Systèmes d'exploitation

Shell et Perl

Bruno BEAUFILS

Université Lille 1

Année 2014/2015

Licence Professionnelle

da2i

Plan

Perl

1. Introduction

Quoi, pourquoi ?
Objets manipulés, variables
Opérateurs
Les structures de contrôle
Hachages
Entrées/Sorties
Expressions rationnelles – Quelques exemples

Qu'est-ce que Perl ?

- Langage de programmation **très souple**
- Combine des fonctionnalités de `sh`, `sed` et `awk`
- Syntaxe et idées de *Perl* empruntées à `sh`, `sed`, `awk`, `C`, `C++`, `BASIC/PLUS`.
- Créé par Larry Wall.
- Initialement crée pour traiter des fichiers `log` pour extraire des données, produire des rapports puis enrichissement et diffusion sur Internet.
- Grande facilité pour traiter les fichiers `log`
  $\Rightarrow$ succès auprès des administrateurs système dès 1989.
- Aujourd'hui employé dans de nombreux domaines par tous ceux qui doivent manipuler ou analyser rapidement de nombreuses données, que ce soit des pages Web ou des séquences d'ADN.

Pourquoi l'apprendre ?

- *Perl* est un outil polyvalent (un seul outil au lieu de plusieurs avec des syntaxes différentes).
- *Perl* est portable (Unix, Windows, Mac, ...).
- Par rapport au shell :
  - langage *semi-compilé*, plus rapide que les scripts shell ;
  - dispose d'un débogueur interactif ;
  - évite l'utilisation coûteuse des pipelines et commandes Unix
- Très bonne interface avec les systèmes facilitant l'écriture de démons, surtout pour les applications réseaux.
- En vrac : paquetages, modularité, peut traiter des fichiers textes ou binaires, pas de limitations arbitraires sur la taille des données, ... et ...langage objet !

Pourquoi faire ?

- Traitement de fichiers (optimisé pour fichiers texte mais peut manipuler fichiers binaires).
- Formatage de données.
- Écriture de démons.
- Maquettes de projets avant d'utiliser un langage compilé.
- Écriture de scripts (tout ce qui peut être fait avec un script shell).
- cgi
- ...

La richesse de *Perl* fait qu'il est toujours possible de traiter un problème de plusieurs façons d'où le slogan :

There's more than one way to do it.

Comment exécuter un programme *Perl* ?

- Écrire un script dont la première ligne est

  `#!/usr/bin/perl`,

  le rendre exécutable et . . . l'exécuter.
- Lancer *Perl* avec comme argument le script à exécuter et ses arguments :

  `perl monScript arg1 arg2 ...`
- Tester un ensemble d'instructions sans écrire de script en utilisant l'option `-e` :
  - `perl -e 'instruction1; ... ; instructionN;'`

Programme *Perl*

Perl manipule des objets grâce à des instructions.

- Les instructions peuvent prendre les formes suivantes :
  - `expression ;`
  - `expression modifieur;`
  - `[label] bloc`
  - `structure_de_controle`
- Un bloc est une suite d'instructions entre accolades.
- Un programme est un ensemble d'instructions.
- Les commentaires commencent par un caractère **#** et se terminent à la fin de la ligne.

Exemple.

```
bash$ perl -e 'print "coucou\n";'
coucou
```

Les objets manipulés par *Perl*

Les objets peuvent être manipulés de plusieurs manières, par :

- les constantes ;
- les variables ;
- les références.

Perl ne dispose que de quelques types de données prédéfinis :

- les **scalaires** (une seule valeur simple) ;
- les **tableaux** (liste ordonnée de scalaires) ;
- les **hachages** aussi appelés tableaux associatifs (ensemble non ordonné de paires clef/valeur, la chaîne clef donne accès à la valeur scalaire associée).

N.B. Voir aussi les handles de fichiers, sous-programmes, typeglobs et formats (qui peuvent être considérés commes des types de données).

Les variables

- Les variables sont toujours de l'un des trois types de base.
- Elles ne sont pas déclarées, elles sont créées dynamiquement à la demande.
- Le nom d'une variable commence par une lettre ou un caractère '_' suivi de toute combinaison de lettres, chiffres et caractère '_'. La taille du nom est limitée à 255 caractères. Attention, *Perl* distingue les majuscules et les minuscules.
- Un préfixe est utilisé devant le nom pour indiquer le type de la variable :
  - `$nom` est un scalaire ;
  - `@nom` est un tableau ;
  - `%nom` est un hachage.

Les scalaires

Un scalaire contient toujours une seule valeur qui peut être un **nombre**, une **chaîne** ou une **référence**.

- Littéraux numériques :

123	entier
123.456	décimal
6.08E-2	notation scientifique
0xFFFF	hexadécimal
056	octal
4_456_345_544	entier avec séparateur

Les conversions nécessaires sont automatiques :

```
bash$ perl -e 'print 123e2 - 12300;'
0
```

Les scalaires (2)

- Littéraux chaînes de caractères :
  - délimités par des apostrophes (quotes), dans ce cas la chaîne est complètement protégée (excepté \\ et \' : qui permettent d'inclure un anti-slash ou une apostrophe dans la chaîne).
  - délimités par des guillemets, les anti-slashs et les variables (commençant par $ ou @) sont alors interprétés. Attention le caractère ‘ n'est PAS interprété.

  Il y a conversion automatique entre chaînes et nombres :

```
bash$ perl -e 'print 123 - "123", "\t", length(0xFF), "\n";'
0       3
bash$
```

- Valeurs booléennes :
  - `0`, `"0"`, `""` et `undef` valent **faux** ;
  - le reste vaut vrai.

Exemple

```
bash$  cat essai
#! /usr/bin/perl
$nb = 12;
$chaine='bonjour\'toto $nb \n';
printf $chaine;
$chaine="bonjour\'toto $nb \n";
printf $chaine;
$chaine="toto\@machin.fr \x41 D \ua \n";
printf $chaine;

bash$  essai
bonjour'toto $nb \nbonjour'toto 12
toto@machin.fr A D A
```

Les listes

Une liste est une collection de constantes scalaires de forme :

```
(element1, element2, ..., elementN)
```

- la taille de la liste peut varier de manière dynamique ;
- les différents éléments de la liste ne sont pas tous forcément de même nature ;
- la valeur d'une liste est son nombre d'éléments.

Exemple.

`(1,2,3)`	liste de trois éléments numériques
`('cd', '', 'ab')`	liste de trois chaînes
`()`	liste vide
`(1, ' ', "\n", 1e10)`	liste *mixte*

Contexte

Chaque opération *Perl* se déroule dans un contexte particulier. Le résultat d'une opération dépend du contexte dans lequel elle est évaluée. Il existe deux contextes principaux : scalaire ou liste.

```
bash$  cat essai
#! /usr/bin/perl
@liste=('a','b','c');
$l=('a','b','c');
print @liste, "\n", $l, "\n", scalar @liste, "\n";

bash$  essai
abc
c
3

bash$
```

Rq : les backquotes

De même qu'en shell, une commande Unix placée entre backquotes (caractère ‘) est exécutée et son résultat est renvoyé.

- L'interprétation de la chaîne entre backquotes est identique à celle d'une chaîne entre guillemets.
- Dans un contexte scalaire, le résultat est la sortie standard de la commande.
- Dans un contexte de liste, le résultat de la commande est découpé en lignes, chaque ligne formant un élément de la liste (selon la variable `$/`).

Variables : généralités

- Les variables sont crées dynamiquement à la demande. Un tableau grossit par ajout d'éléments.
- Une variable non définie s'évalue comme une chaîne vide (elle vaut donc faux).
- Les variables scalaires ne sont pas typées. En fonction du contexte, le contenu sera interprété comme une chaîne, un nombre ou un booléen.
- Le préfixe $, @, ou % est toujours nécessaire pour accéder à la variable.
- Il est possible d'avoir des variables de types de base différents de même nom.
- Il n'est pas nécessaire de mettre les variables chaînes entre guillemets pour protéger leur valeur (contrairement au shell).

Variables scalaires

- Modification et création par affectation : signe =.
  Exemples.
  `$n = 12;` nombre
  `$gn=5_294_967_295 ;` très grand nombre !
  `$nusers=‘who | wc -l‘;` exécution d’une commande
- Pour utiliser une variable :
  - `$nom;`
  - `${nom}` pour lever les ambiguités.

  Exemples.
  `${n}34` donne 1234
  `$n34` variable de nom n34

Variables tableaux

- Modification et création par affectation : signe =.
  Exemples.
  `@tab = ('el1', 'el2', 'el3');` affectation d'une liste
  `@tab = ();` liste vide
  `@tab = ‘ls‘;` les éléments sont les noms de fichiers renvoyés par `ls`.
- Dans un contexte de liste, `@tab` vaut la liste de tous les éléments composant le tableau.
- Dans un contexte scalaire, `@tab` vaut le nombre d'éléments du tableau.
- Entre guillemets, `@tab` vaut une chaîne de caractères formée de ses éléments séparés par des espaces.

Exemples

```
bash$  cat essai
#! /usr/bin/perl

@tab = ('a','b', 1, 'c','d');
print @tab, "\n";
print "@tab", "\n";
print @tab + 0, "\n";

bash$  essai
ab1cd
a b 1 c d
5

bash$
```

Accès aux éléments d'un tableau

Syntaxe :

```
$tableau[expression]
```

Attention, il s'agit du symbole `$` !!!

- Le premier élément est à l'indice `0`.
- Le dernier élément est à l'indice `$#tableau`.
- Les indices négatifs permettent de numéroter les éléments à partir de la fin.
- Lorsque l'on crée l'élément d'indice `n`, tous les éléments manquant d'indice compris entre `0` et `n-1` sont créés avec la valeur `""` ou `0`.

Exemples

```
bash$  cat essai
#! /usr/bin/perl
@jours = ('lu', 'ma', 'me', 'je', 've', 'sa', 'di');
print $#jours, "\n";
print $jours[5], " ", $jours[-2], "\n";
$jours[4] = "ven";
$jours[10] = "ind";
print $#jours, "\t@jours\n";
$indice=4;
print "$jours[$indice+1]\n";

bash$  essai
6
sa sa
10      lu ma me je ven sa di    ind
sa

bash$
```

Accès à un sous-ensemble du tableau

Syntaxe `@tableau[liste-indices]`.

- La liste peut être le résultat de l'évaluation d'une expression ou une liste constante (parenthèses facultatives).
- Le même indice peut apparaître plusieurs fois dans la liste d'indices.

Exemples.

```
#! /usr/bin/perl
@jours = ('lu', 'ma', 'me', 'je', 've', 'sa', 'di');
@conges = (2,5,6);
print "Jours de conge : ", "@jours[@conges]", "\n";
@absences = (1,3,1,1);
print "Absences : ", "@jours[@absences]", "\n";

bash$  essai
Jours de conge : me sa di
Absences : ma je ma ma
```

Manipulation de listes

- Chaque élément d'une liste est un scalaire.
- Lors de la construction d'une liste, chaque élément est une expression à évaluer :
  - si le résultat est un scalaire, il est ajouté à la liste ;
  - si le résultat est une liste, tous les éléments de la liste sont insérés dans la liste en construction.
- Une liste de variables peut être utilisée en partie gauche d'une affectation.

Exemples.

`($x,$y) = ($y, $x);`	échange les valeurs de x et y
`($x,$y) = (1,2,3,4);`	3 et 4 sont ignorés
`($x,$y) = (1);`	y vaut 0 ou ""
`($x,@reste) = (1,2,3,4);`	x vaut 1, reste (2,3,4)
`(@tout,$x) = (1,2,3,4);`	x vaut 0 ou ""

Tableaux et listes

- Dans toute expression qui requiert une liste, on peut utiliser un tableau.
- On peut considérer une liste comme un tableau temporaire.
- Attention :
  - si on affecte une liste à un scalaire, c'est le dernier élément de la liste qui est affecté ;
  - si on affecte un tableau à un scalaire, c'est le nombre d'éléments du tableau qui est affecté.

Exemples.

```
$dernier=('a','b','c');
print $dernier, "\n";                                    affiche c
@tab=('a','b','c');
$scal = @tab;
print $scal, "\n";                                       affiche 3
```

Exemples

```
bash$  cat essai
#! /usr/bin/perl

$elem = (1,2,3,4)[2];
@lignes3et5 = (`cat toto.txt`) [3,5];

print $elem,"\n";
print @lignes3et5;

bash$  essai
3
ddddddddddddddddddd
fffffffffffff

bash$
```

Variables prédéfinies

Il existe de nombreuses variables prédéfinies parmi lesquelles :

- `$0` nom du script ;
- `$_` argument par défaut ;
- `$$` numéro du processus ;
- `$<` uid réel du processus ;
- `$(` gid réel du processus ;
- `$?` code de retour du dernier " ou pipe ou appel système ;
- `$"` séparateur des éléments d'un tableau dans une chaîne entre guillemets ;

Variables prédéfinies (2)

- `$,` séparateur des arguments écrits par `print` (rien par défaut) ;
- `$.` numéro de la dernière ligne lue ;
- `@ARGV` arguments passés au script. Attention, `$ARGV[0]` est le premier argument et pas le nom du script (comme en C) ;
- `@_` arguments dans une fonction ou un sous-programme ;
- `%ENV` environnement ;
- ...

Consulter le manuel pour plus de détails.

Les opérateurs

- Certains opérateurs s'appliquent à des termes scalaires, dans ce cas les opérandes sont évaluées dans un contexte scalaire.
- Certains opérateurs s'appliquent à des termes listes, dans ce cas les opérandes sont évaluées dans un contexte de liste.
- Certains opérateurs sont mixtes, dans ce cas leur action dépend du contexte souhaité.

Quelques exemples d'opérateurs :

- incrémentation et décrémentation :
  - **++** et **−** ;
- traitement de bits :
  - **~** complément à un,
  - « et » décalages de bits gauche et droite,
  - **&** ***et*** bit à bit,
  - | et ^ ***ou*** et ***ou exclusif*** bit à bit ;

Opérateurs (2)

- comparaisons :
  - `<`, `>`, `<=`, `>=`, comparaisons de nombres,
  - `lt`, `gt`, `le`, `ge`, comparaisons de chaînes,
  - `==`, `!=`, `<=>` égalité de nombres,
  - `eq`, `ne`, `cmp`, égalité de chaînes ;
- opérateurs logiques :
  - `!`, `&&` et || négation, **et** et **ou** logiques,
  - `not`, `and`, `or` et `xor` négation, **et**, **ou** et **ou exclusif** logiques de priorité basse ;
- opérations arithmétiques :
  - `**` élévation à la puissance.
  - `+`, `-`, `*`, `/`, `%`, addition, soustraction, multiplication, division et modulo.
- opérateurs sur les chaînes :
  - `.` concaténation.

Opérateur de répétition

Syntaxe `expression-gauche x expression-droite`

- L'expression droite est évaluée en contexte scalaire.
- En contexte scalaire, l'expression de gauche est évaluée et son résultat est répété selon l'expression de droite (évaluée en contexte scalaire).
- En contexte de liste, si l'expression de gauche est une liste entre parenthèses, elle est répétée selon l'expression de droite (évaluée en contexte scalaire).

```
print '-' x 10;                                  affiche 10 tirets
print 'ab' x 3;                                  affiche ababab
@tab = 1 x 3;
print "@tab";                                     affiche 111
@tab = (1) x 3;
print "@tab";                                     affiche 1 1 1
@tab = (@tab) x 2;
print "@tab", "\n";                               1 1 1 1 1 1
```

Les modifieurs

Un modifieur est une forme que l'on peut ajouter à la fin d'une expression :

- le modifieur peut changer :
  - le fait que l'instruction va être exécutée ou non,
  - le nombre de fois où l'instruction va être exécutée ;
- un seul modifieur peut être utilisé, juste avant le ' ; ' ;
- les modifieurs possibles sont :
  - `if expression;`
  - `unless expression;`
  - `while expression;`
  - `until expression;`

Exemple

```
bash$  cat essai
#! /usr/bin/perl

$x = 2;
$y = 5;

print "x egal y \n" if $x == $y;
print "x different de y \n" if $x != $y;

print "x egal y \n" unless $x != $y;
print "x different de y \n" unless $x == $y;

bash$  essai
x different de y
x different de y
```

Instruction *if*

```
if (expression)
   BLOC
elsif (expression)
   BLOC
   ...
else
   BLOC
```

Les parties **elsif** et **else** sont facultatives. Comme les instructions sont forcément sous forme de BLOC (donc entre accolades), il n'y a pas d'ambiguité possible.
Exemple.

```
if ($x == $y) { print "x egal y \n";}
else { print "x different de y \n" ;}
```

Rq. On peut utiliser **unless** à la place de `if` dans une forme `if ...` ou `if ... else...`

Instruction *while*

```
LABEL: while (expression)
          BLOC
```

ou

```
LABEL: while (expression)
          BLOC1
       continue
          BLOC2
```

- Le label est optionnel, c'est une chaîne de caractères qui est, par convention en majuscules.
- Le `BLOC2` sera exécuté chaque fois que l'on termine le `BLOC1`, avant de recommencer l'évaluation de l'expression.
- **Rq.** On peut utiliser **`until`** à la place de `while`.

Exemple

```
bash$  cat essai
#! /usr/bin/perl
$i = 1;
until ($i == 3) {
    print "test until : bloc1 tour $i\n";
    $i = $i+1;
}
while ($i != 5) { print "test while : bloc1 tour $i\n";}
continue {print "test while : bloc2 tour $i\n"; $i = $i+1;}

bash$  essai
test until : bloc1 tour 1
test until : bloc1 tour 2
test while : bloc1 tour 3
test while : bloc2 tour 3
test while : bloc1 tour 4
test while : bloc2 tour 4
```

Instruction *for*

```
LABEL: for (expression; expression; expression)
           BLOC
```

- Les trois expressions correspondent respectivement à l'initialisation, la condition et la réinitialisation. Chacune est optionnelle.
- Les initialisation et réinitialisation peuvent concerner plusieurs variables (séparer les affectations ou autres par des virgules).
- Si la condition n'est pas présente, elle est considérée comme vraie.

Instruction *foreach*

```
LABEL: foreach var (liste)
          BLOC
```

- Parcourt la liste et assigne tour à tour chaque élément de la liste à la variable `var`.
- La variable est implicitement locale à la boucle, elle reprend sa valeur précédente à la sortie de la boucle.
- Si `liste` est un *vrai* tableau plutôt qu'une liste, chaque élément du tableau est modifiable par l'intermédiaire de la variable de boucle.
- Le bloc `continue` peut également être présent.

Exemples

```
bash$  cat essai
#! /usr/bin/perl
for ($i=0; $i<4; $i++)
   {@vect[$i] = $i+10;}
print "@vect\n";
foreach $nom (`ls`)
   {print $nom;}
@tab= (1,2,3,4);
foreach $elem (@tab)
   {$elem=$elem *2;}
print "@tab\n";

bash$  essai
10 11 12 13
fichier.txt
essai
2 4 6 8
```

Contrôle de boucles

Il existe des instructions permettant de rompre le flot *normal* des boucles :

- `last LABEL`
- `next LABEL`
- `redo LABEL`

- Le `LABEL` est falcutatif, dans ce cas, on se réfère à la boucle la plus interne.
- La commande **`last`** sort immédiatement de la boucle. Le bloc `continue`, s'il existe, n'est pas exécuté.
- La commande **`next`** permet de *sauter* à l'itération suivante. Le bloc `continue`, s'il existe, est exécuté avant que la condition ne soit réévaluée.
- La commande **`redo`** permet de redémarrer le bloc de boucle sans réévaluer la condition. Le bloc `continue`, s'il existe, n'est pas exécuté.

Exemples

Considérons le programme suivant en *programmation classique*.

```
bash$  cat essai
#! /usr/bin/perl
@tab1 = (2,0,5,8,5);
@tab2 = (3,0,2,10,5,12);
for ($i=0; $i < @tab1; $i++)
{
   $j=0;
   while ($j < @tab2 && $tab1[$i] <=  $tab2[$j]) {
       $tab1[$i] += $tab2[$j];
       $j++;
   }
}
print "@tab1\n";

bash$  essai
5 3 5 8 5
```

Exemples (2)

Le programme suivant est identique, dans un style *plus Perl*.

```
#! /usr/bin/perl
@tab1 = (2,0,5,8,5);
@tab2 = (3,0,2,10,5,12);

A: foreach $ceci (@tab1)
{
   B: foreach $cela (@tab2)
   {
       next A if $ceci > $cela;
       $ceci+=$cela;
   }
}
print "@tab1\n";
```

Les hachages

- Un hachage est un tableau associatif, c'est-à-dire un tableau dont les valeurs ne sont pas sélectionnées par des indices entiers mais par des indices quelconques.
- Les « indices » sont nommés « clés » et sont choisis par le programmeur.
- L'ordre interne des données n'est pas connu du programmeur, il permet de ne pas parcourir toute la liste pour accéder à une valeur. Y toucher risque de limiter l'efficacité d'une telle structure.

Accès aux éléments

- Une variable hachage est toujours préfixée par le symbole %.
- Chaque élément d'un hachage est un scalaire (comme pour les tableaux).
- On accède à un élément par une clé qui est également un scalaire (comme dans le cas des tableaux, l'élément étant un scalaire, l'accès se fait avec un symbole $).
- Comme pour les tableaux, on crée de nouveaux éléments en leur affectant une valeur.

Exemples.

```
$mon_hachage{"aaa"} = 3;                          clé aaa, valeur 3
$mon_hachage{3.14} = "Pi";                        clé 3.14, valeur Pi
$i = "toto";
$mon_hachage{$i} = 4;                             clé toto, valeur 4
print $mon_hachage{"toto"}, "\n";                 affiche 4
```

Variables et littéraux

Exemples.

```
print %mon_hachage, "\n";
```
affiche `aaa3toto43.14Pi`

```
@mon_tab = %mon_hachage;
print "@mon_tab", "\n";
```
affiche `aaa 3 toto 4 3.14 Pi`

- Il n'existe pas de « véritable » représentation littérale d'un hachage.
- Un hachage est représenté par une liste qui se lit de gauche à droite comme une suite de paires clé/valeur.
- On peut obtenir une copie d'un hachage par affectation.
- On peut inverser les rôles clé/valeur en utilisant l'opérateur reverse.

Exemples

```
bash$ cat essai
#!/usr/bin/perl
$mon_hachage{"aaa"} = 3;
$mon_hachage{3.14} = "Pi";
$mon_hachage{"toto"} = 4;

%copie = %mon_hachage;
print %copie, "\n";
%copie = reverse %copie;
print %copie, "\n";
print $copie{"Pi"},"\n";

bash$ essai
aaa33.14Pitoto4
Pi3.143aaa4toto
3.14

bash$
```

Fonction *keys*

- Fournit la liste de toutes les clés utilisées pour un hachage.
- Dans un contexte scalaire, fournit le nombre d'éléments du hachage (comme le hachage d'ailleurs).

```
bash$ cat essai
$hach{"aaa"} = 3;
$hach{3.14} = "Pi";
$hach{"toto"} = 4;
if (%hach){
    foreach $cle (keys(%hach)){
        print "cle : $cle, valeur : $hach{$cle}\n";
    }
}
else {print "Aucun element\n";}

bash$  essai
cle : aaa, valeur : 3
cle : toto, valeur : 4
cle : 3.14, valeur : Pi
```

Autres fonctions sur les hachages

- La fonction `values(%hach)` renvoie la liste de toutes les valeurs contenues dans le hachage (dans l'ordre correspondant à celui des clés renvoyées par `keys`).
- La fonction `delete($hach{$cle})` permet de supprimer un élément du hachage.
- La fonction `each(%hach)` permet d'accéder successivement (à chaque appel) à tous les couples clé/valeur du hachage sous forme de liste. Elle renvoie la liste vide lorsqu'il n'existe plus de couple.
- Il est possible d'accéder à une *tranche* de hachage : `@hach{$cle1, $cle2, $cle3}` représente la tranche du hachage `hach` correspondant aux clés citées.

Exemples

```
bash$ cat essai
#!/usr/bin/perl
%hach = ("cle1", 1, "cle2", 2, "cle3", 3, "cle4", 4);
delete $hach{"cle2"};
while (($cle,$valeur)=each (%hach)) {
     print "cle : $cle, valeur : $valeur\n";
}
@nouveau{"cle5","cle6","cle7"} = (5,6,7);
print $nouveau{"cle6"},"\n";
@nouveau{keys %hach} = values %hach;
print "@nouveau{keys %nouveau}","\n";

bash$  essai
cle : cle4, valeur : 4
cle : cle1, valeur : 1
cle : cle3, valeur : 3
6
4 5 6 7 1 3
```

Ouvrir un fichier

- Trois possibilités :
  - `open(MONHANDLE, "fichier.txt")` ouvre le fichier `fichier.txt` en **lecture** et lui associe le handle `MONHANDLE`.
  - `open(MONHANDLE, ">fichier.txt")` ouvre le fichier `fichier.txt` en **écriture** et lui associe le handle `MONHANDLE`.
  - `open(MONHANDLE, ">>fichier.txt")` ouvre le fichier `fichier.txt` en **ajout** et lui associe le handle `MONHANDLE`.
- Les trois retournent « vrai » en cas de succès et « faux » en cas d'échec.
- Pour fermer le fichier : `close(MONHANDLE)` (qui peut également échouer).

Lire un fichier

- Le fichier se lit ligne par ligne en utilisant le handle obtenu à l'ouverture entre les symboles <> (opérateur d'entrée).
- En contexte scalaire, à chaque évaluation, la ligne suivante du fichier est retournée, `undef` lorsqu'il n'y a plus de ligne disponible.
- En contexte de liste, retourne toutes les lignes restantes.
- Chaque fois qu'un test de boucle ne dépend que de l'opérateur d'entrée, la ligne lue est automatiquement affectée à la variable `$_`.
- Le handle de l'entrée standard est `STDIN`.

Exemples : équivalent de la commande `cat fichier`.

```
#!/usr/bin/perl
open(HAND, $ARGV[0]);
while (<HAND>) {print;}
```

Fonctions *chop et* chomp

Deux fonctions de manipulations des chaînes peuvent être utiles lors de la lecture d'un fichier :

- `chop($chaine)`
  - supprime le dernier caractère de la chaîne,
  - elle retourne le caractère supprimé,
  - si la chaîne est vide, la fonction ne retourne rien ;
- `chomp($chaine)`
  - si la chaîne se termine par un caractère `'\n'`, la fonction le supprime,
  - si la fonction a supprimé un caractère `'\n'`, elle retourne `1`, sinon, elle retourne `0`.

Écrire dans un fichier

Il suffit pour écrire dans un fichier de :

- de disposer d'un handle ouvert en écriture ou en ajout ;
- d'utiliser la fonction `print` avec, en premier argument, le handle du fichier.

Les handles de la sortie standard et de la sortie standard d'erreur dont `STDOUT` et `STDERR`.

Exemples : copie d'un fichier.

```
#!/usr/bin/perl
open(DE, "toto.txt");
open(VERS, ">toto2.txt");
while (<DE>) {
    print VERS;
}
close(DE);
close(VERS);
```

Fonction *die*

S'il n'est pas possible d'ouvrir un fichier (ou de le fermer), aucune erreur ne survient lorsque l'on tente de le lire ou d'y écrire. Il faut donc vérifier les retours des fonctions ou utiliser `die` :

- cette fonction prend une liste et l'envoie sur la sortie standard (comme `print`) ;
- si le message à afficher ne se termine pas par un `'\n'`, la fonction lui ajoute le nom et le numéro de ligne du fichier où s'est produite la sortie par `die` ;
- la fonction termine le processus (avec un code de retour non nul) ;
- il est également possible d'utiliser la variable `$!` qui contient le message d'erreur du dernier appel système.

Exemple

```
bash$ cat essai
#!/usr/bin/perl
$de = "t1.txt";
$vers = "t2.txt";
open(DE, $de) || die "Echec open $de : $!";
open(VERS, ">$vers") || die "Echec open $vers : $!";

while (<DE>) {
    print VERS;
}
close(DE) || die "Echec close $de : $!";
close(VERS)|| die "Echec close $vers : $!";

bash$  essai
Echec open t1.txt : No such file or directory at essai line 4.
```

Opérateur <>

- L'opérateur diamant **<>** fonctionne comme `<HANDLE>`.
- Il lit les données depuis les fichiers dont les noms figurent dans le tableau `@ARGV`.

Exemples.
Équivalent de la commande `cat f1... fN`.

```
#!/usr/bin/perl
while (<>) {
   print;
}
```

Afficher les lignes du fichier passé en argument en ordre inverse.

```
#!/usr/bin/perl
print reverse (<>);
```

Opérateurs de test sur les fichiers

Opérateur	Test
`-r`	en lecture par iud/gid effectifs
`-w`	en écriture par iud/gid effectifs
`-x`	exécutable par iud/gid effectifs
`-o`	appartenant à iud/gid effectifs
`-R`	en lecture par iud/gid réels
`-W`	en écriture par iud/gid réels
`-X`	exécutable par iud/gid réels
`-O`	appartenant à iud/gid réels
`-e`	existence
`-z`	existence et taille égale à 0
`-s`	existence et taille différente de 0

Opérateurs de test sur les fichiers (2)

Opérateur	Test
`-f`	fichier régulier
`-d`	répertoire
`-l`	lien symbolique
`-S`	socket
`-p`	tube nommé
`-b`	fichier bloc
`-c`	fichier caractère
`-t`	fichier associé à un terminal
`-u`	setuid bit positionné
`-g`	setgid bit positionné
`-k`	sticky bit positionné

Opérateurs de test sur les fichiers (3)

Opérateur	Test
`-T`	fichier « texte »
`-B`	fichier « binaire »
`-M`	Nombre de jours depuis dernière modif
`-A`	Nombre de jours depuis dernier accès
`-C`	Nombre de jours depuis dernière modif de l'inode

Exemple. Afficher les noms des fichiers lisibles dans le répertoire courant.

```
#!/usr/bin/perl
foreach (‘ls‘) {
    chomp;
    print "$_\n" if -r;
}
```

Utilisation en Perl

- Les expressions rationnelles (telles que celles vue avec `sed`) peuvent être utilisées directement en *Perl*.
- Toutes les expressions rationnelles utilisées avec les différents outils UNIX tels de `sed`, `vi`, `emacs`, `grep`, ...peuvent également être décrites en *Perl*.
- L'utilisation des expressions rationnelles facilite la manipulation et la recherche dans les fichiers sans avoir besoin de faire appel à des commandes externes.

Exemple. Afficher toutes les lignes d'un fichier qui contiennent le mot `toto`.

```
#!/usr/bin/perl
while (<>) {
    if (/toto/) { print; }
}
```

Caractères

Pour désigner un caractère dans une expression rationnelle, il est possible d'utiliser :

- **le caractère** ;
- `.` pour tout caractère excepté `'\n'` ;
- `[azfltof]` pour un caractère dans l'ensemble ;
- `[0-9]` pour un caractère de l'intervalle ;
- `^` pour le complémentaire ;
- `\^` pour un caractère (ici `^`) ayant une signification spéciale ;
- `\d` pour un chiffre et `\D` pour un non chiffre ;
- `\w` pour un caractère de mot et `\W` pour un caractère ne pouvant se trouver dans un identificateur ;
- `\s` pour un espacement `\S` pour un non espacement.

Caractères – Exemples

`[0-9]`	chiffre
`[0-9ab]`	chiffre ou 'a' ou 'b'
`[a-zA-Z_]`	lettre ou souligné
`[^a-z]`	tout sauf une minuscule
`[\s,?\.;:!]`	espacement ou ponctuation
`[\da-fA-F]`	chiffre hexadécimal
`[^\^]`	tout sauf un ^

Plusieurs caractères

Pour désigner une suite de caractères dans une expression rationnelle, il est possible d'utiliser :

- **la suite** elle-même ;
- `*` pour zéro ou plusieurs fois le caractère qui précède ;
- `+` pour une ou plusieurs fois le caractère qui précède ;
- `?` pour zéro ou une fois le caractère qui précède ;
- `6` pour six fois le caractère qui précède ;
- `3,8` pour 3 à 8 fois le caractère qui précède ;
- `5,` pour 5 fois ou plus le caractère qui précède ;
- les précédents motifs se remplacent de gauche à droite en englobant le plus de caractères possibles (*gloutons*), il est possible de les rendre *paresseux* en les faisant suivre d'un point d'interrogation.

Plusieurs caractères – Exemples

Expression	Mots
`ab?c`	`ac, abc`
`ab*c`	`ac, abc, abbc, ...`
`ab+c`	`abc, abbc, ...`
`a.b*`	`anbbb, a!bbbbb, ...`
`a.+.*`	`abcdefg` ($.+$)
`a.+?.*`	`a b cdefg` ($.+$, $.*$)

Autres

- Mémorisation d'une partie entre parenthèse et utilisation par `\n` si l'expression est la $\texttt{n}^{\text{ème}}$ entre parenthèses dans l'expression.
  Exemple : `(.)(.)\2\1`, les palindromes de 4 lettres.
- Alternatives séparées par des |. Exemple : `toto|titi|tutu`, l'un des mots au choix.
- Ancres `\b` pour une limite de mot, `^` pour un début de ligne (si bien situé), `$` pour une fin de ligne (si bien situé), ...
  Exemple : `^toto\b`, ligne commençant par le mot `toto` n'étant pas préfixe d'un autre mot.
- L'utilisation de `i` après l'expression permet d'ignorer majuscules et minuscules.
- Les variables sont évaluées dans les expressions rationnelles.

Substitution

L'opérateur de substitution ressemble à celui de `sed`

`s/expr/chaine/gi`

- remplace la première occurrence du motif décrit par l'expression par la chaîne donnée ;
- l'option `g` permet d'appliquer la substitution à toutes les occurrences ;
- l'option `i` permet d'ignorer les majuscules/minuscules ;
- il est possible d'utiliser des variables.

Exemple.
`s/toto/titi/gi` remplace toutes les occurrences de `toto` (en majuscules ou minuscules) par `titi`.

Variables

Après une comparaison réussie, des variables spéciales permettent de récupérer des parties de la chaîne (sinon, les variables sont indéfinies) :

- `$1`, `$2`, ...contiennent les valeurs de `\1`, `\2`, ...
- `$&` contient la partie de la chaîne qui a correspondu à l'expression rationnelle ;
- `$‘` contient la partie de la chaîne qui précède la partie qui a correspondu à l'expression rationnelle ;
- `$’` contient la partie de la chaîne qui suit la partie qui a correspondu à l'expression rationnelle.

Ces variables peuvent également être utilisées dans les substitutions.
Exemple.
`s/([0-9]+)/($1)/g` mettre les nombres entre parenthèses.

Opérateur =~

Il est possible d'appliquer les comparaisons et les substitutions à d'autres cibles que la variable $_.

Exemples.

```
bash$  cat essai1
#!/usr/bin/perl
$chaine= "Bonjour toto";
if ($chaine =~ /.*(..)\1/)
     { print "chaine valide\n";}
else { print "chaine non valide\n";}

bash$  cat essai2
#!/usr/bin/perl
while ($a = <STDIN>)
{
    $a =~ s/([0-9]+)/($1)/g;
    print $a;
}
```

Opérateur *split*

- Il permet de découper une chaîne, par exemple pour en extraire les champs.
- Il prend en arguments une expression rationnelle et une chaîne et retourne les parties de la chaîne qui ne correspondent pas au motif de l'expression rationnelle.

Exemple.

```
bash$ cat essai
#!/usr/bin/perl
$a = "bonjour, il fait   beau.";
@mots = split(/[, .]+/,$a);
print "@mots\n";

bash$ essai
bonjour il fait beau
```

Opérateur *join*

- Il permet de « coller » les éléments d'une liste en les séparant par une chaîne donnée.
- Attention, la chaîne n'est pas une expression rationnelle !

Exemple.

```
bash$ cat essai
#!/usr/bin/perl

@noms = ("dupont", "durant", "carpentier");
$liste = join(",", @noms);
print "$liste\n";

bash$  essai
dupont,durant,carpentier
```