteur (#!/usr/bin/perl -d). De plus il vous faut inclure dans le corps du programme à débugger la bibliothèque Perl5db.pl. Attention pour débugger il ne doit pas y avoir d'erreur de syntaxe.

En effet, l'interpréteur lance le Perl, qui vérifie la syntaxe et ensuite fait tourner le programme par lui même au travers du debuggeur. Lorsque qu'il est lancé le débuggeur affiche une série de données, puis il écrit la première instruction Perl du programme et enfin fait apparaître un prompt.

Grâce aux commandes ci-dessous, vous exécuterez un pas à pas de votre programme.

11.2.2 Commandes

Voici quelques commandes que ce prompt aimera:

q : quitter le débuggage.

command : exécute la commande Perl (ex: $a=1;)

n : exécute la commande écrite au dessus du prompt DB

p expr : écrit la valeur de expr

s : pas à pas (entre dans les procédures)

r : pour terminer le débuggage en terminant l'exécution

l : écrit les lignes du programme.

<CR> :Opérateur de répétition (répète la commande précédente n ou s)

c ligne : continue jusqu'à la ligne du programme spécifiée.

w ligne : écrit le contenu de la ligne du programme (permet de visualiser le contenu d'une procédure).

Voilà, vous en savez autant que moi sur le débuggage élégant.

11.3 Comment récupérer la sortie d'un programme

Il existe deux façons 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.

11.4 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. La méthode conseillée 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);
```

11.5 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;
}
```

Chapitre 12

12 Les fonctions Perl par catégorie

Ce chapitre est un résumé des fonctions les plus importantes (plus couramment utilisées) classées par catégorie.

12.1 Les fonction de manipulation de listes

Pour manipuler les listes, Perl propose plusieurs fonctions.

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)
```

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)
```

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.

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

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 octets.

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

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)
```

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);
```

12.2 Les fonctions sur les tableaux associatifs

keys

Syntaxe : keys %HASH

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

values

Syntaxe : values %HASH

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

each

Syntaxe : each %HASH

each retourne une liste _a 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.

delete

Syntaxe : delete $HASH{KEY}

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

12.3 Les fonctions de manipulation de chaînes

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

substr *Note*: elle peut ^etre utilis_e comme lvalue (ie a_ect_e).

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

index, rindex

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

D'autres fonctions sont également définies :

length
Donne la longueur de la chaîne passée en paramètre.

l'opérateur . (point)
Concaténation de deux chaînes.

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

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

```
$clair = <STDIN>;
$crypte = (getpwuid($<))[1];
$passwd = crypt($clair, $crypte);
```

lc, lcfirst, uc, ucfirst Opérations sur la casse de la chaîne.

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

12.4 Les fonctions d'Entrée-Sortie

close (FILEHANDLE)
Ferme un fichier. Retourne Vrai si la fermeture a été correcte.

die LIST
Ecrit la valeur de List sur la sortie d'erreur.
exemple. 8.3.1 open (@A,"<$Test") jj die "Erreur";

eof (FILEHANDLE)
Retourne 1 si la donnée suivante sur FILEHANDLE est la fin du fichier.

getc (FILEHANDLE)
Lit le caract_ere provenant du _chier FILEHANDLE ou de STDIN.

open (FILEHANDLE,EXPR)
Ouverture d'un fichier EXPR. si EXPR est précédé par :
< le fichier est en entrée (lecture).
> le fichier est ouvert en sortie (écriture).
>> mode ajout.

print FILEHANDLE LIST
Ecriture sur fichier.

print LIST
Ecriture sur la sortie standard (Ecran).

12.5 Les fonctions sur les répertoires et processus

chdir EXPR

La fonction chdir de Perl n'accepte qu'un seul argument - une expression dont l'évaluation produit un nom de répertoire. chdir renvoie true lorsque le changement de répertoire à réussi, et false en cas d'échec.

Exemple : chdir("/etc") || die "Impossible d'aller en /etc ($!)";

chmod EXPR

Permet de modifier les droits d'accès.

Exemple : chmod(0666,"truc","bidule");

chown LIST

Change le propriétaire et le groupe d'une liste de fichier

exemple. 8.4.1 chown $uid, $gid, @ary;

exec LIST

exécute et ne revient JAMAIS.

exemple. 8.4.2 exec echo 'Le programme est fini';

utilisez la commande system LIST pour revenir au programme.

12.6 Les fonctions de calcul mathématique

abs (VALUE) Retourne la valeur absolue de VALUE.

atan2 (Y,X) Retourne l'arctangente de $\frac{Y}{X}$ dans l'intervalle -π et π.

cos (EXPR) Retourne le cosinus.

defined (EXPR) Retourne vraie si EXPR à une valeur réelle.

exp (EXPR) Retourne l'exponentiel népérien de EXPR.

hex (EXPR) Retourne la valeur décimale de EXPR considéré comme un hexadécimal.

int (EXPR) Retourne la partie entière.

log (EXPR) Retourne le logarithme (base e) de EXPR.

oct (EXPR) Retourne la valeur décimale de EXPR considéré comme un octal.

rand (EXPR) Retourne un nombre compris entre 0 et EXPR. Si EXPR est omis alors rand renvoie un nombre entre 0 et 1

sin (EXPR) Retourne le sinus de EXPR.

sqrt (EXPR) racine carrée

12.7 Les autres fonctions

On retrouve généralement les mêmes qu'en C ou en shell.

Petit référentiel Perl

Je me propose, ici, de dresser une liste non exhaustive de commandes Perl.

Dans la liste suivante, les parenthèses sont utilisées pour définir les éventuels arguments, elles ne sont pas nécessaires mais il est vivement conseillé de les utiliser pour ne pas se risquer dans les pièges de priorité du langage.

Quand rien n'est spécifié, les fonctions sont supportées par les versions 4 et 5 du langage. Lorsque, la fonction n'est supportée que par la version 5, cela est signalé par le mot ***(Perl 5)***.

Les arguments indiqués entre crochet, signifient qu'ils sont optionnels.

/PATTERN/

recherche la chaîne *PATTERN*; voir m/PATTERN/

```
$_="chaine de caractères" ;
if ( /^c/) { print "la chaine commence par un c" ; }
```

?PATTERN?

recherche la première chaîne *PATTERN*; cette commande est utile plutôt pour tester la présence d'une chaîne de caractères dans un fichier. Cette commande sera supprimée dans les versions futures de Perl, il ne faut donc pas l'utiliser.

abs (VALEUR) *(Perl 5)*

donne la valeur absolue d'un nombre.

accept(NEWSOCKET,GENERICSOCKET) (UNIX)

cette subroutine est calquée sur la procédure *accept* d'Unix.

alarm(SECONDS) (UNIX)

envoie un signal SIGALARM au processus au bout d'un temps donné en seconde par le paramètre SECONDS
Retour : true en cas de succès, false sinon.

atan2(Y,X)

retourne l'arctangente de X Y dans le cercle -PI, PI

bind(SOCKET,NAME) (UNIX)

bind un socket comme l'appel système Unix du même nom
Retour : true en cas de succès, false sinon.

binmode(FILEHANDLE)

signale que le fichier sera lu en binaire (comme ftp). Dans ce mode aucune transformation du fichier n'est faite en cours de lecture.

bless (REF, [PACKAGE]) *(Perl 5)*

l'objet REF est connu comme objet de PACKAGE si celui-ci est donné, ou du PACKAGE courant.
Retour : la référence

caller([EXPR])

sans argument, retourne le contexte de la subroutine courante, avec une expression retourne le contexte et le niveau de l'expression en paramètre.
Exemple :

```
sub exemple {
($var1,$var2,$var3,$var4) = caller ($var4) ;
} ;
&exemple ;
print "$var1 , $var2 , $var3, $var4" ;
```

Produit le résultat suivant: main , c:\pmail\essai.pl , 4, main'exemple

chdir(EXPR)

chdir change le répertoire courant pour EXPR. Si EXPR est omis, chdir va dans le répertoire d'accueil (home).
Retour : true si le changement a pu avoir lieu, false sinon

chmod(acces, LIST) (*Unix*)

change les droits d'accès d'une liste de fichiers et les met à la valeur *acces* qui doit être numérique.
Retour : Le nombre de fichiers qui ont pu être changés de mode.
Exemple : chmod (777, ex1, ex2);

chomp([VARIABLE|LISTE]) *(Perl 5)*

est une version plus sécurisée de la fonction chop. Retire tous les caractères suivants le caractère $\ (qui est le séparateur de champs). Sans argument, chomp opère sur $_. Avec en argument une liste, l'opération est effectuée sur chaque élément de la liste.
Retour : le nombre de caractères supprimés.

chop([LIST])

élimine le dernier caractère de la liste. Ceci est très utilisé pour retirer la marque de nouvelle ligne d'une chaîne. Sans argument opère sur la variable $_
Retour : le caractère retiré

```
$chaine="exemple\n" ;
chop ( $chaine ); # $chaine perd son caractère \n
```

chown(uid, gid, LIST)(*Unix*)

donne à la liste de fichiers LIST les uid et gid voulus.
Retour: le nombre de fichiers dont l'*uid* et le *gid* ont été changés.

chr NOMBRE (*Perl 5*)

Retourne le caractère ASCII représenté par le nombre. (chr(65)="B")

chroot(FILENAME)(*Unix*)

Même commande que l'appel système du même nom.

close(FILEHANDLE)

ferme un fichier défini par son descripteur. (cf open). En cas de fermeture d'un PIPE, la fonction attend que le process soit terminé
Retour : TRUE si la fermeture s'est passé normalement.

closedir(DIRHANDLE)

ferme un répertoire ouvert par opendir.

connect(SOCKET,NAME) (*Unix*)

connecte le SOCKET à la librairie *NAME*

cos(EXPR)

calcule le cosinus d'une expression EXPR donné en radian

crypt (TEXTE,CLE) (*Perl 5*)

Chiffre une chaîne de caractères avec la clé. Pour mémoire, la clé sur les systèmes UNIX est obtenue par la séquence

```
$pwd = (getpwuid($<))[1];
$salt = substr($pwd, 0, 2);
```

defined(EXPR)

retourne un booléen indiquant si la valeur EXPR est une expression définie dans le programme

delete (EXPR) (*Perl 5*)

Supprime la valeur du tableau où elle est définie.

die(LIST)

quitte le programme en affichant la chaîne LIST sur le canal d'erreur.

do BLOCK

retourne la valeur de la dernière commande de la séquence de commandes indiquée par BLOCK
Dans une fin de boucle par exemple, exécute une fois de plus la dernière instruction de la boucle avant de sortir.

do SUBROUTINE (LIST)

exécute la subroutine (do est remplacé par &)
Retour : retourne le résultat de la dernière commande de la subroutine.

do EXPR

exécute le fichier EXPR comme un fichier perl

dump LABEL(*Unix*)

cause un core dump qui permet d'analyser le comportement du fichier en tant que processus.

each(ASSOC_ARRAY)

à partir d'un tableau associatif (c'est à dire indexé par des chaînes de caractères) *each* renvoie un tableau à deux éléments constitué par la clé et la valeur de l'élément suivant. Ceci permet de parcourir tous les éléments d'un tableau associatif sans donner explicitement l'ensemble de ces index.
Si le tableau ne comporte plus d'élément, la valeur *False* est retournée.
Le tableau ne doit pas être modifié pendant le parcours.
Exemple : Pour imprimer toutes les valeurs d'environnement, la séquence suivante est particulièrement bien adaptée :

```
while (($cle,$valeur) = each %ENV) {
    print "$cle=$valeur\n";
}
```

eof(FILEHANDLE)

teste si la fin du fichier défini par son descripteur est atteinte.
Retour : TRUE (1) si la fin du fichier est atteinte.

eval(EXPR)

permet d'exécuter l'expression EXPR en perl. La valeur retournée est la valeur résultat de la dernière commande traitée par EXPR.

exec(LIST)

exécute un programme.

exists (EXPR) (*Perl 5*)

retourne la valeur TRUE si l'expression existe dans un tableau.
Exemple : print "existe \n" if exists $array{$key};

exit(EXPR)

quitte le programme en cours en envoyant le code de sortie EXPR. Si EXPR est omis, la valeur 0 est celle par défaut.

exp(EXPR)

calcule l'exponentielle de l'expression EXPR

fcntl(FILEHANDLE,FUNCTION,SCALAR) (*Unix*)

exécute l'appel système UNIX fcntl.

fileno(FILEHANDLE)

donne le descripteur de fichier d'un fichier défini par son descripteur. (voir select())

flock(FILEHANDLE,OPERATION) (*Unix*)

voir les man pages de flock pour locker un fichier.

fork (*Unix*)

exécute une commande fork et retourne l'identification du process père et 0 pour le process fils.

formline (PICTURE, LIST) (*Perl 5*)

fonction interne utilisée pour les formats, elle est utilisable en Perl 5. Voir la partie format.

getc(FILEHANDLE)

retourne le caractère suivant lu sur le fichier dont le nom de descripteur est FILEHANDLE. Dans le cas, où aucun argument n'est donné, la lecture est faite sur l'entrée standard.

getlogin (*Unix*)

renvoie le nom de login de l'utilisateur courant.

getpeername(SOCKET) (*Unix*)

donne l'adresse de la socket connectée avec la socket donnée en argument.

getpgrp(PID) (*Unix*)

donne le numéro de groupe du processus identifié par l'identifiant PID. Si PID est égal à 0, PID est le processus courant.

getppid (*Unix*)

retourne le PID du process père.

getpriority(WHICH,WHO) (*Unix*)

donne la priorité du process (voir l'appel système du même nom)

getpwnam NAME, getgrnam NAME, gethostbyname NAME, getnetbyname NAME, getprotobyname NAME, getpwuid UID , getgrgid GID , getservbyname NAME,PROTO , gethostbyaddr ADDR,ADDRTYPE , getnetbyaddr ADDR,ADDRTYPE , getprotobynumber NUMBER , getservbyport PORT,PROTO getpwent , getgrent , gethostent , getnetent , getprotoent , getservent setpwent , setgrent , sethostent STAYOPEN , setnetent STAYOPEN , setprotoent STAYOPEN , setservent STAYOPEN , endpwent , endgrent , endhostent , endnetent , endprotoent , endservent (*Perl 5*)

Ces procédures ont la même signification que les fonctions UNIX du même nom.

Exemple :

```
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,
        $shell) = getpw*
($name,$passwd,$gid,$members) = getgr*
($name,$aliases,$addrtype,$length,@addrs) = gethost*
($name,$aliases,$addrtype,$net) = getnet*
($name,$aliases,$proto) = getproto*
($name,$aliases,$port,$proto) = getserv*
```

getsockname SOCKET (*Perl 5*)

retourne l'adresse sous forme empaquettée de SOCKET
Exemple :

```
$sockadr = 'S n a4 x8';
$masockadr = getsockname(S);
($family, $port, $myaddr) =
unpack($sockaddr,$mysockaddr);
```

getsockopt SOCKET,LEVEL,OPTNAME (*Perl 5*)

retoune l'option de la SOCKET demandée

glob EXPR (*Perl 5*)

retourne la valeur de EXPR avec les extensions de nom de fichier comme l'OS le fait.
Exemple :

```
$valeur = glob ('test.*') ;
print "$valeur";
# imprime test.pl si ce fichier est présent sur le disque
```

gmtime(EXPR)

convertit dans un tableau de 9 éléments une heure donnée par EXPR.

```
($sec,$min,$heure,$jour,$mois,$annee,$wday,$yday,$isdst) =
gmtime(time);
```

Si EXPR est omis, donne l'heure du système.

goto LABEL

à n'utiliser sous aucun prétexte car elle n'est pas implémentée sur tous les interpréteurs Perl.

grep(EXPR,LIST)

évalue si l'expression EXPR est présente dans chaque élément de la liste LIST et retourne un tableau avec les valeurs de recherche à vrai ou faux.
Si LIST est résumé à un seul élément, la valeur de retour est le nombre d'occurrences de EXPR.
EXPR peut contenir toutes les expressions régulières connues en Perl.

hex(EXPR)

convertit la valeur hexadécimale EXPR en une valeur décimale.

index(STR,SUBSTR,POSITION)

renvoie la position de la première occurrence de SUBSTR dans STR après POSITION. Si POSITION est omis, commence au début.

import : (*Perl 5*)

Ceci n'est pas réellement une fonction mais une méthode permettant d'importer un nom depuis un module. La fonction use() utilise cette méthode.

int(EXPR)

donne la valeur entière de l'expression EXPR

ioctl(FILEHANDLE,FUNCTION,SCALAR) (*Unix*)

est identique à l'appel système Unix du même nom.

join(EXPR,LIST)

transforme la liste ou le tableau LIST en une chaîne de caractères séparés par les caractères EXPR. C'est l'inverse de la fonction split.
Exemple : $ville = join (':', @ville) ;

keys(ASSOC_ARRAY)

renvoie un tableau constitué de toutes les clés du tableau associatif.
Exemple :

```
@keys = keys %ENV;
@values = values %ENV;
while ($#keys >= 0) {
    print pop(@keys), '=', pop(@values), "\n";
}
```

kill(numero,LIST) (*Unix*)

envoie le signal *numero* à une liste de Process LIST. Si le numéro donné est négatif, *kill* envoie un process groupe.
Retour : Le nombre de proccess sur lesquels un signal a été envoyé.

last LABEL

permet une rupture dans les boucles et les instructions répétitives pour aller à l'étiquette LABEL.

lc EXPR : (*Perl 5*)

retourne EXPR en caractères minuscules

lcfirst EXPR : (*Perl 5*)

retourne EXPR dans la capitalisation d'origine sauf le premier caractère qui est forcé en minuscule.

length(EXPR)

donne le nombre de caractères de l'expression EXPR .

link(FILE1,LINK)(Unix)

un lien LINK sur le fichier FILE.
Retour: 1 en cas de succès, 0 sinon.

listen(SOCKET,QUEUESIZE)(Unix)

identique à l'appel système du même nom

local(LIST)

déclare une liste de variables dans le block courant, dans la subroutine etc.

localtime(EXPR)

donne l'heure système dans un tableau à neuf éléments comme suit :
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)
=localtime(time);

log(EXPR)

donne le logarithme en base e de EXPR

lstat(FILEHANDLE)(Unix)

identique à la fonction *stat* mais sur un lien.

m/PATTERN/gio

Cette commande est sans doute la plus usitée en Perl. Elle permet de tester la présence d'une chaîne de caractères dans une chaîne. Mais elle est très déroutante car elle peut être utilisée de nombreuses façons.

- Le caractère m est optionnel, il peut être omis si le caractère qui le suit est /, sinon il est indispensable.
- les symboles / peuvent être remplacés par tout autre caractère (ce qui est utile si le caractère / est présent dans le PATTERN)
  Exemple : m:/etc/passwd; permet de rechercher la chaîne */etc/passwd*
- le symbole g indique qu'il faut réitérer la recherche plusieurs fois
- le symbole i permet de ne pas distinguer les majuscules des minuscules
- le symbole o permet de ne pas interpréter la chaîne PATTERN, c'est à dire à ne pas développer les expressions régulières comprises dans PATTERN.
- Enfin, la commande peut être utilisée de 3 façons :
  - seule, sa valeur de retour est placée dans $_
  - avec =~ sa valeur est placée en tant que booléen dans la variable de retour
  - avec !~ sa valeur est placée comme négation du booléen dans la variable de retour

Exemples :

- $base= ($file=~m|.*/([^/]+)$|); est la commande basename d'UNIX qui renvoie le nom d'un fichier quand il est collé au bout de son chemin d'accès .
- if ($reponse=~ /OUI/i) teste $reponse est OUI, Oui, OUi, ou oui.

map BLOCK|EXPR LIST : (Perl 5)

évalue BLOCK ou EXPR pour tous les éléments de LIST et retourne une liste composée des résultats de toutes les évaluations
Exemple :

```
@chars = map(chr, @nums);     # convertit la liste des nombre
                              de @nums dans les caractères correspondants.
```

mkdir(REP,MODE)

Crée le répertoire REP, avec les permissions (*Unix*) MODE
Retour : 1 en cas de succès 0 sinon.

msgctl(ID,CMD,ARG), msgget(KEY,FLAGS) msgsnd(ID,MSG,FLAGS), msgrcv(ID,VAR,SIZE,TYPE,FLAGS)

voir les appels systèmes V du même nom.

my EXPR : (Perl 5)

déclare les variables locales aux bloc courant. A la différence de la fonction *local* les variables déclarées sont totalement masquées à l'extérieur du bloc ce qui est utile pour les déclarations de subroutines.

next LABEL

sort de la boucle ou de la fonction répétitive en cours pour aller sur l'instruction suivant le block.

no Module LIST : (*Perl 5*)

l'opposé de la fonction use.

oct(EXPR)

transforme la valeur octale EXPR en une valeur décimale.

open(FILEHANDLE,EXPR)

ouvre le fichier EXPR, et lui associe le descripteur FILEHANDLE. Si EXPR est omis, le nom du fichier est FILEHANDLE. Si le nom du fichier commence par :

- > : le fichier est ouvert en écriture et vidé le cas échéant
- >> : le fichier est ouvert en écriture pour que les données qui seront écrites le soient à la fin du fichier.
- < : le fichier est ouvert en lecture.
- >+< : le fichier est ouvert en lecture et écriture.

Si la valeur EXPR vaut :

- >- : ouvre la sortie standard.
- - : ouvre l'entrée standard
- | : le fichier est un pipe

Des commandes peuvent être enchaînées derrière le symbole |, par exemple "|sort" triera la sortie du fichier ouvert.
Retour : valeur non nulle en cas de succès.

opendir(DIRHANDLE,EXPR)

Ouvre un répertoire nommé EXPR avec un descripteur de répertoire DIRHANDLE. Cette ouverture permettra d'exécuter les fonctions readdir(), telldir(), seekdir(), rewinddir() et closedir().
Retour : true en cas de succès

ord(EXPR)

donne la valeur numérique décimale du caractère ASCII EXPR. Si EXPR est une chaîne de caractères, *ord* donne la valeur ASCII du premier caractère de la chaîne

pack(TEMPLATE,LIST)

transforme un tableau ou une liste de valeurs en une structure binaire
Retour : la chaîne de caractères contenant la structure binaire
TEMPLATE est une séquence de caractères qui donne l'ordre et le type des valeurs comme suit :

- AAn chaîne ascii complétée par des blancs
- aAn chaîne ascii complétée par des 0.
- cA caractères signés.
- CAn caractères non signés
- sA entier short signé
- SAn entier short non signé
- iA entier signé
- IAn entier non signé
- lA entier long signé
- LAn entier long non signé
- nA entier short dans l'ordre RPC
- NA entier long dans l'ordre RPC
- fA réel simple précision
- dA réel double précision
- pA pointeur sur une chaîne de caractères
- vA entier short dans l'ordre little endian
- VA entier long dans l'ordre little endian
- xA caractère null

- XBack un caractère au dessus.
- @Null remplit dans la position absolue
- uA chaîne uuencodée
- bA chaîne de bits en ordre ascendant
- BA chaîne de bits en ordre descendant
- hA chaîne en hexadécimal ordre ascendant
- HAchaîne en hexadécimal ordre descendant

Chaque lettre peut être suivie par un compteur de répétition.

Exemples:

```
$toto = pack("cccc",65,66,67,68); donne "ABCD"
$toto = pack("c4",65,66,67,68); même chose
$toto = pack("ccxxcc",65,66,67,68); donne "AB\0\0CD"
$toto = pack("s2",1,2); donne "\1\0\2\0"
```

pipe(READHANDLE,WRITEHANDLE) (*Unix*)

ouvre un pipe comme l'appel système du même nom.

pop(ARRAY)

donne la dernière valeur d'un tableau.

pos SCALAR : (*Perl 5*)

retourne la position donnée par la dernière commande m//g.

print(FILEHANDLE LIST)

affiche une chaîne de caractères ou une liste. Retourne une valeur non nulle en cas de succès. Si FILEHANDLE est spécifié, l'écriture se fait dans le fichier de descripteur FILEHANDLE. Dans le cas contraire l'écriture se fait sur la sortie standard.

Exemple1:

```
$exemple="chaine de caractère\n libre \n " ;
#Cette syntaxe permet d'afficher la syntaxe entre  <<MARQUE
#(tous les symboles doivent être attachés) et la balise MARQUE
qui doit
# être mise en première colonne
print <<MARQUE;
   impression de $exemple
   et du reste
MARQUE
```

Ceci affichera le texte après substitution de la chaîne exemple.

Exemple2:

```
#combinaisons

#Cette syntaxe permet d'afficher la séquence en utilisant une
seule fois
# la commande print
print <<MARQUE1 , <<MARQUE2 ;
  bonjour
  bonsoir
MARQUE1
  Comment ça va ?
  Très bien !
MARQUE2
Ceci affichera
  bonjour
  bonsoir
  Comment ça va ?
  Très bien !
```

printf(FILEHANDLE LIST)

permet d'afficher la chaîne de caractères comme avec *print* mais en utilisant les arguments de la commande sprintf pour le formatage des données (voir sprintf).

push(ARRAY,LIST)

ajoute LIST à la fin du tableau ARRAY.

q/STRING/, q/STRING/, qq/STRING/ et qx/STRING/

permettent de coter une portion de chaîne de caractères sans coter tous ces caractères. q correspond à une simple cote et qq à une double. Le caractère / peut être remplacé par n'importe lequel des caractères non contenus dans STRING.

Exemple :

```
$toto = q!Qu'il est laid "le bidet"! ;
```

quotemeta EXPR : (*Perl 5*)

retourne la valeur de EXPR avec tous les caractères interprétés comme caractères de contrôle d'expression régulière précédés d'un antislash.

Exemple :

```
$valeur = "(3,4)toto.titi" ;
$valeur = quotemeta ($valeur) ;
print "$valeur\n" # imprime \(3\,4\)toto\.titi
```

rand(EXPR)

renvoie un nombre aléatoire compris entre 0 et EXPR ou entre 0 et 1 si EXPR est omis.

read(FILEHANDLE,SCALAR,LENGTH,OFFSET)

lit au plus LENGTH octets rangés dans la variable SCALAR depuis le fichier identifié par le descripteur FILEHANDLE. Le paramètre OFFSET sert à spécifier que les données lues, seront rangées avec le décalage OFFSET dans la chaîne de caractères.
Retour : le nombre d'octets lus.

readdir(DIRHANDLE)

retourne le fichier suivant dans le répertoire ouvert avec le descripteur de répertoire DIRHANDLE

readlink(EXPR) (*Unix*)

donne la valeur du lien symbolique EXPR.

recv(SOCKET,SCALAR,LEN,FLAGS) (*Unix*)

reçoit sur le socket SOCKET, LEN octets qui sont stockés dans la variable SCALAR. FLAGS a le même usage que dans l'appel système du même nom.

redo LABEL

exécute une boucle en se branchant à l'étiquette LABEL sans tester la condition de sortie de boucle.

ref EXPR : (*Perl 5*)

retourne la valeur TRUE si EXPR est une référence du type REF. Si EXPR est du type SCALAR, ARRAY, HASH, CODE GLOB c'est cette valeur qui est retournée. Dans le cas où EXPR n'est d'aucun de ces types FALSE est retourné.

rename(OLDNAME,NEWNAME)

renomme le fichier OLDNAME en NEWNAME.

require(EXPR)

requiert la librairie EXPR.

reset(EXPR)

réinitialise les variables commençant par la valeur EXPR.

return LIST

retourne la valeur LIST d'une procédure. Si return est omis, la procédure retournera la valeur de la dernière commande de la subroutine.

reverse(LIST)

dans un tableau ou une liste LIST, retourne la liste dans l'ordre inverse.

rewinddir(DIRHANDLE)

positionne la valeur du répertoire courant pour la procédure *readdir* à la valeur DIRHANDLE.

rindex(STR,SUBSTR,POSITION)

renvoie la position de la dernière occurrence de SUBSTR dans STR après POSITION. Si POSITION est omis, commence à la fin de la chaîne.

rmdir(FILENAME)

supprime le répertoire FILENAME si celui si est vide.
Retour : 1 en cas de succès**.**

s/PATTERN/REPLACEMENT/gieo

recherche la chaîne PATTERN pour la remplacer par la chaîne REPLACEMENT. Le paramètre g indique que la substitution doit être faite sur toutes les occurrences. Le paramètre i indique que les caractères majuscules ou minuscules sont pris en compte. Le paramètre e indique que la chaîne REPLACEMENT est une expression à évaluer et non pas une chaîne de caractères. Les caractères / peuvent être remplacés par un caractère quelconque. Les caractère =~ et != permettent d'affecter la chaîne donnée à gauche.
Retour : le nombre de substitutions effectuées.

Exemples :

- s/\bblanc\b/noir/g; # ne change pas blanche
- $path =~ s|/usr/bin|/usr/local/bin|;
- s/Login: $nom/Login: $valeur/;

scalar(EXPR)

force EPXR à être interprété en tant que scalaire et retourne la valeur interprétée.

seek(FILEHANDLE,POSITION,WHENCE)

positionne le pointeur de fichier dont le descripteur est FILEHANDLE à la position POSITION.
Retour : 1 en cas de succès.

seekdir(DIRHANDLE,POS)

positionne le répertoire courant sur DIRHANDLE, au vu de la procédure readdir. POS doit être une valeur retournée par telldir.

select(FILEHANDLE)

donne le descripteur de fichier sélectionné. Ceci permet d'écrire ou de lire sur un fichier sans utiliser explicitement le descripteur.

semctl(ID,SEMNUM,CMD,ARG) (*Unix*)

identique à l'appel système du même nom.

semget(KEY,NSEMS,SIZE,FLAGS) (*Unix*)

identique à l'appel système du même nom.

semop(KEY,OPSTRING) (*Unix*)

identique à l'appel système du même nom.

send(SOCKET,MSG,FLAGS,TO) (*Unix*)

envoie un message sur un socket. Prend le même FLAG que l'appel système du même nom

setpgrp(PID,PGRP) (*Unix*)

positionne le PID comme l'appel système du même nom.

setpriority(WHICH,WHO,PRIORITY) (*Unix*)

donne la priorité à un process, un groupe de process ou un utilisateur.

setsockopt(SOCKET,LEVEL,OPTNAME,OPTVAL) (*Unix*)

positionne les options du socket courant.

shift(ARRAY)

décale les valeurs d'un tableau du dernier vers le premier.
Retour : la première valeur du tableau
Exemple : pour enlever le premier champ de la ligne "100:Maire:Gilles" on peut faire la chose suivante :

```
@champs=split (':',"100:Maire:Gilles") ;
shift (@champs) ;
$ligne =join (':',@champs ) ;
```

shmctl(ID,CMD,ARG) shmget(KEY,SIZE,FLAGS) shmread(ID,VAR,POS,SIZE) shmwrite (ID,STRING,POS,SIZE) (*Unix*)

voir les appels système V du même nom au sujet de la gestion de la mémoire partagée.

shutdown(SOCKET,HOW) (*Unix*)

coupe une connexion comme l'appel système du même nom.

sin(EXPR)

donne le sinus en radian de EXPR

sleep(EXPR)

provoque une attente de EXPR secondes

socket(SOCKET,DOMAIN,TYPE,PROTOCOL) (*Unix*)

crée un socket comme l'appel système du même nom.

socketpair(SOCKET1,SOCKET2, DOMAIN,TYPE,PROTOCOL) (*Unix*)

voir l'appel système du même nom

sort(SUBROUTINE LIST)

trie une liste est retourne la liste triée. Si le paramètre SUBROUTINE est présent, il spécifie le nom d'une procédure qui retourne un entier inférieur, égal ou supérieur à 0 selon l'ordonnancement des éléments de la liste. On retrouve souvent ici les procédures > ou <=.

Exemples:

- @articles = sort @files;
- @articles = sort {$a cmp $b} @files;
- @articles = sort {$a <=> $b} @files;
- sub byage { $age{$a} <=> $age{$b}; }
  @sortedclass = sort byage @class;

splice(ARRAY,OFFSET,LENGTH,LIST)

supprime LENGTH éléments du tableau ARRAY qui sont situés à l'offset OFFSET et les remplace par les éléments contenus dans LIST. Si LENGTH et omis, supprime tous les éléments à partir de OFFSET

Retour : le nombre d'éléments supprimés.

split(/PATTERN/,EXPR,LIMIT)

éclate une chaîne de caractères EXPR dans un tableau qui est retourné. Le caractère PATTERN (par défaut le caractère blanc) sert de délimiteur de champs. PATTERN peut être plus long qu'un seul caractère. Le paramètre LIMIT donne le nombre maximum d'éléments du tableau à remplir, par défaut il n'y a pas de limite. C'est l'inverse de la fonction join.

Exemple 1 :

```
($login, $passwd, $uid, $gid, $gcos, $home, $shell)
  = split(/:/);
```

Exemple 2 :

```
($login, $passwd, @reste) = split(/:/);   # les champs suivants sont
rangés dans le tableau @reste
```

sprintf(FORMAT,LIST)

Retourne une chaîne de caractères formatée avec les convention C de printf. Le caractère * n'est pas supporté.

sqrt(EXPR)

donne la racine carrée de EXPR

srand(EXPR)

positionne la valeur de seuil pour la fonction random

stat(FILEHANDLE)

retoune un élément à 13 valeurs donnant les statistiques d'un fichier comme suit : ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size, $atime,$mtime,$ctime,$blksize,$blocks) = stat($filename);

study SCALAR|$_ : (*Perl 5*)

étudie SCALAR et renvoie les chaînes de caractères avant de faire une opération de dubstitution par exemple. Ceci permet de gagner du temps. (En cours de rédaction)

substr(EXPR,OFFSET,LEN)

extrait la sous chaîne commençant à l'adresse OFFSET et de longueur LEN dans l'expression EXPR. Si OFFSET est de valeur négative, le déplacement est pris à partir de la fin de la chaîne. Si LEN est omis retourne la totalité de la chaîne à partir de l'adresse OFFSET.

symlink(OLDFILE,NEWFILE) (*Unix*)

crée un nouveau lien symbolique NEWFILE sur le fichier OLDFILE.
Retour : 1 en cas de succès.

syscall(LIST) (*Unix*)

provoque l'appel système donné par le premier élément de la liste LIST.

sysread(FILEHANDLE,SCALAR,LENGTH,OFFSET) (*Unix*)

lit LENGTH octets de données depuis le fichier identifié par son descripteur FILEHANDLE. Le résultat est mis dans la variable SCALAR. Cette opération est faite par l'appel système read.
Retour : le nombre d'octets lus.

system(LIST) (*Unix*)

fork un process appelé par la commande exec.

syswrite(FILEHANDLE,SCALAR,LENGTH,OFFSET)

écrit LENGTH octets de données depuis la variable SCALAR sur le fichier défini par son descripteur FILEHANDLE, avec l'appel système write.
Le déplacement OFFSET permet de placer les données lues avec un déplacement.
Retour : le nombre d'octets effectivement écrits.

tell(FILEHANDLE)

retourne la position courante du descripteur de fichier FILEHANDLE.

telldir(DIRHANDLE)

retourne la position du descripteur de répertoire DIRHANDLE utilisé par la procédure readdir.

tie VARIABLE,PACKAGENAME,LIST : (*Perl 5*)

lie une fonction à un paquetage qui permettra l'utilisation de VARIABLE. Typiquement, ceci est utilisé avec le paquetage DBM. Un paquetage concernant un tableau associatif doit avoir les méthodes suivantes : TIEHASH objectname, LIST ; DESTROY this ; FETCH this, key ; STORE this, key, value ; DELETE this, key ; EXISTS this, key ; FIRSTKEY this ; NEXTKEY this, lastkey. Un paquetage concernant un tableau ordinaire comporte les méthodes suivantes: TIEARRAY objectname, LIST ; DESTROY this ; FETCH this, key ; STORE this, key, value . Un paquetage concernant les valeurs scalaires doit avoir les méthodes suivantes : TIESCALAR objectname, LIST ; DESTROY this ; FETCH this ; STORE this, value.

Time

donne le nombre de secondes écoulées depuis le 1er Janvier 1970 .
Le résultat est fourni sous la forme d'un tableau à quatre éléments donnant l'heure système et l'heure utilisateur, ainsi que les temps écoulés pour le process.
Les temps sont ($user,$system,$cuser,$csystem) = times;

times : (*Perl 5*)(*Unix*)

retoune un tableau de quatre éléments donnant le temps utilisateur et le temps CPU en secondes pour le processus en cours et les processus fils.

tr/SEARCHLIST/REPLACEMENTLIST/cds

traduit toutes les occurrences des caractères présents dans SEARCHLIST en caractères de remplacement donnés par la liste REPLACEMENTLIST.
Le caractère c signifie que SEARCHLIST est complémenté, le caractère d que les caractères trouvés doivent être détruits. Le caractère s spécifie que les caractères répétitifs doivent être résumés à un seul d'entre eux.
Retour : le nombre de caractères remplacés ou détruits.
Exemple :

- $c = tr/*/*/; # nombre d'étoiles dans $_
- $c = tr/0-9//; # nombre de digits dans $_
- tr/a-zA-Z//s; # Giiiillles devient Giles
- ($HOST = $host) =~ tr/a-z/A-Z/; #capitalise

truncate(FILEHANDLE,LENGTH) (*Unix*)

tronque le fichier ouvert avec le descripteur FILEHANDLE en un fichier de LENGTH octets.

uc EXPR : (*Perl 5*)

retoune les caractères de EXPR en majuscule.

ucfirst EXPR : (*Perl 5*)

retourne les caractères de EXPR avec la première lettre forcée en majuscule.

umask(EXPR) ()*Unix*

met la valeur umask pour le process et retourne l'ancienne valeur de umask. Si EXPR est omis, retourne la valeur courante de umask

undef(EXPR)

annule la définition de l'expression EXPR.

unlink(LIST)

supprime la liste de fichiers LIST et retourne le nombre de fichiers détruits.

unpack(TEMPLATE,EXPR)

fait l'inverse de la procédure PACK, à savoir qu'à partir d'une structure *unpack* fabrique un tableau de valeurs. Le paramètre TEMPLATE a les mêmes définitions que dans pack.

unshift(ARRAY,LIST)

à l'opposé de shift, fabrique un tableau ARRAY à partir d'une liste LIST.

untie VARIABLE : (*Perl 5*)

annule la liaison entre VARIABLE et le package lié par tie.

utime(LIST)

change les dates d'accès et de modification sur chacun des fichiers de la liste. Les deux premiers éléments de la liste doivent être numériques et concernent la date d'accès et de modification.
Retour : le nombre de fichiers dont la date a pu être changée.

use MODULE [LIST] : (*Perl 5*)

est équivalent à BEGIN { require MODULE; import MODULE LIST; }

values(ASSOC_ARRAY)

retourne un tableau contenant toutes les valeurs d'un tableau associatif. Les valeurs sont rendues dans un ordre récupérable avec les fonctions key() et each ()

vec(EXPR,OFFSET,BITS)

traite une chaîne de caractères EXPR comme un vecteur d'entiers non signés et retourne la valeur du bit spécifié par BITS. Les vecteurs créés avec vec() peuvent être manipulés avec les opérateurs |, & et ^. OFFSET est le déplacement à partir duquel on traite l'octet.
Retour : numéro du process achevé ou -1 en cas d'échec.

wait (*Unix*)

attend que le process fils soit terminé.

waitpid(PID,FLAGS) (*Unix*)

attend qu'un process identifié par PID se termine .
Retour : numéro du process achevé ou -1 en cas d'échec.

wantarray

retourne true si le contexte de la procédure qui s'exécute attend un tableau en entrée et faux si elle attend un scalaire.

warn(LIST)

produit un message d'erreur sur la console

write(FILEHANDLE)

écrit une donnée sur un fichier dont le descripteur est FILEHANDLE.

y/// : (*Perl 5*)

idem tr.

Bibliographie

[1] 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).

[2] Wall (Larry). - Perl référence 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.

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

[4] Vromans (Johan). - Perl Reference Guide.
jv@nl.net.
Carnet de référence contenant toutes les commandes classées par sujet. Le source L.A.T.E.X. est disponible par ftp sur les sites CPAN.