PERL

Pierre-François Bonnefoi

Qu‘est ce que PERL ?

PERL signifie “Practical Extraction and Report Language”.
Que l’on pourrait (essayer de) traduire par « langage pratique d‘extraction et d’édition ».
Créé en 1986 par Larry Wall (ingénieur système).

C‘est :

- Un langage de programmation (il a les mêmes possibilités que C, de l’objet comme C++...)
- Un logiciel gratuit (que l’on peut se procurer sur Internet notamment)
- Un langage interprété :
  - pas de compilation (en interne : une phase de pré-compilation)
  - moins rapide qu‘un programme compilé (mais implémente les meilleurs algorithmes en interne)
  - chaque « script » nécessite d’avoir l‘interpréteur Perl sur la machine pour s’exécuter.

Pourquoi Perl est devenu populaire :

- portabilité : Perl existe sur la plupart des plateformes aujourd‘hui (Unix, VMS , Windows, Mac, Amiga, Atari ...)
- gratuité : disponible sur Internet
- simplicité : Quelques commandes permettent de faire ce qu’un programme de 500 lignes en C ou en Pascal faisait.
- robustesse : Pas d‘allocation mémoire à manipuler, chaînes, piles, noms de variables illimités...

« PERL est un langage qui vous aide à faire votre travail » *Larry Wall*
« Il est conçu pour simplifier les tâches faciles, sans rendre les tâches difficiles impossibles. »

Pourquoi Perl ?

Il possède de nombreuses qualités :

- optimisé pour traiter des fichiers textes (*il permet de générer et d'analyser des fichiers textes, autorise une organisation « humaine » de l'information*)
- idéal pour les tâches d'administration système (*il dispose de points d'accès vers toutes les fonctions et possibilités d'Unix*)
- possède une syntaxe proche du C, des shell Unix (*"Sh"*), des outils d'analyse (*"awk", "grep", "sed"*...)
- indépendant de la gestion mémoire des données (*peut traiter n'importe quelle taille de fichiers, de données*...)
- dispose d'un vaste choix de modules pré-écrits dans de nombreux domaines (*interrogation de base de donnée avec SQL, programmation web en CGI, réalisation d'interface graphique avec Tk, programmation réseau*...)
- autorise la réalisation rapide des programmes puissants (*prototypage*) pouvant tourner sur différentes machines
- s'adapte au style du programmeur (*différents moyens de faire un même travail suivant la sensibilité du programmeur*)
- manipule tout type de données (*du binaire à un texte organisé autour de « phrases »*).
- gére « facilement » tout type de fichier et les informations qui les concernent (*date, droit d'accès*...)
- lance des processus (*lancement d'une application, éventuellement en parallèle*)
- contrôle ces processus (*récupére les « sorties » et injecte les « entrées » d'un processus*)
- ...

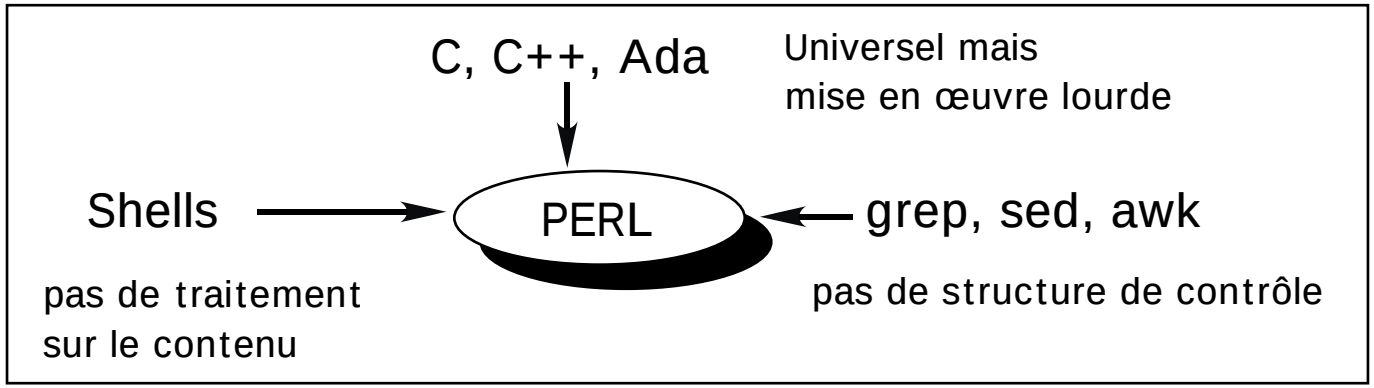
Schéma de conception de PERL

Langages de programmation (C, C++, ADA, Pascal...)
- Manipulation/Traitement de données complexes

Langages de commande (Sh, CSh, Bash, Ksh...)
- Manipulation de Fichiers/Répertoires
- Lancement de processus (Commande, Pipeline)

Utilitaires de conversion/Formatage (grep, sed, awk...)
- Conversion de chaînes (*"pattern matching"*)
- Traitement de Fichiers ligne à ligne ("*reporting*")

Fusion des fonctionnalités



Un programme de base

```
#!/usr/local/bin/perl
#
# programme tout bete
#
print ’Salut le monde.’; # Affiche un message
```

La première ligne

Tous les programmes en Perl commence avec cette ligne :
`#!/usr/local/bin/perl`

- Elle peut varier d’un système à un autre.
- Elle indique à la machine ce qu’elle doit faire quand le programme est exécuté (ie, elle lui indique d’exécuter le programme avec le Perl).

Commentaires et instructions

Les commentaires peuvent être insérés dans un programme en les précédant du symbole # :
- *tout ce qui suivra ce symbole jusquà la fin de la ligne sera ignoré*.
- *la seule manière d’étendre un commentaire sur plusieurs lignes est de commencer chaque ligne par #.*

Le reste correspond aux instructions en Perl, qui doivent se terminer par un point-virgule.

Un programme de base (suite et fin)

Simple affichage

La fonction print renvoie des informations.

- Elle affiche la chaîne de caractère “Salut le monde”,
- L’instruction se termine par un point-virgule.

Exécution du programme

Édition du source du programme avec un éditeur (Emacs par exemple) et création d’un fichier texte contenant ce source

Transformation du source en exécutable :

```
chmod u+x nom_du_programme
```

Exécution du programme obtenu :

```
perl nom_du_programme
./nom_du_programme
nom_du_programme
```

Exécution du programme en mode « avec alarmes » :

```
perl -w nom_du_programme
```

Exécution du programme en mode « déboguage » :

```
perl -d nom_du_programme
```

Données scalaires

- Chaînes de caractères : encadrées par “, ‘ (quote) ou `(back-quote)

fonctionnement similaire aux shells Unix :

  - dans une chaîne encadrées par deux “, les variables sont interpolées
  - dans une chaîne encadrées par deux ‘, les variables ne sont pas interpolées
  - dans une chaîne encadrées par deux `, le contenu de lachaîne est le résultat de l’exécution de la ligne de commande

- Nombres
  - Exemples : 123 123.45 123.45e10 0xffff (valeur héxadécimal) 0755 (valeur octale)
  - La conversion nombre vers chaîne et vice-versa est automatique et dépend des opérations appliquées

- Booléen Vrai ou Faux
  - Un scalaire est interprété à False dans un contexte booléen (un test par exemple) si c’est une chaîne vide ou le nombre 0 ou son équivalent en chaîne “0”.
  - Les autres valeurs sont considérées comme vraies !

- Valeurs defined ou undefined
  - Il existe deux types de scalaires nuls : defined et undefined. La fonction defined() peut être utilisée pour déterminé ce type
  - undefined correspond à un renvoi d’erreur (fin de fichier, variable non initialisée...).

Variables scalaires « $var »

Notation

Les scalaires sont précédés du caractère $

```
$nom_de_variable
$i = 0; $c = 'a';
$mon_fruit_prefere = 'kiwi';
$racine_carree_de_2 = 1.41421;
$chaine = '100 grammes de $mon_fruit_prefere';
```

- initialisation par défaut : valeur *undefined* (“0” ou chaîne vide)

- pas de déclaration
  - attention au déboguage !
  - donc pas de type !
  - attention aux fautes d'orthographes !

- globales par défaut (la variable est visible dans tout le programme)

- les noms des variables sont composés de chiffres, lettres et souligné (_), mais ils ne doivent pas commencer par un chiffre, et la variable $_ est reservée.

- Perl est sensible à la casse (différence minuscule, majuscule),
  $a et $A sont deux variables distinctes.

Exemples

Chaînes

```
'coucou'                    # coucou
'c\'est pas gagné!'         # c'est pas gagné !
'A bientôt \n'              # A bientot \n
"A bientôt \n"              # A bientôt suivi d'un retour à la ligne
`ls *.back`                 # le résultat de l'exécution de la ligne de commande ls *.back
```

Interpolation de variables dans une chaîne

```
$var = "appréc";
$variable = "beau";

print "C'est $variable";       # c'est beau
print "C'est ${var}iable";     # c'est appréciable
print 'c'est $variable';       # c'est $variable
```

Interchangeabilité nombres ~ chaîne de caractères

```
$valeur = '9';
$calcul = $valeur * 6;
print "$calcul";          # affiche 54
```

Opérations et assignements

Pour les nombres :

```
$a = 1 + 2;       # Ajoute 1 à 2, et l'affecte à $a
$a = 3 - 4;       # Soustrait 4 à 3, et l'affecte à $a
$a = 5 * 6;       # Multiplie 5 et 6, et l'affecte à $a
$a = 7 / 8;       # Divise 7 par 8, et affecte 0,875 à $a
$a = 9 ** 10;     # Eleve 9 à la dixième puissance
$a = 5 % 2;       # Le reste de 5 divise par deux (division euclidienne)
++$a;             # Incrementation de $a, puis on retourne la valeur $a
$a++;             # On retourne la valeur $a puis incrementation de $a
--$a;             # Décrementation de $a, puis on retourne la valeur $a
$a--;             # On retourne la valeur $a puis décrementation de $a
```

Opérateurs logiques :

```
&&      # Et logique
||      # Ou logique
!       # Non logique

$a = $b || "10";   # affecte la valeur de $b à $a, si cette valeur est définie sinon la chaîne "10"
```

Opérations et assignements (suite et fin)

Pour les chaînes de caractères :

Concaténation :

```
$a = $b . $c;    # Concaténation de $b et $c
```

Répétition :

```
$a = $b x $c;    # $b repété $c fois
$succés = “hip ” x 3 . “hourra”; # donne “hip hip hip hourra”
```

Extraction :

```
$a = substr(“hourra”, 1, 2)      # donne “ou”
```

Position index :

```
$a = index(“hourra”, “ou”);      # donne 1
```

Conversion binaire :

```
$chaîne = pack(“cccc”, 65, 66, 67, 68);   # donne “ABCD”
```

Affectation de plusieurs lignes directement depuis le source du programme :

```
$str = <<IDENTIFICATEUR;
        une ligne,  une autre ligne...
IDENTIFICATEUR
```

Comment assigner une valeur à une variable :

```
$a = $b;      # Assigne $b à $a
$a += $b;     # Ajoute $b à $a
$a -= $b;     # Soustrait $b à $a
$a .= $b;     # Contatene $b à $a
```

Listes et tableaux @

Représentation littéral

```
(1, 2, 3)          # la liste des éléments 1, 2 et 3
(“coucou”, 4.5)    # une liste peut contenir différents types d’éléments
(1..3)             # équivalent à (1, 2, 3)
```

Le .. signifie de tant à tant

```
(3.3 .. 6.1)       # équivalent à (3.3, 4.3, 5.3)
()                 # la liste vide
($var, “mot”)      # la liste composée du contenu de la variable var et de “mot”
```

Les variables de type tableaux

Notation

@nom_du_tableau

Exemples

```
@fruits  = (“pommes”, “poires”, “cerises”);
@musique = (“pipo”, “violon”);
@alphabet = (‘a’..‘z’)       ;
@cartes = (‘01’..‘10’,‘Valet’,‘Dame’,‘Roi’);
```

Il est possible d’affecter un tableau à un autre

```
@A = @B;
@tab = (1, 2, 3);
@TAB = (4, 5, 6);
@chiffres = (0, @tab, @TAB, 7, 8, 9);
```

Accés aux éléments d'un tableau @

Par l'affectation :

```
@tab = (1, 2, 3);
($a, $b, $c) = @tab;    # $a vaut 1, $b vaut 2 et $c vaut 3
($a, @TAB) = @tab       # $a vaut 1 et @TAB vaut (2, 3)
($a, $b) = ($b, $a)     # et hop une permutation !
```

Par l'indice : *Comme en C, l'indice commence à 0 !*

```
@tab = (1, 2, 3);
$a = $tab[0];      # $a contient 1
```

Attention quand on accéde à un tableau on manipule un scalaire d'où le $ au lieu du @ !

```
$b = $tab[$a];    # $b contient 2
$d = $tab[10];    # $d contient undefined
```

On peut sélectionner des tranches de tableaux

```
@nouveau_tableau = @tab[0, 2]; # @nouveau_tableau contient (1, 3)
@tableau = @tab[0 .. 2];      # @tableau contient (1, 2, 3)
```

Bornes du tableau :

```
$#tab représente l'indice supérieur du tableau @tab
$[tab représente lindice inférieur du trableau @tab
$nb_element = $#tab - $[tab - 1;
$nb_element = scalar (@tab);
```

Modification et accés aux tableaux @

Opérateurs push() et pop () : *Le tableau vu comme une “pile”*

```
@tab = (1, 2);
$a = 3;
@tab = (@tab, $a);      # ajoute le contenu de la variable $a au tableau @tab
push (@tab, $a);        # réalise la même opération : ajout par la droite du tableau
push (@tab, 4, 5, 6);   # @tab = (1, 2, 3, 4, 5, 6);
$b = pop ( @tab );      # supprime l'élément le plus à droite du tableau
                        # @tab = (1, 2, 3, 4, 5) et $b = 6
```

Opérateur shift() et unshift() : ajout et suppression par la gauche

```
@tab = (2, 3);
$a = 1;
unshift(@tab, $a);      # @tab = (1, 2, 3)
unshift(@tab, -1, 0);   # @tab = (-1, 0, 1, 2, 3)
$b = shift(@tab);       # @tab = (0, 1, 2, 3)  et $b = -1
```

Inversion et tri d'un tableau

```
@tab = (1, 3, 5, 0, 2, 4);
@tab = reverse(@tab);       # @tab = (4, 2, 0, 5, 3, 1)
@tab = sort(@tab);          # @tab = (0, 1, 2, 3, 4, 5)
@tabstr = (“petit, “moyen”, “grand”);  # @tabstr = (“grand”, “moyen”, “petit”)
```

Conversion d'un tableau en chaîne

```
print @tabstr;      # affiche “grand moyen petit”
```

Structure de contrôle : les tests

Les opérateurs de test reprennent la syntaxe du C :

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

Attention aux différences de traitement entre chaînes de caractères et nombre :

Nombres	Chaînes	Signification	
<	lt	Inférieur à	
<=	le	Inférieur ou égal à	
>	gt	Supérieur à	
>=	ge	Supérieur ou égal à	
==	eq	Égal à	
!=	ne	Différent de	
<=>	cmp	Comparaison	*renvoie -1, 0, ou 1 selon que le premier argument est inférieur, égal ou supérieur au second*

```
$a == $b        # Est-ce que $a est numericalement égal à $b?
                 # Attention !! N'utilisez jamais l'opérateur = (affectation).
$a != $b        #  Est-ce que $a est numéricalement différent de  $b?
$a ne $b        #  Est-ce que la chaîne $a est différent à la chaîne $b?
```

On dispose aussi des opérateur logiques et, ou et non :

```
($a && $b)          # Est-ce que $a et $b sont vrai?
($a || $b)      # Est-ce que $a ou $b est vrai?
!($a)           # Est-ce que $a est faux?
```

Structures de contrôle : les conditions

Commandes simples

- Chaque commande doit être terminée par un point-virgule
- Elle peut être éventuellement suivie d'un "modifier" :

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

Exemple : print "Test réussi\n" if ($var == 1);

Conditions implicites

Dans une instruction Perl :

```
$fichier = $ARGV[0] || "defaut.txt";   # l'opérateur « || » n'évalue que ce qui est nécessaire
                                       (l'évaluation s'arrête à la première valeur évaluée à True)

Is_Windows_running() && die "Please, execute me on a REAL OS...\n"; # l'opérateur « && »
                                                                    s'arrête à la première valeur évaluée à False
```

Conditions explicites

```
if (EXPR) BLOCK                          # BLOCK est une séquence d'instructions délimitée par {}
if (EXPR) BLOCK else BLOCK               # les accolades sont obligatoires même dans le cas
if (EXPR) elsif (EXPR) BLOCK ... else BLOCK      # d'une seule instruction

unless (EXPR) BLOCK                  # exécute le block si l'expression est fausse
unless (EXPR) BLOCK else BLOCK
```

Structures de contrôle : les boucles

La boucle foreach (pour chaque élément dans ensemble)

```
foreach $i (@liste)
{
  print $i;    # affiche chaque élément du tableau @liste
}
```

Exemple :

```
@val = (2, 5, 9);
foreach $n ( @val)
{
# $n référence chaque valeur de @val
$n *= 3;
}  # on a maintenant @val = (6, 15, 27)
```

La boucle for (comme en C)

```
for(init_expression ; test_expression ;  incrément_expression)
{
#corps de la boucle
}
```

Exemple :

```
for( $i = 0; $i < 10; $i++)
{
print $tab[$i];
}
```

Structures de contrôle : les boucles (suite et fin)

La boucle while, until et do...while

```
while (EXPR)
{
  # à évaluer tant que l'expression est vraie
}
until (EXPR)
{
  # à évaluer tant que l'expression est fausse
}
```

```
do
{
        # à évaluer tant que l'expression est
        # vraie et au moins une fois
}
while (EXPR)
```

Ruptures de séquence : last, next et redo

```
while ( exp1 ) {
inst1;
if (exp2) {
        inst2;
        last; # sortie
        }
inst3;
}
# reprise du last
```

```
while ( exp1 ) {
inst1;
if ( exp2 ) {
        inst2;
        next; # ocurrence suivante
              # du while
        }
inst3;
# reprise du next
}
```

```
while ( exp1 ) {
# reprise du redo
inst1;
if ( exp2 ) {
        inst2;
        redo; # recommence le while
              # sans évaluer exp1 à
              # nouveau
        }
inst3;
}
```

Tableaux associatifs % ou tables de « hachages »

Ce sont des tableaux dont l'indice n'est plus limité à un entier positif.
Les éléments peuvent être indexés par des chaînes de caractères.

```
%ages = (   "Michel Dupont", 39,        # association de “Michel Dupont” à 39
            "Larry Wall", 34,           # association de “Larry Wall” à 34
            "Néo", 27,
            "Michel durand", "21 ans depuis deux jours" );

%tab = ( “coucou”, “ça va”, 123.4, 567);
```

Accés aux éléments du tableau associatif

```
$tab{“salut”} = “ tout va bien ?”; # association de “ tout va bien ?” à l'index “salut”
$tab{“coucou”} .= “ ?”;           # concaténation de “ ?” à la valeur associé à l'index “coucou”
$tab{123.4} += 433;               # ajout de 433 à la valeur associée à l'index 123.4
print $tab{“hello”};              # undefined
%copie_de_tab = %tab;
```

Tableaux associatifs prédéfinis

```
$homedir = $ENV{‘HOME’};  # permet d'avoir accès aux variables d'environnement du processus
$SIG{‘UP’} = ‘IGNORE’;    # permet d'ignorer le traitement d'un signal
```

Tableaux associatifs (Suite et fin)

Opérateur keys()

```
@clés = keys(%tableau_associatif);      # récupére un tableau contenant toutes les clés d'indexation
```

Opérateur values()

```
@valeurs = values(%tableau_associatif);      # récupére les valeurs de chaque association
```

Opérateur each()

```
while ( ($cle, $valeur) = each(%tab))   # permet de passer en revue les différentes associations
{
   print $cle.“ “.$valeur;
}
```

Opérateur delete()

```
%tab = (”coucou“ => “ça va ?”, 123.4 => 567);  # => et synonyme de la virgule
@element = delete ($tab{“coucou”});
        # @element = (“coucou”, “ça va ?”); et %tab = (123.4, 567);
%tab = (%tab, “une_cle”, “une_valeur);      #ajout d'une association à %tab
```

Entrées/Sorties standards

Trois descripteurs de fichiers standards :
STDIN (entrée courante), STDOUT (sortie courante), STDERR (sortie d'erreur)

L'impression sur la sortie standard

```
print "bonjour \n";      # sort par défaut sur STDOUT
```

L'impression sur la sortie d'erreur

```
print STDERR "ceci est un message d\'erreur";
```

Lecture sur l'entrée standard

```
print "Entrez votre nom : ";
$nom = <STDIN>;        # lit un mot en provenance de l'interface d'entrée (clavier par exemple)
chomp($nom);           # permet de supprimer le caractère \n ajouté lors de la saisie
print "Hello $nom\n";
```

Opérateur <>

- dans un contexte scalaire : lit les données jusqu'au prochain \n soit une ligne
- dans un contexte de tableau : lit toutes les lignes

```
@les_lignes = <STDIN>;
foreach $ligne (@les_lignes)
    { print "> $ligne"; }        # affiche toutes les lignes saisies en entrée
```

Entrées/Sorties sur fichier quelconque

Descripteur

DESCRIPTEUR en majuscule

Ouverture / Fermeture

```
open (DESCRIPTEUR,        “nom_fichier”);
open (MONFIC,             “fichier”);       # ouverture en lecture/écriture
open(MONFIC_LECTURE,      “< fichier”);     # ouverture en lecture seule
open(MONFIC_ECRITURE,     “> fichier”);     # ouverture en écriture seule
open(MONFIC_AJOUT,        “>> fichier”);    # ouverture en écriture à la suite
```

Toujours tester les opérations d’ouverture

```
open(MONFIC, “fichier”) || die (“Ne peut ouvrir le fichier”);
close(MONFIC);          # fermeture du fichier
```

Lecture

```
while($ligne = <MONFIC>)
      {
            $ligne contient une ligne du fichier
```

Écriture

```
print MONFIC “La valeur est $var\n”;
```

Les fonctions

Définition

```
sub nom_sous_routine {
  # des intructions Perl
}
```

Appel avec le caractère &

- sans paramètre :

```
&fonction_de_traitement ;
```

- avec paramètres :

```
&imprimer(“hello”, “world”);
```

Résultat de l’appel

Le résultat de l’appel d’une fonction est la valeur de la dernière expression évaluée

```
sub somme_a_b { $a + $b; }
$a = 2; $b = 3; $c = &somme_a_b;     # $c vaut 5
```

Tableau de paramètres @_

```
sub somme2 { $_[0] + $_[1]; }
$c = &somme2 ($a, $b);
```

Protection des variables locales à une fonction : my()

```
my($var_locale) = 10;        # la variable $var_locale n’est connue que dans la fonction
```

Les expressions régulières

Une des caractèristiques les plus intéressantes de Perl, qui le rend adapté aux traitements 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 chaîne correspond à un motif
- afin d'extraire des informations
- afin d'y effectuer des substitutions

Méta-caractères

^	début de chaîne
$	fin de chaîne
.	n'importe quel caractère excepté newline \n
|	alternative
()	groupement et mémorisation
[]	classe de caractère

Pour utiliser un méta-caractère, le faire précéder de \

Quantificateur

*	apparaît 0, 1 ou plusieurs fois
+	apparaît 1 ou plusieurs fois
?	apparaît 0 ou 1 fois
{n}	apparaît exactement n fois

Caractères spéciaux

\t	tabulation	\n	newline	\r	retour-chariot	\s	espace
\w	lettre	\W	non-lettre	\d	chiffre	\D	non-chiffre

\S non-espace

Exemples

[abc] *un des caractères* [a-z] *une des minuscules* coucou | salut *l'un ou l'autre*
^\d$ *des chiffres du début à la fin, soit un nombre*

Les expressions régulières : recherche de motif

Les expressions régulières permettent de rechercher un motif dans une chaîne

Opérateur =~ (contient) et !~ (ne contient pas)

```
if ($a =~ /pat/)              # vrai si la chaîne $a contient le mot “pat”
if ($ligne =~ /^Données/)     # vrai si la chaîne $ligne commence par “Donnés”
```

Paramètres

g	recherche globale
i	ne pas tenir compte de la casse des caractères (pas de différence minuscule/majuscule)
etc	

Exemple

```
if($var =~ /login/i)      # vrai si $var contient login ou Login ou lOGin ou...
```

Les expressions régulières : substition et translation

Les expressions régulières permettent de substituer des caractères par d'autres dans une chaîne

Syntaxe :

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

Exemple :

```
$ligne =~ s/laco1/alpha1/g;     # indique de substituer la chaîne laco1 par alpha1 de manière globale
$texte =~ s/paris/Paris/gi;
```

Remarques :

Si le paramètre chaîne est omis il y a seulement suppression du motif trouvé

Les expressions régulière permettent de traduire des caractères par d'autres dans une chaîne

Syntaxe :

```
$var =~ tr/liste1/liste2/option;
```

Exemple :

```
$ligne =~ tr/a-z/A-Z/;      # passe les caractères a à z en majuscule
```

Les expressions régulières : mémorisation

Les expressions régulière permettent de mémoriser des suites de caractères dans un motif

Accés aux caractères précédant le motif recherché

la variable spéciale $` (dollar quote)

Accés aux caractères composant le motif recherché

la variable spéciale $&

Accés aux caractères suivant le motif recherché

la variable spéciale $‘ (dollar back-quote)

Éclatement et collage

Il est possible d'éclater une chaîne en sous-parties, stockées ensuite dans un tableau (éclatement).
Il est possible de joindre les éléments d'un tableau dans une chaîne (collage).

Opérateur split()

Exemple :

```
$informations = "Premier:Ministre:Acteur:14, rue Saint Honoré";
@personne = split(/:/, $informations);
```

ce qui a pour effet d'affecter à @personne :

```
@personne = ("Premier", "Ministre", "Acteur", "14, rue Saint Honoré");
```

Si on a préalablement affecté $informations à $_, on peut effectuer un appel plus simple :

```
@personne = split(/:/);
```

On peut utiliser les ER : par exemple, si on ne connait pas le nombre de colonnes, on peut écrire :

```
$_ = "Dupont:Jean::Boulanger:::Avenue des champs Elysées";
@personne = split(/:+/);
```

ce qui donnera

```
@personal = ("Dupont", "Jean",
                "Boulanger", "Avenue des champs Elysées");
```

L'opérateur join() réalise l'opération inverse.

Éclatement et collage (suite et fin)

Un paragraphe peut être séparé en phrase, une phrase en mots, et un mot en caractères.

```
@caracteres = split(//, $mot);          # séparation en caractères
@mots = split(/ /, $phrase);            # séparation suivant les espaces
@phrases = split(/\./, $paragraphe);    # séparation suivant les points
```

Les éléments d'un tableau peuvent être réunis au sein d'une chaîne de caractères

```
$chaine = join (" ", @tableau);
```

Gestion de Processus

system() permet d'exécuter une ligne de commande

```
system("date");
system("date >ficdate");
system("more /etc/passwd");
```

back-quote ``

```
$date = `date`;
foreach $fichier (`ls *.back`) {
  print "fichier backup : $fichier"; }
```

Pipeline associe la sortie d'un processus à l'entrée d'un autre

```
open (FINGER, " finger |");          # lecture du pipe
@finger = <FINGER>;
open (LPR, "| lpr");                 # écriture du pipe
print LPR $rapport;
```

Fork scission d'un processus en deux processus distincts

```
if (fork) { # je suis le fils
} else { # je suis le pere
        }
```

Utilisation du réseau : conception d'un client

Obtention de l'adresse Internet d'une machine

```
$targethost = "alphainfo.unilim.fr";
$internet_addresse = gethostbyname($target_host) or die "hote absent: $target_host";
$internet_addresse = pack('C4', 193, 50, 185, 1);      # adresse donnée sous forme numérique
```

Configuration d'une socket de communication

```
$protocole = getprotobyname('tcp');   # sélection du protocole TCP
$destination = pack('S n a4 x8', AF_INET, $target_port, $internet_addresse); # package final de
                                                # toutes les informations nécessaires à la connexion

socket(SOCK, AF_INET, SOCK_STREAM, $protocole) or die "socket: $!";      # création du socket
connect(SOCK, $destination) or die "connect: $!";       # tentative de connection
setsockopt(SOCK, SOL_SOCKET, SO_REUSEADDR, 1);  # configuration relative au système

autoflush SOCK, 1;      # on ne désire pas de buffer sur la socket

foreach $ligne (@lignes)
{
   print SOCK $ligne;    # transmission des données en direction du serveur
}
```

Utilisation du réseau : conception d'un serveur

Configuration du port de communication

```
$port_comm_serveur = 8080;     # port de communication n° 8080
$protocole = getprotobyname('tcp');   # sélection du protocole TCP
socket(SERVEUR, AF_INET, SOCK_STREAM, $protocole) || die "Ouverture socket: $!";
setsockopt(SERVEUR, SOL_SOCKET, SO_REUSEADDR,1)) || die "Configuration socket: $!";
                    # configuration relative au système
bind(SERVEUR, sockaddr_in($port_comm_serveur, INADDR_ANY)) || die "Accrochage socket: $!";
                                # accrochage de la socket au numéro de port sélectionné
listen(SERVEUR, SOMAXCONN) || die "Ecoute socket: $!";      # configuration de l'écoute

while($socket_courante = accept(CLIENT, SERVEUR))   # attente de connection d'un client
{
   ($port_comm_client, $iaddr) = sockaddr_in($socket_courante); # infos relatives au client
   $pid = fork;            # Création d'un processus enfant
   if ($pid) {             # le père attend que l'enfant est terminé de gérer la requête
      wait; close CLIENT;
   }
   else
   {
      open STDIN, "<&CLIENT";     # accrochage des E/S standards de l'enfant sur la socket
      open STDOUT, ">&CLIENT";
      exec "GestionRequete" || die "exec: $!";       # lancement de "GestionRequete" avec STDIN et
   }                                                 # STDOUT redirigés vers la socket
}
```