Une introduction à Perl.

Patrick Poulingeas.

3 mai 2005.

1. Introduction.

Perl (*Practical extraction and report language*) est un langage de script créé par Larry Wall en 1987. Il est très employé en administration système en raison de ses facilités à manipuler des données comme des chaînes de caractères, des tableaux, des fichiers, etc. Ces aptitudes l'ont amené à occuper une position dominante dans le domaine des scripts CGI.

Le langage dispose d'un grand nombre de bibliothèques (appelées « modules »), soit en standard, soit sur le CPAN (*Comprehensive Perl Archive Network*) qui constitue un ensemble de ressources (modules, documentations, etc.) concernant Perl. Perl peut ainsi être employé en programmation système, en programmation réseau, pour créer des interfaces graphiques, accéder à des bases de données, etc.

L'idée principale de Perl est de concevoir le langage par analogie avec une langue (humaine). De même que dans une langue il est possible d'exprimer une même chose de plusieurs façons, Perl propose de nombreuses constructions syntaxiques pour une même opération (« *There is more than one way to do it.* »).

En pratique, le langage constitue une sorte de synthèse entre des outils Unix comme sed et grep et des langages comme les shells Unix, C et awk.

Les scripts, placés dans des fichiers texte, sont compilés à la volée en bytecode par l' « interpréteur » Perl ; puis le bytecode obtenu est immédiatement exécuté.

2. Les différentes méthodes pour exécuter un programme Perl.

1. Par appel de l'interpréteur depuis la ligne de commande :

```
> perl -e 'print "Hello, world !\n";'
Hello, world !
```

2. Par exécution d'un programme contenu dans un fichier texte en appelant l'interpréteur :

```
> cat hello.pl
print "Hello, world !\n";
> perl -w hello.pl
Hello, world !
```

L'option -w (*warnings*) permet d'avoir des messages d'avertissement à la façon de l'option -Wall de gcc.

3. Par exécution d'un programme contenu dans un fichier texte :

```
> cat hello.pl
#!/usr/bin/perl
print "Hello, world !\n";
> chmod u+x hello.pl
> ./hello.pl
Hello, world !
```

3. Remarques générales.

En Perl, les commentaires suivent le symbole « # » et s'étendent jusqu'à la fin de la ligne. On ne dispose pas de symbole permettant de commenter plusieurs lignes simultanément (comme « /* » et « */ » en C).

Les instructions se terminent par un « ; ».

L'appel des fonctions peut se faire avec ou sans parenthèses :

```
print "Hello, world !\n";
print ("Hello, world !\n");
```

Il est cependant parfois nécessaire de mettre des parenthèses, par exemple en raison de la faible priorité de l'opérateur « , » qui sert à séparer les paramètres effectifs d'une fonction.

Le paramètre d'une fonction peut parfois être omis. Le paramètre est alors la variable spéciale « $_ » dont le contenu est automatiquement mis à jour en fonction du contexte d'exécution.

4. Les types de données.

4.1. Introduction.

Perl n'oblige pas à déclarer les variables, ni le type de celles-ci. On peut cependant effectuer une déclaration de variable avec le mot-clé « my ».

Parmi les principaux types de données, on note :

- les scalaires (nombres, chaînes de caractères)
- les tableaux
- les tables de hachage (ou tableaux associatifs)
- les FILEHANDLE (descripteurs de fichiers)

Une notion fondamentale en Perl est celle de contexte d'évaluation. La façon dont une opération se déroule dépend du contexte dans lequel elle est placée (De même, dans une langue, un mot peut prendre un sens différent selon le contexte dans lequel il est employé).

Exemple :

```
my @t = ("a","b","c");          # t est un tableau de chaînes
my $x = @t;
```

$x est un scalaire.

Situé en partie gauche d'une affectation, il impose une évaluation en contexte scalaire de la partie droite de l'affectation.

L'évaluation d'un tableau en contexte scalaire est son nombre d'éléments.

Donc, après la deuxième affectation ci-dessus, on a : $x == 3.

Il existe deux contextes principaux :

- le contexte scalaire (se divisant en contexte de chaîne, contexte numérique et contexte booléen),
- le contexte de liste (~ tableaux + tables de hachage).

Des fonctions permettent de forcer le contexte :

```
$x = scalar(@t);        # $x vaut 3
```

4.2. Les scalaires.

4.2.1. Les variables scalaires.

Les noms de variables scalaires commencent par le symbole « $ ». On peut ensuite employer des chiffres, des lettres ou le symbole « _ ». On notera que Perl est sensible à la casse.

Quand une variable scalaire n'est pas initialisée, elle prend la valeur spéciale « undef » (qui est par ailleurs une valeur souvent renvoyée par une fonction en cas d'erreur).

```
my $msg;                # $msg vaut undef

$x = undef;             # Attribution de la valeur undef à la variable $x

if (defined($msg))              # Fonction permettant de tester si un scalaire
                                # est défini ou non
{
        print "$msg\n";
}
```

4.2.2. Les chaînes de caractères.

Les chaînes sont délimitées en Perl par des doubles quotes (") ou des simples quotes (').

Dans une chaîne délimitée par des doubles quotes, le contenu est interprété :

```
$nom = "Durand";
$msg = "Bonjour $nom !";    # $msg vaut « Bonjour Durand ! »

$m = "salu";
$s = "Je vous ${m}e.";      # $s vaut « Je vous salue. »
```

Si l'on oublie les accolades dans l'instruction précédente, c'est la valeur du scalaire $me qui est recherchée.

On dispose aussi des symboles d'échappement : « \n », « \t », « \" », etc.

Dans un littéral chaîne délimité par des simples quotes, le contenu est pris tel quel.

```
$msg = 'Salut $nom.\n';        # $msg vaut « Salut $nom.\n »
```

Remarque :
Contrairement au C, le caractère \0 ne signale pas une fin de chaîne et peut très bien apparaître au milieu d'une chaîne :

```
$s = "x\0f";                   # Chaîne de 3 caractères
```

Quelques opérateurs et fonctions pour les chaînes de caractères :

- La concaténation se fait avec l'opérateur « . » :

```
$x = "bon"."jour";             # $x vaut « bonjour »
$x = "bon".3;                  # Contexte de chaîne dû à l'opérateur « . »
                               # 3 est converti en chaîne
                               # $x vaut « bon3 »
$x .= $y;                      # Equivalent à : $x = $x.$y;
```

- Répétition d'une chaîne :

```
$s = "bla"x3;                  # $s vaut « blablabla »
```

- Longueur d'une chaîne :

```
length($s)
```

Exemple :

```
my $s = 'ba\n';
print length($s)."\n";         # Affiche : 4
```

- $c = chop($s);
  Supprime le dernier caractère de $s ($s est modifiée) et le place dans $c.

- chomp ($s);
  Supprime le dernier caractère de $s **s'il s'agit d'une fin de ligne (« \n »)**.

- $m = reverse($s);
  Stocke dans $m la chaîne contenue dans $s inversée.

- substr($s,$pos,$longueur)
  Renvoie la sous-chaîne extraite de $s à partir de la position $pos et de longueur $longueur.
  Les positions sont indexées à partir de 0.
  Remarque : Cette fonction peut apparaître en partie gauche d'une affectation.

```
my $msg = "salut toi";
substr($msg,5,1) = "ation a ";         # $msg vaut « salutation a toi »
```

- Recherche d'une sous-chaîne dans une chaîne :

```
index($chaine,$sous_chaine)            # Recherche à partir du début de la chaîne
index($chaine,$sous_chaine,$position_debut_recherche)
```

La fonction renvoie -1 si elle ne trouve pas la sous-chaîne dans la chaîne.

- *Here document* (Document en ligne) :

```
$msg = <<FIN;
bla bla
bla bla bla
bla
FIN
print $msg;
```

Affiche sur la sortie standard :

```
bla bla
bla bla bla
bla
```

Tout ce qui se trouve avant le mot « FIN », y compris les fins de ligne, est stocké dans la variable $msg.

- Affichage d'une chaîne :

```
print "Salut";                    # Sans retour à la ligne automatique

print STDERR "Probleme\n";        # Utilisation d'un FILEHANDLE
                                  # prédéfini
```

- Lecture d'une chaîne depuis l'entrée standard :

```
$ligne = <STDIN>;                 # L'appui sur <Entrée> indique la fin de la
                                  # saisie. $ligne contiendra un « \n » en fin
                                  # de chaîne.
```

Exemple d'utilisation de la variable spéciale « $_ » :

```
while (<STDIN>)
{
        print;
}
```

Les instructions précédentes sont équivalentes à :

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

Fonctionnement des programmes ci-dessus :
L'utilisateur saisit une chaîne et appuie sur <Entrée>. La valeur saisie est alors placée dans $_. On affiche ensuite le contenu de $_.
Pour sortir de la boucle, on appuie sur CTRL-D, ce qui a pour effet de retourner la valeur undef dans $_.

- Opérateur d'exécution de commandes (*backtick*) :

```
$informations = `ls -l ~/.bashrc`;      # Permet de récupérer le résultat d'une
                                        # commande Shell
print $informations;
```

Résultat obtenu :

```
-rw-r--r--  1 poulinge profs 872 oct  4  2004 /home/profs/poulinge/.bashrc
```

A l'intérieur des apostrophes inverses, les variables Perl sont interpolées. En contexte scalaire, le résultat de la commande est stocké intégralement dans la variable chaîne.

Remarque : En contexte de chaîne, « undef » est évalué comme une chaîne vide.

4.2.3. Les nombres.

On dispose des opérateurs classiques : + - * / % ++ --
La division est une division réelle. Pour avoir une division entière, on écrit :

```
$x = int(2/3);          # $x vaut 0
```

L'opérateur ** correspond à la puissance.
On dispose des fonctions suivantes : sin, cos, exp, log, abs, sqrt, rand, …

En contexte numérique, les chaînes sont converties en nombres :

```
$s = "99";
$s = $s + 2;
print "$s\n";           # Affiche 101
```

Les chaînes commençant par un caractère autre qu'un chiffre, la chaîne vide et la valeur spéciale « undef » valent 0.
Exemple :

```
$s = "";
$x = $s + 0;
print "$x\n";           # Affiche 0
```

Avec l'option -w, on a le message d'avertissement suivant :

```
Argument "" isn't numeric in addition (+) at ./programme.pl line 5.
```

La ligne 5 étant celle où l'on a l'affectation : $x = $s + 0;

4.3. Les tableaux.

Une variable de type tableau se définit de la sorte :

```
my @tab=('ab',3,"z");
```

En partie droite de l'affectation apparaît une liste de scalaires. On voit qu'une liste (et donc un tableau) peut être formée d'éléments de types différents.

On dispose de facilités pour construire des listes :

```
my @tab = (1..10);    # Tableau de 10 éléments, les nombres de 1 à 10
my @tab = ('a'..'c'); # @tab vaut ('a','b','c')
```

```
my @tab = (2,'a')x3;  # @tab vaut (2,'a',2,'a',2,'a')
```

Les listes que l'on manipule sont des listes « linéaires ». Il y a ainsi un aplatissement des listes :

```
my @tab = (1,(3,4),'a');        # @tab vaut (1,3,4,'a')
```

Quand on déclare un tableau sans lui attribuer une liste comme valeur, on crée un tableau vide. Aussi :

```
my @tab;
```

est équivalent à :

```
my @tab = ();                   # () est la liste vide
```

Chaque élément d'un tableau étant un scalaire (du fait de l'aplatissement des listes), on accède à ceux-ci avec le préfixe des scalaires : « $ ».

```
my @tab = ("a",5);
$tab[0] = "b";                  # Modifie le 1er élément du tableau @tab
print "$tab[1]\n";              # Affiche 5
```

Les indices des tableaux commencent à 0.

On accède au dernier élément avec la notation $tab[-1], à l'avant-dernier avec la notation $tab[-2], etc.

Les tableaux sont dynamiques :

```
@tab = ("a",3);
$tab[999] = "z";
```

@tab est un tableau de 1000 éléments. Les éléments de 2 à 998 valent « undef ».

On peut effectuer des affectations entre tableaux :

```
my @t1 = ("b",2);
my @t2 = ('A'..'Z',"_");
@t1 = @t2;                      # Copie par valeur
```

Les affectations peuvent faire intervenir des listes :

```
($a,$b) = (1,3);        # $a vaut 1 et $b vaut 3
($a,$b) = (1,3,5);      # Même chose ; 5 est ignoré
($a,$b) = (1);          # $a vaut 1 et $b vaut « undef »
($a,$b) = @t;           # $a vaut $t[0] et $b vaut $t[1]
($a,$b) = ($b,$a);      # Permutation de $a et $b
($a,@t) = @s;           # $a vaut $s[0], @t absorbe tous les autres éléments
                        # de @s
 ($a,@t,$b) = @s;       # $a vaut $s[0], @t absorbe tous les autres éléments
                        # de @s, $b vaut « undef »
```

On peut aussi utiliser des tranches de tableaux :

```
my @t = ("a",2,1,'k');
my @s = @t[0..2];               # @s vaut ("a",2,1)
my @r = @t[1,3];                # @r vaut (2, 'k')
```

On dispose en outre de l'opérateur suivant pour les tableaux :

$#tab : fournit l'indice du dernier élément (sous-entendu différent de « undef ») du tableau @tab.

En contexte scalaire, un tableau s'évaluant comme son nombre d'éléments, pour obtenir cette valeur, il suffit d'écrire :

```
$taille = @tab;
```

Remarque sur l'affichage « direct » d'un tableau avec print :

L'instruction « print » a la forme générale suivante :

```
print liste;
```

Les éléments de la liste, placés dans le contexte de chaîne de caractères, sont alors affichés, côte à côte, sur la sortie standard.

Exemple :

```
print 2,"aa",atan2(1,1)*4,'zzz';
```

affiche :

```
2aa3.14159265358979zzz
```

(sans retour à la ligne).

Un tableau pouvant être vu comme une liste, les instructions suivantes :

```
@t = (5,'a',"lp");
print @t;
```

produisent l'affichage :

```
5alp
```

(sans retour à la ligne).

Interpolation d'un tableau dans un littéral chaîne de caractères délimité par des doubles quotes :

Dans ce cas, chaque élément est interpolé puis affiché. A l'affichage, les éléments sont séparés par des espaces.

```
$x = 7;
@tab = (4,"a","$x","yy");
print "@tab";
```

Résultat obtenu :

```
4 a 7 yy
```

Ajout et suppression à gauche d'un tableau :

```
my @t = (3..6);
unshift(@t,1,2);        # @t vaut (1,2,3,4,5,6)
$x = shift(@t);         # $x vaut 1, @t vaut (2,3,4,5,6)
```

Ajout et suppression à droite d'un tableau :

```
my @t = (3..6);
push(@t,7,8,9);         # @t vaut (3,4,5,6,7,8,9)
$x = pop(@t);           # $x vaut 9 et @t vaut (3,4,5,6,7,8)
```

Avec push et pop, on peut gérer un tableau comme une pile.

Avec push et shift, on peut gérer un tableau comme une file.

Quelques fonctions agissant sur les tableaux ou les listes :

- Concaténation des éléments d'une liste dans une chaîne :

  $chaine = join($separateur,*liste*);

  Exemples :
  $s = join("::",1,3,5); # $s vaut "1::3::5"
  $s = join(" ",@tab); # $s vaut les éléments de @tab séparés par un espace

- Découpage d'une chaîne de caractères :

  liste = split(/*séparateur*/,$chaine);
  Le séparateur intervenant est une expression régulière.

  Exemples :
  my @t = split(/-/,"10-05-02"); # @t vaut ("10","05","02")
  ($x,$y) = split(/ /,"la le lo"); # $x vaut "la", $y vaut "le"

- Tri d'un tableau :

  @trie = sort(@tab); # On trie en utilisant l'ordre ASCII

- Inversion d'un tableau :

  @inverse = reverse(@tab);

- Sélection des éléments d'une liste :

  liste_résultat = grep { *critère* } *liste_départ*;

  Le critère est une expression booléenne dans laquelle $_ désigne un élément de la liste de départ. Si l'expression est évaluée à « vrai », l'élément est conservé dans la liste résultat.
  $_ parcourt tous les éléments de la liste de départ.

  Exemples :
  @t = grep { $_ >= 0 } $x,$y,$z; # Conserve dans @t les scalaires ≥ 0

  @nombres = (0..10);
  @t = grep { $_ % 2 == 0 } @nombres; # @t vaut (0,2,4,6,8,10)

- Application d'une opération à tous les éléments d'une liste :

  liste_résultat = map({ *expression* } *liste_départ*);

  Exemple :
  @nombres = (0..5);
  @au_carre = map({ $_**2 } @nombres); # @au_carre vaut (0,1,4,9,16,25)

Un tableau prédéfini :
@ARGV : Ce tableau contient les arguments de la ligne de commande pour le script. Contrairement au C, $ARGV[0] n'est pas le nom du script mais le premier argument de celui-ci (Pour obtenir le nom du script, on dispose de la variable spéciale $0).

Remarque :
Les tableaux vus dans ce cours n'ont qu'une dimension. Pour avoir des tableaux multidimensionnels, du fait de l'aplatissement des listes, on a recours à des références (~ pointeurs) sur des tableaux (Les références sont des scalaires).

4.4. Les tables de hachage.

Ce sont des associations entre une clé et une valeur.
Exemple :

```
my %index_telephonique = ("Olivier" => "05.55.12.00.00",
                          "Eric"    => "05.55.13.00.00",
                          "Jean"    => "05.55.13.00.01");
```

Remarque :

- La partie droite est une liste. Le symbole spécial « => » peut être remplacé par une virgule comme dans une liste « normale ».
- Une clé est une chaîne ; une valeur associée à une clé est un scalaire.

La déclaration de variable suivante :

```
my %h;
```

crée une table de hachage vide. Elle est équivalente à :

```
my %h = ();
```

- Accès à un élément d'une table de hachage grâce à sa clé :

```
my $num_Jean = $index_telephonique{Jean};
```

Si la clé est une chaîne comportant des caractères autres que des lettres, des chiffres, le caractère « _ » ou le caractère « - », il faut mettre le nom de la clé entre apostrophes ou guillemets pour obtenir la valeur qui lui est associée.

- Dans une table de hachage, une clé est unique :

```
my %prix = ("livre"  => 10,
            "disque" => 20);
$prix{livre} = 15;              # Change la valeur associée à la clé « livre »
```

- Ajout d'une nouvelle clé à une table de hachage :

```
# En reprenant la table %prix précédente
$prix{dvd} = 37;      # Ajoute l'association : dvd => 37
$prix{cd}++;          # Crée et associe à la clé « cd » la valeur 1
                      # On considère, dans un contexte numérique,
                      # qu'il y avait une valeur initiale égale à 0
```

- Quand la clé n'existe pas, la valeur retournée est « undef ».

- Suppression d'une clé d'une table de hachage :

```
delete $prix{dvd};
```

- Prédicat permettant de tester si une clé existe dans une table de hachage :

```
if (exists($prix{dvd}))
{
   print "Le prix d'un dvd est $prix{dvd}\n";
}
else
{
   print "Pas de dvd en vente\n";
}
```

- On peut faire des affectations entre tables de hachage :

```
my %h1 = ("a" => 1,"z" => 2);
my %h2 = ('ZX' => 81,'TRS-' => 80);
%h1 = %h2;              # Copie par valeur
```

- On dispose de la fonction keys pour obtenir la liste des clés d'une table de hachage :

```
@objets = keys(%prix);
```

L'ordre des clés est quelconque et ne respecte pas l'ordre d'introduction des clés dans la table de hachage.

- La fonction values retourne la liste des valeurs d'une table de hachage :

```
@numeros = values(%index_telephonique);
```

Une table de hachage prédéfinie :
%ENV : Permet d'accéder aux variables d'environnement du processus.
Exemple :

```
print "$ENV{PATH}\n";
```

5. Les structures de contrôle.

- Les booléens n'existent pas en tant que tels en Perl.
  Dans un contexte booléen, sont considérés comme « faux » :
  - 0
  - "0" ou '0'
  - "" ou '' (La chaîne vide – avec des doubles ou des simples quotes)
  - undef

  Toutes les autres valeurs sont considérées comme « vrai ».

- Les opérateurs de test :
  Il y a une distinction entre ceux réservés aux nombres et ceux réservés aux chaînes de caractères.

Nombres	Chaînes	Signification
==	eq	Egal à
!=	ne	Différent de
<	lt	Strictement inférieur à
>	gt	Strictement supérieur à
<=	le	Inférieur ou égal à
>=	ge	Supérieur ou égal à

- Les opérateurs booléens du C se retrouvent en Perl :
  || && !

Les opérateurs || et && procèdent à une évaluation paresseuse. Ce qui permet d'écrire (un peu comme dans des scripts shell) :

```
@ARGV || die "Pas de parametre fourni au script";
```

En contexte scalaire, @ARGV vaut son nombre d'éléments. Si le nombre d'éléments de @ARGV est ≥ 1, on n'évalue pas la partie droite de l'expression. On n'évalue cette dernière (avec pour effet de bord l'affichage d'un message et l'arrêt du script) que si le tableau @ARGV est vide (car, alors, son nombre d'éléments est égal à 0 – ce qui est considéré comme « faux » dans un contexte booléen).

L'expression

```
@ARGV || die "Pas de parametre fourni au script";
```

signifie bien :

« des arguments ou sinon le script s'arrête ».

Autre exemple :

```
$< == 0 && die "Interdit a root";
```

« $< » est l'UID réelle du processus.

- ```
  if ($x == 0)
  {
          print "valeur nulle\n";
  }
  else
  {
          print "valeur non nulle\n";
  }
  ```

Les accolades sont obligatoires même s'il n'y a qu'une instruction à exécuter.

On dispose du mot-clé « elsif » :

```
if ($x == 0)
{
        print "zero\n";
}
elsif ($x == 1)
{
        print "un\n";
}
elsif ($x == 2)
{
        print "deux\n";
```
```

```
}
```

La condition peut suivre une instruction :

```
print "$x\n" if (defined($x));          # Affiche la valeur de $x si $x ≠ undef
```

On dispose d'une instruction unless dont la sémantique est :

unless(*condition*) ↔ if (!*condition*)

```
print "$x\n" unless ($x == 0);          # Affiche $x sauf si $x == 0
```

- ```
  while ($x < 10)
  {
          $x++;
  }

  $x++ while ($x < 10);

  do
  {
          $x++;
  } while ($x < 10);

  until ($x >= 10)
  {
          $x++;
  }

  $x++ until ($x >= 10);

  do
  {
          $x++;
  } until ($x >=10);
  ```

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

- Parcours d'une liste :

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

foreach $x (1..3,"a")
{
   print "$x\n";
}
```

Si l'on modifie la variable de la boucle ($element dans le 1er exemple précédent), cela a pour effet de modifier l'élément correspondant dans le tableau.

Utilisation de la variable spéciale $_ :

```
foreach (@tableau)
{
        print "$_\n";
}
```

- Itération sur les couples (clé,valeur) d'une table de hachage :

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

On peut aussi évidemment écrire :

```
foreach $cle (keys(%table))
{
        print "$cle => $table{$cle}\n";
}
```

- Instructions de rupture de séquence :
  - last : sortie immédiate de la boucle
  - next : provoque la fin de l'exécution des instruction du bloc ; dans le cas d'un « for », on met à jour la variable de boucle ; on recommence ensuite un parcours en effectuant le test de la structure itérative.
  - redo : exécute à nouveau les instructions du bloc, sans mettre à jour la variable de boucle ni tester la condition de la boucle.

  Remarque : On ne peut pas employer ces trois instructions dans une structure do/while ou do/until.

- Remarque : en Perl, on ne dispose pas d'un équivalent de switch/case.

6. Les fonctions.

On définit une fonction avec le mot-clé « sub ». Les arguments d'une fonction sont des scalaires, stockés dans un tableau spécial @_. Si la fonction retourne un résultat (scalaire ou liste), on utilise l'instruction « return ».

Exemple :

```
sub somme
{
        my $x = $_[0];          # « my » permet d'avoir une variable locale
        my $y = $_[1];
        return $x+$y;
}
```

Autre écriture :

```
sub somme
{
```

```
        my ($x,$y) = @_;
        return $x+$y;
}
```

L'appel d'une fonction se fait avec la syntaxe :

&nom_fonction liste_de_paramètres

Le symbole « & » n'est pas nécessaire si la fonction a été définie avant d'être appelée ou si la liste est en entre parenthèses.

Exemples :

```
somme(2,5);          # La valeur de retour est ignorée
$z = somme(2,5);     # $z vaut 7
$z = somme(2,5,9);   # $z vaut 7, 9 est ignoré
$z = somme(2);       # $z vaut 2 ; dans la fonction, $y vaut « undef » (ce qui
                     # entraîne un message d'avertissement avec l'option –w )
```

Exemple de fonction retournant une liste :

```
sub somme_produit
{
        my $x = shift;          # shift s'applique par défaut à @_
                                # dans une fonction
        my $y = shift;
        return ($x+$y,$x*$y);
}
```

Appels de cette fonction :

```
($a,$b) = somme_produit(1,4);       # $a vaut 5, $b vaut 4
@res = somme_produit(-2,1);         # @res vaut (-1,-2)
```

Exemple de fonction récursive :

```
sub factorielle
{
        my ($n) = @_;
        return undef if ($n < 0);
        return 1 if ($n == 0 || $n == 1);
        return $n*factorielle($n-1);
}
```

Remarque : les paramètres d'une fonction sont passés par référence dans le tableau @_.

```
sub permuter
{
        my $x = $_[0];
        $_[0] = $_[1];
        $_[1] = $x;
}
```

```
$a = 2;
$b = 7;
print "$a\t$b\n";              # Affiche : 2   7
permuter($a,$b);
print "$a\t$b\n";              # Affiche : 7   2
```

7. Les fichiers.

- Opérateurs logiques applicables aux noms de fichiers ou aux FILEHANDLE :
  Ce sont des opérateurs similaires à ceux des shells Unix.

  -d : teste si le nom suivant est un répertoire
  Exemple :

```
if (-d "/home")
{
        print "/home est un repertoire\n";
}
```

  - e : teste si la chaîne correspond à un répertoire ou un fichier existant
    (e : *exists*)

  - z : teste si le fichier est vide

  - r : teste si le script a des droits en lecture
  - w : teste si le script a des droits en écriture
  - x : teste si le script a des droits en exécution

On peut utiliser ces opérateurs sur des chaînes correspondant à des noms de fichiers ou sur des FILEHANDLE.

- Les fichiers se manipulent grâce à des FILEHANDLE (« gestionnaire de fichier »). Par convention, ceux-ci sont écrits en majuscule. Ce sont des variables spéciales qui ne commencent par aucun symbole (Pas de « $ » devant un FILEHANDLE).

- Ouverture d'un fichier :
  open (FILEHANDLE,*chaîne_de_caractères*);

La chaîne de caractères contient le nom du fichier précédé de zéro, un ou deux caractères spécifiant le mode d'ouverture du fichier. Ces caractères sont :

Caractères	Mode d'ouverture
aucun	lecture
<	lecture
>	écriture (avec remise à zéro au départ)
>>	ajout en fin de fichier

En cas d'erreur, open retourne la valeur « undef ».

Exemple d'utilisation :

```
open(FICHIER,"<donnees.txt") || die "Impossible d'ouvrir donnees.txt";
```

- Lecture dans un fichier :

```
$ligne = <FICHIER1>;        # Lecture d'une seule ligne
                            # Renvoie « undef » en fin de fichier
                            # Le « \n » de fin de ligne est mis dans la variable
@total = <FICHIER2>;        # Lecture de la totalité du fichier
                            # Chaque ligne constitue alors
                            # un élément du tableau @total
```

Exemples :

```
while (defined($ligne=<FICHIER>))
{
        print "$ligne";
}

while (<FICHIER>)
{
   print;
}

@contenu_global = <FICHIER>;
foreach $ligne (@contenu_global)
{
        chomp $ligne;
        print "Administrateur\n" if ($ligne eq "root");
}
```

- Ecriture dans un fichier :

```
print  FICHIER "ba",22,"$v";
printf FICHIER "%06d",$nombre;
```

La spécification de la chaîne de format de printf est identique à celle de la fonction C correspondante.

- Fermeture d'un fichier :

```
close (FILEHANDLE);
```

- Quelques FILEHANDLE prédéfinis :

STDIN : l'entrée standard
STDOUT : la sortie standard
STDERR : la sortie d'erreur standard
ARGV : Avec ce FILEHANDLE, les lignes lues sont celles des fichiers passés en paramètres au script. Si le script n'a aucun paramètre, on lit l'entrée standard.
Remarque : <> est équivalent à <ARGV>
Exemple d'utilisation de <> :

```
> cat texte.txt
aa
bb
1
> cat afficher.pl
#!/usr/bin/perl -w
```

```
while (<>)
{
        chomp;
        print;
}
print "\n";
> ./afficher.pl texte.txt
aabb1
> ./afficher.pl texte.txt texte.txt
aabb1aabb1
```

- Utilisation de tubes (*pipes*) :
  Récupérer la sortie d'une commande shell :

```
open CONTENU,"ls|";
while (<CONTENU>)
{
        print;
}
close CONTENU;
```

  Fournir l'entrée d'une commande shell :

```
open COURRIER,"|mail -s coucou machin\@truc.fr";
print COURRIER "Salut";
close COURRIER;
```

- Le globbing :
  On peut utiliser en Perl les jokers du shell pour obtenir un ensemble de fichiers. On dispose pour cela de la fonction glob.

  Exemple :

```
@fichier_images = glob('*.jpg');      # Tous les fichiers d'extension jpg
                                      # du répertoire courant
```

  Autre syntaxe possible :

```
@fichier_images = <*.jpg>;  # Résultat identique à l'instruction précédente
```

  Exemple d'utilisation : afficher tous les fichiers vides du répertoire courant

```
foreach $nom (<*>)
{
        print "$nom : fichier vide\n" if (-z $nom);
}
```

8. Les expressions régulières.

Les expressions régulières permettent en Perl d'extraire et de manipuler des informations facilement. Elles servent à :

- tester si un motif est contenu ou non dans une chaîne
- extraire des informations d'une chaîne
- effectuer des substitutions dans une chaîne

Exemples :

- Rechercher si le mot « chat » est présent dans une chaîne :

```
$s1 = "Le chat est ici";
$s2 = "ch cha tab";

if ($s1 =~ m/chat/)                    # m : match
{
        print "chat present dans $s1\n";       # Exécuté
}
else
{
        print "chat absent dans $s1\n";
}

if ($s2 =~ /chat/)                     # m (match) par défaut
{
        print "chat present dans $s2\n";
}
else
{
        print "chat absent dans $s2\n";        # Exécuté
}
```

- Remplacer le mot « chat » par le mot « chien » dans une chaîne :

```
$s1 = "Le truc est ici";
$s2 = "Le chat est ici";
$s3 = "Le chat est un chat";
$s4 = $s3;

$s1 =~ s/chat/chien/;          # s : substitute
                               # $s1 inchangé
$s2 =~ s/chat/chien/;          # $s2 vaut : « Le chien est ici »
$s3 =~ s/chat/chien/;          # $s3 vaut : « Le chien est un chat »
                               # On a changé uniquement la 1ère occurrence
                               # de « chat »
$s4 =~ s/chat/chien/g;         # g : global
                               # $s4 vaut « Le chien est un chien »
```

L'expression régulière /chat/ ne pose aucun problème d'interprétation : chaque caractère de l'expression correspond à lui-même. Il existe cependant des caractères qui prennent une signification spéciale dans une expression régulière. Ils permettent de spécifier divers motifs servant à construire l'expression régulière.

Exemples de symboles particuliers :

- Les ensembles de caractères :
  . : n'importe quel caractère sauf « \n »
  [ ] : définition d'un ensemble de caractères
    [abcz] : l'ensemble {a,b,c,z}
    [a-cz] : l'ensemble {a,b,c,z}
    [^b-d] : l'ensemble de tous les caractères sauf les caractères b,c et d

Exemple d'utilisation :

Vérifier si au moins un chiffre apparaît dans une chaîne $s :

```
if ( $s =~ /[0-9]/ ) { #... }
```

\d : un chiffre (équivalent à [0-9])
\D : un non-numérique (équivalent à [^0-9])
\w : un caractère d'un « mot » (équivalent à [a-zA-Z0-9_])
\W : équivalent à [^a-zA-Z0-9_]
\s : un blanc (équivalent à [\n\t\r\f])
\S : tous les caractères sauf un blanc

- Les quantificateurs :

Ce sont des symboles comme « * » qui indique qu'un motif peut être présent dans la chaîne 0 fois ou plus.

Les quantificateurs s'appliquent au motif les précédant le plus petit possible.

Exemples :

/z*/ recherche la présence de « z » 0 fois ou plus
/za*/ recherche « z » ou « za » ou « zaa » ou…

Quantificateur	Signification	Exemple	Mots recherchés
*	0 fois ou plus	/z*/	mot vide, z, zz, …
+	1 fois ou plus	/z+/	z, zz, …
?	0 ou 1 fois	/z?/	mot vide ou z
{n}	n fois exactement	/z{2}/	zz
{n,}	au moins n fois	/z{2,}/	zz, zzz, …
{n,m}	entre n et m fois	/z{2,4}/	zz, zzz, zzzz

Exemples :

- Reconnaître un nombre décimal (simplifié) dans une chaîne :

  ```
  /[+-]?\d+\.\d+/
  ```

  On remarque le « \ » avant le « . » pour indiquer que l'on recherche le caractère « . » et qu'il ne s'agit pas du méta-caractère correspondant à un motif spécial.

  Les chaînes suivantes sont en correspondance avec (sont *matchées* par) l'expression régulière ci-dessus :

  ```
  0.15
  -23.0
  az-2.7a8#
  ```

  Les chaînes suivantes ne sont pas en correspondance avec l'expression régulière :

  ```
  +368
  .64
  -13.
  +0,7
  ```

- ```
  $s = "mb1245z3";
  $s =~ s/[0-9]+/0/;
  print "$s\n";                    # Affiche « mb0z3 »
  ```

  On remarque que l'expression « [0-9]+ » a *matché* 1245 et pas seulement le chiffre 1. Un quantificateur *matche* le plus de caractères possibles de telle sorte que l'expression régulière soit satisfaite (On parle de comportement « glouton » (*greedy*)).
  ```

- $s = "12 3#5";
  $s =~ s/[0-9]+#/nombre/;
  print "$s\n"; # Affiche : « 12 nombre5 »

- Regroupement :
  /ab*/ permet de *matcher* a, ab, abb, abbb, …
  Ce comportement est dû au fait que le quantificateur s'applique au plus petit motif qui le précède (ici, le caractère « b »).
  Les parenthèses permettent de constituer un motif par regroupement de plusieurs motifs.
  Ainsi : /(ab)*/ *matche* le mot vide, ab, abab, ababab, etc.

- Alternatives :
  Supposons que l'on désire *matcher* l'un des trois mots suivants :
  chat, chien, oiseau
  On dispose de l'opérateur « | » pour cela.
  L'expression régulière que nous cherchons s'écrit :
  /chat|chien|oiseau/

  Exemples :
  /Tom|Jean|Jacques Roy/ *matche* : « Tom », « Jean » ou « Jacques Roy ».
  /(Tom|Jean|Jacques) Roy/ *matche* : « Tom Roy », « Jean Roy »
  ou « Jacques Roy ».

- Début et fin de chaîne :
  On peut indiquer que le(s) motif(s) recherché(s) commence(nt) ou termine(nt) la chaîne analysée.
  Exemples :
  /^a+/ La chaîne commence par « a », ou « aa », ou « aaa », …
  /\d$/ La chaîne se termine par un chiffre
  /^(la)+$/ La chaîne est « la », ou « lala », ou « lalala », …

- Mémorisation :
  Un regroupement (qui se fait avec des parenthèses) permet aussi de mémoriser l'expression en correspondance avec ce regroupement.
  Pour référencer ces expressions, on utilise \1, \2, …, \9 dans l'expression régulière.
  Exemples :
  - Rechercher si une chaîne possède au moins 2 fois le même chiffre :
    /(\d).*\1/
  - /^Un ([a-zA-Z]+) est un \1/
    Cette expression régulière *matche* :
    Un chat est un chat
    Un chien est un chien
    Un chat est un chat ou pas
    Un chat est un chatzf8
    Cette expression régulière ne *matche* pas :
    Un chat est un chien
    Oui ! Un chat est un chat
    un chat est un chat
    Un chat est sans doute un chat

Dans le second membre d'une substitution, on utilise les notations $1, $2, …, $9.

Exemple : ajouter « le » devant le premier mot (constitué uniquement de lettres) dans une chaîne $s.

```
$s = "chat mange";
$s =~ s/([a-zA-Z]+)/le $1/;
print "$s\n";                    # Affiche : « le chat mange »
```

- Récupérer les sous-chaînes en correspondance :

On peut se placer dans un contexte de liste. Le résultat de l'analyse est alors la liste des variables $1, $2, …

Exemple : récupérer les 2 premiers chiffres consécutifs apparaissant dans une chaîne :

```
$s = "ab3z!!27";
($x,$y) = ($s =~ /(\d)(\d)/);          # $x vaut 2, $y vaut 7
```

Les variables $1, $2, … continuent à exister après l'analyse.

```
$s = "ab3z!!27";
$s =~ /(\d)/;
$x = $1;                               # $x vaut 3
```

$& vaut toute la sous-chaîne *matchant*.
$` vaut toute la sous-chaîne qui précède la chaîne *matchant*.
$' vaut toute la sous-chaîne suivant la chaîne *matchant*.
Exemple :

```
$s = "ba aa & cde";
if ($s =~ /(a+) & ([a-z])/)
{
        print "$1\n";                 # Affiche « aa »
        print "$2\n";                 # Affiche « c »
        print "$&\n";                 # Affiche « aa & c »
        print "$`\n";                 # Affiche «  ba  » (ba suivi d'un espace)
        print "$'\n";                 # Affiche «  de »
}
```

Remarques :

- Dans une expression régulière, on protège les symboles spéciaux par le caractère d'échappement « \ » si on veut les utiliser tels quels : « \. », « \[», « \/ », « \(»etc.
- On dispose des symboles spéciaux usuels :
  \n : saut de ligne
  \r : retour chariot
  \t : tabulation
- On dispose d'un opérateur permettant de tester si une chaîne n'est pas en correspondance avec une expression régulière : !~
  On a : if (!($s =~ /*regexp*/)) { … }
  ↔ if ($s !~ /*regexp*/) { … }