Le langage Perl

Emmanuel Daucé

1. Introduction

Qu'est-ce que PERL?

- P.E.R.L signifie « *Practical Extraction and Report language* » (langage pratique d'extraction et d'édition).
- Créé en 1986 par Larry Wall (ingénieur système). Il remplace des langages d'extraction plus anciens, en particulier awk.
- PERL est devenu un langage multi-usage. Ses applications vont bien au delà de l'extraction de textes.

A quoi PERL est-il adapté?

- A l'origine: langage de "réduction de données", conçu pour:
  - Naviguer dans les **fichiers,**
  - **Scruter** de grandes quantités de texte (recherche, substitution),
  - créer et utiliser des **structures de données dynamiques,**
  - **afficher** des rapports facilement formatés
- On l'utilise actuellement pour:
  - Génération de fichiers HTML (CGI)
  - Accès aux bases de données
  - Conversion de format de fichiers

Perl n’est adapté pour :

- Le calcul scientifique :
  - Perl n’est pas compilé => problèmes de performances.
  - Pas de gestion fine de la précision des valeurs numériques
- Le traitement de fichiers en mode binaire

Evolutions

- Le langage PERL est en passe de devenir le langage de référence pour la bioinformatique. Ses avantages :
  - Traitement des chaînes de caractères
  - Mise en réseau
  - Prototypage rapide
- Possibilité d'écrire des interfaces interactives (il existe un module Perl-Tk qui le permet)

Projets

- Les 3 dernières séances de ce cours seront consacrées à la mise en œuvre du langage PERL.
  - Projets à rendre par binômes

Comment créer un programme Perl?

- Un programme Perl est un fichier texte. Il peut être créé et modifié par n'importe quel éditeur de texte.
- Exemple :

```
#!/usr/bin/perl
print "Salut tout le monde \n";
```

- Si le fichier contenant le programme s'appelle `mon_programme.pl`, il faut rendre celui-ci exécutable comme suit :

```
chmod +x mon_programme.pl
```

- Il est désormais possible d'exécuter le programme comme suit :

```
./mon_programme.pl
```

Analyse rapide du programme

```
#!/usr/bin/perl  ①
# ceci est un commentaire normal ②
print "Salut tout le monde! \n" ; ③ ④ ⑤
```

1. La première ligne est un commentaire «spécial » qui indique à l'interpréteur shell d'interpréter les lignes suivante avec le programme perl, situé à l'emplacement /usr/bin/perl
2. Les commentaires Perl commencent par le signe **#**
3. Perl distingue majuscules et minuscules. **print** est un mot-clé connu, mais **Print** est inconnu.
4. Le caractère **\n** représente le passage à la ligne comme en C ou Java.
5. Les instructions se terminent par un point virgule **;**

Affichage du nombre de lignes d'un fichier

```
#!/bin/perl
open (F,'monfichier');   # ouverture d'un fichier en lecture
$i=0;                    # initialisation du compteur
while (<F>) {            # pour chaque ligne lue
   $i++;                 # incrémentation du compteur
}
close F;                 # fermeture du fichier
print "Nombre de lignes : $i" ;
                         # affichage du contenu du compteur
```

Autre exemple

```
#!/usr/bin/perl
@lines = `perldoc -u -f atan2` ;
foreach (@lines) {
  s/\w<([^>]+)>/\U$1/g;
  print;
}
```

2. Données scalaires

Valeur scalaire

- Les valeurs manipulées par un programme Perl peuvent être
  - de type **numérique** (entier, réel)
  - ou de type **texte** (on parle aussi de « chaîne de caractère »).

Les valeurs numériques (nombres)

- Les nombres s'expriment classiquement (comme en C):

```
12          # une valeur entière positive
+10         # une autre valeur entière positive
-34         # une valeur entière négative
+3.14       # un réel positif
5e15        # 5 fois 10 puissance 15
033         # 33 en code octal soit 27 en décimal
x1F         # 1F en hexa soit 31 en décimal
```

Les opérateurs arithmétiques

- Les opérateurs arithmétiques sont les suivants :

`+` (addition)	`-` (soustraction)
`*` (multiplication)	`**` (puissance)
`/` (division)	`%` (modulo)

Expressions numériques

- Une expression numérique est donnée par la combinaison de nombres et d'opérateurs, selon les règles de l'arithmétique.

```
2 + 3          # 2 plus 3, soit 5
5.1 - 2.4      # 5,1 moins 2,4, soit 2,7
3 * 12         # 3 fois 12, soit 36
14 / 2         # 14 divisé par 2, soit 7
10.2 / 0.3     # 10,2 divisé par 0,3, soit 34
10 / 3         # division réelle (et non
               # entière), soit 3,3333…
```

Les textes : chaînes de caractères

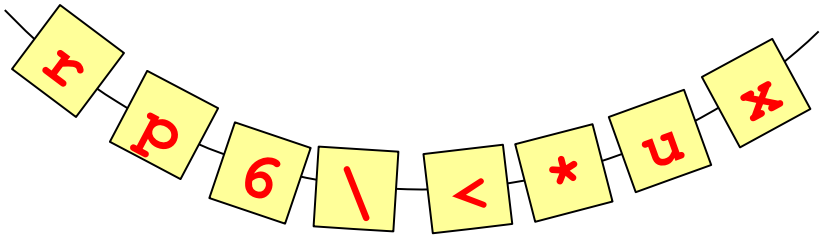
- Un caractère est un signe alphabétique, numérique ou de ponctuation tels qu'on les trouve sur le clavier d'un ordinateur : **`r`**, **`P`**, **`6`**, **`\`**, **`<`**, **`*`**, **`u`**, **`x`**, **`~`**,…
- A chaque caractère est associé un nombre compris entre 0 et 255 : son code ASCII.
- Les caractères « invisibles » sont désignés par un code (caractères d'échappement) :

  `\t` : tabulation

  `\n` : passage à la ligne

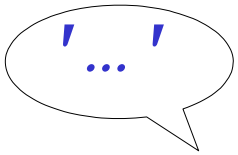
  `\a` : cloche

  etc…



- Un texte est représenté par une séquence de caractères (couramment appelée **chaîne de caractères**, ou *string* en anglais)

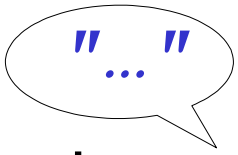
Représentation des chaînes de caractères : apostrophes simples



- Il est nécessaire de distinguer :
  - une portion de texte littérale
  - le texte du programme lui même
- Une première méthode consiste à encadrer la portion de texte par des apostrophes simples:

```
'john'          # Quatre caractères : j,o,h,n
'georges'       # Sept caractères
''              # Chaîne vide (aucun caractère)
'\\'            # Barre oblique inverse : \
                # (caractère d'échappement)
'l\'avion'      # Chaîne contenant une apostrophe
'Salut
à toi'          # Salut, nouvelle ligne, à toi
```

Représentation des chaînes de caractères : guillements



- Seconde méthode : la portion de texte est encadrée par des guillemets *"..."*
- Les guillements permettent l'interprétation des caractères d'échappement suivants : **`\t`** (tabulation), **`\n`** (passage à la ligne), **`\b`** (backspace), **`\\`** (barre oblique inverse), **`\"`** (guillemets)...

```
"claude"              # équivalent de 'claude'
"Salut!\n"            # Salut!, passage à la ligne
"pomme\tpoire"        # pomme, tabulation, poire
"\"quoi?\" dit-il" # "quoi?" dit-il
```

Les opérateurs pour les textes

- Les opérateurs de concaténation sont les suivants :

```
.  (concaténation simple)

x  (concaténation multiple)
```

Concaténation de textes : exemples

- Concaténation simple:

```
"Salut" . "à toi!"          # équivaut à "Salutà toi!"
"Salut" . ' ' . "à toi!"    # équivaut à 'Salut à toi!'
'Salut à toi!' . "\n"       # équivaut à "Salut à toi!\n"
```

- Concaténation multiple:

```
"bonjour" x 3  # équivaut à "bonjourbonjourbonjour"
"bonjour " x 2 # équivaut à "bonjour bonjour "
5 x 4          # équivaut à "5" x 4 soit "5555"
```

Expression mixtes

- La nature d'une expression (valeur numérique, texte) peut dépendre du contexte.
  - Si les opérateurs sont numériques, on parle de contexte numérique
  - Si les opérateurs sont des opérateurs de chaînes, on parle de contexte de chaîne
- Exemples d'expressions mixtes

```
"2" * "12"    # vaut 24 : les textes sont interprétées
              # selon leur valeur numérique
2 . 12        # vaut "212" : les nombres sont interprétés
              # en tant que chaînes de caractères
"2 euros le kilo" * "12 kilos"
              # vaut également 24 (les blancs et
              # les espaces sont ignorés)
"vous devez " . "2 euros le kilo" * "12 kilos" . " euros"
              # équivaut à "vous devez " . 24 . " euros",
```

Les variables

- Une variable est un identificateur qui désigne une *mémoire* élémentaire destinée à stocker une valeur scalaire ou composée.
- Le contenu d'une variable est susceptible d'évoluer au cours de l'exécution du programme.
- En Perl, les variables sont constituées du caractère **$** suivi d'une série de caractères et de chiffres servant d'identificateur :
- Exemples :

```
$x, $y, $i, $j, $circonférence, $rayon,…
$nom, $prenom, $texte_1, $texte_2, …
$une_tres_longue_variable…
```

Les affectations

- Pour affecter une valeur de type numérique ou texte à une variable, on utilise l'opérateur d'affectation **=**.
- Exemples :

```
$x = 12;          # $x reçoit la valeur 12
$y = 'Salut';     # $y reçoit le texte 'Salut'
$y = $x + 3;      # $y reçoit la valeur actuelle
                  # de $x plus 3 (soit 15)
$y = $y * 2;      # $y est multiplié par 2
                  # (et vaut 30)
```

Raccourcis pour l'affectation

- Pour incrémenter ou décrémenter une variable numérique entière `$i`:

```
$i++;     # incrémentation
$i--;     # décrémentation
```

- Si `?` est un opérateur, l'expression

```
$x = $x ? $val;
```

peut être raccourcie en

```
$x ?= $val;
```

Exemples :

```
$x += 5;   # équivaut à $x = $x + 5;
$y **= 3;  # équivaut à $y = $y ** 3;
$texte .= $ajout;
           # équivaut à $texte = $texte . $ajout
```

La valeur undef

- Toute variable non déclarée est mise à la valeur **`undef`**.
- Cette valeur est équivalente à la valeur `0` en contexte numérique, et au texte vide `''` en contexte texte.
- Exemple :

```
$x = 12;
$z = $x * $y; # vaut 0 car $y vaut undef

print "Bonjour, $nom."
  # affiche Bonjour, . (car $nom vaut undef)
```

3. Les entrées/sorties

Généralités

- Les entrées/sorties permettent de rendre un programme interactif :
  - Affichage de résultats de calcul sur la sortie standard: **STDOUT**.
  - Lecture de valeurs sur l'entrée standard : **STDIN**.
- Classiquement, lors de l'exécution d'un programme, l'entrée standard est constituée par le clavier, et la sortie standard par l'écran.
- Sur le système UNIX, il est facile de rediriger l'entrée et la sortie standard : on associe l'une ou l'autre à un fichier texte :

```
monprog.pl < commandes.txt
monprog.pl > resultat.txt
```

redirection de l'entrée

redirection de la sortie

L'affichage avec print

- L'opérateur **`print`** est une fonction prédéfinie permettant l'affichage sur la sortie standard.
- Cet opérateur est indispensable pour rendre compte des opérations effectuées par le programme.
- Les différents arguments de l'affichage peuvent être séparés par des virgules. Il peut s'agir de nombres, de portions de textes, d'expressions, ou de variables dont on veut consulter la valeur.
- Exemples :

```
print "Salut tout le monde";
print "La réponse est ",6*7,".\n";
```

Interpolation de variables

- Les littéraux de chaînes entre guillemets autorisent l'*interpolation de variables*. Lorsqu'un nom de variable est placé à l'intérieur du texte, celui-ci est remplacé par sa valeur courante lors de l'affichage.
- Exemples :

```
$fruit = "peche";
print "$fruit melba ou tarte aux $fruits ?";
    # donne : peche melba ou tarte aux ?
    # ($fruits est indéfini)
print "$fruit melba ou tarte aux ${fruit}s ?";
    # donne : peche melba ou tarte aux peches ?
$x = 3+4;
print "les $x mercenaires";
    # les 7 mercenaires
```

Pour annuler l'interpolation

On utilise \

```
print "\$fruit vaut $fruit"
              # affiche : $fruit vaut peche
```

Lecture de l'entrée standard

- L'entrée standard est associée au mot-clé **STDIN**.
- Pour lire une ligne de cette entrée (une série de caractères suivis du caractère "Entrée"), on fait appel à l'opérateur de lecture de ligne <…>.
- Celui-ci est appliqué à l'entrée standard: **<STDIN>** nous donne le contenu d'une ligne tapée sur l'entrée standard.
- Exemple :

```
$ligne = <STDIN>;
if ($ligne eq "\n") {
  print "Ceci n'est qu'un ligne vide \n";
} else {
  print "La ligne saisie était $ligne";
}
```

L'opérateur chomp

- Il s'agit d'un opérateur hyper-spécialisé qui sert simplement à supprimer le caractère `'\n'` de l'entrée saisie.
- Il s'agit donc d'une petite opération de nettoyage utilisable à chaque saisie
- On écrit classiquement :

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

Ou plus simplement :

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

La fonction defined

- L'opérateur **<STDIN>** est susceptible de renvoyer **undef** lorsqu'il n'y a plus d'entrée.
- Pour savoir si une valeur est undef, et non la chaîne vide, on utilise l'opérateur defined

```
$texte = <STDIN>;
if defined ($texte) {
  print "la saisie est $texte";
} else {
  print "Aucune saisie disponible";
}
```

4. Structures de contrôle

Généralités

- Diverses structures de contrôle sont proposées en Perl, elles évaluent une expression booléenne comme **vraie** ou **fausse**, et proposent un traitement selon les schémas algorithmiques classiques :
  - Les tests :

```
if (expression booléenne) {
  bloc d'instructions
}
```

  - Les boucles :

```
while (expression booléenne) {
  bloc d'instructions
}
```

Valeurs booléennes des expressions

- L'expression booléenne fournie doit être évaluée comme vraie ou fausse.
- A toute expression numérique ou scalaire peut être associée la valeur vrai ou faux, selon les règles suivantes.
  - Un nombre différent de 0 correspond à la valeur **vrai**
  - Une chaîne non vide ou différente de **"0"** correspond à la valeur **vrai**.
  - Les opérateurs de comparaison retournent **1** pour vrai et **0** pour faux
- ci-dessous quelques exemples d'expressions vraies ou fausses

```
0               # faux
"0"             # faux, c'est 0 converti en chaîne
''              # faux, chaîne vide
'00'            # vrai, différent de 0 ou '0'
1               # vrai
"n'importe quoi" # vrai
undef           # undef rend une chaîne vide
                # donc faux
```

Opérateurs de comparaison

- Les opérateurs de comparaison de chaînes sont les suivants :

`lt` (inférieur)	`gt` (supérieur)
`le` (inférieur ou égal)	`ge` (supérieur ou égal)
`eq` (égalité)	`ne` (différence)

- Les opérateurs de comparaison de nombres sont les suivants :

`<` (inférieur)	`>` (supérieur)
`<=` (inférieur ou égal)	`>=` (supérieur ou égal)
`==` (égalité)	`!=` (différence)

Les tests

- Un bloc d'instructions est constitué par une séquence d'instructions, séparées par des `;` et encadrée par des accolades `{ … }`
- Les tests ont réalisés à l'aide de la structure de contrôle **`if`**

```
if (condition) {
  bloc 1
} else {
  bloc 2
}
```

- Si la condition est vraie, le bloc 1 est exécuté, sinon c'est le bloc 2.
- Exemple :

```
if ($nom gt 'michel') {
  print "'$nom' vient après michel dans le tri
  alphabétique";
}
```

Les boucles

- La boucle **while** répète un bloc de code tant que la condition est vraie

```
while (condition) {
  bloc
}
```

- Attention! Il importe de vérifier que la condition devient fausse au bout d'un certain temps, sinon : boucle infinie!
- Exemple :

```
$n=1;
while ($n < 10) {
  print "je sais compter jusqu'à $n !\n";
  $n++;
}
```

Boucle de saisie de l'entrée

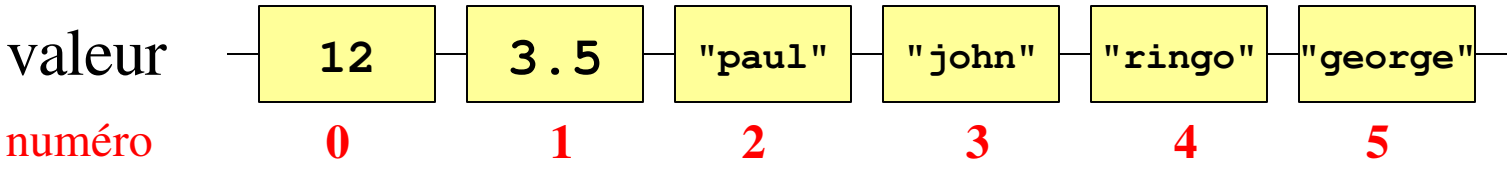
- Lorsque la dernière ligne est atteinte, <STDIN> retourne undef et la boucle while s’arrête

```
while ($ligne = <STDIN>)
  { ... }
```

5. Listes et tableaux

Généralités

- La manipulation des variables composées repose en Perl sur l'utilisation des listes.
- Une liste est constituée par une séquence de valeurs scalaires, indifféremment de type chaîne ou numérique.
- Les éléments d'une liste sont numérotés à partir de 0.
- Il n'y a pas de limite fixée à la taille d'une liste: il est toujours possible d'ajouter ou de supprimer des éléments.



Littéraux de liste

- Un littéral de liste est déclaré par une liste de valeurs entre parenthèses séparées par des virgules
- Exemples :

```
("paul","john", "ringo", "george")
(08,05,1945)
("pomme", 3, "poire", 5)
```

- L'opérateur d'étendue : **. .**

```
(1..5)      # vaut (1,2,3,4,5)
(0, 2..6, 10,12)
            # vaut (0,2,3,4,5,6,10,12)
($a..$b)    # étendue délimitée par les valeurs de $a
            # et $b
('a'..'z') # toutes les minuscules de a à z
```

Le raccourci qw

- Opérateur "quoted words"
- Permet de créer une liste de mots en évitant de saisir tous les guillemets

```
qw/ john paul george ringo /
    # équivalent à ("john","paul","george,"ringo")
qw( éteins le gaz
avant de sortir ! )
    # ("éteins","le","gaz","avant","de","sortir","!")
qw{ /usr/bin
      /usr/local/bin
     /usr/X11/bin }
```

- Remarque : de nombreux délimiteurs sont acceptés `/.../`, `#...#`, `!...!`, `(...)`, `[...]`, `<...>`, ...

Affectation sur les listes

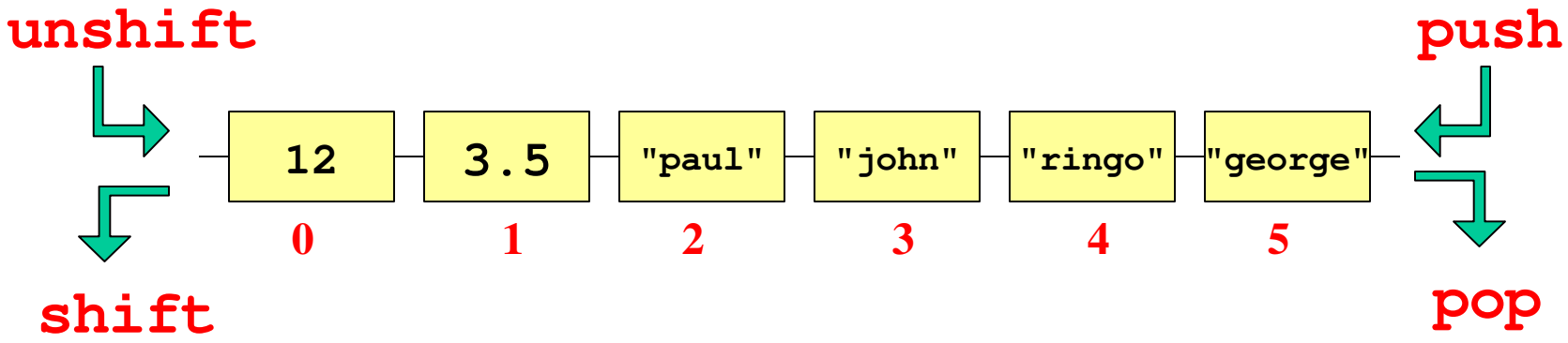
- Affectation en série sur variables scalaires :

```
($x,$y,$z) = (12,45,18);
($x,$y) = ($y,$x);    # échange de valeurs
```

- Identificateur de liste : précédé de **@**

```
@beatles = qw/ john paul george ringo /;
@liste_vide = ();
@grand = (1..1e5);    # 100 000 éléments
@plus_grand=(@grand,@grand);
```

Empilage, dépilage,...



- Opérateurs de manipulation de liste:
  - pop (dépilage)
  - push (empilage)
  - shift (décalage)
  - unshift (insertion)
- Exemples :

```
@table = 5..8;      # @table contient (5,6,7,8)
$x=pop @table;      # @table contient (5,6,7), $x vaut 8
pop @table          # @table contient (5,6)
push @table,0;      # @table contient (5,6,0)
push @table,1..5;   # @table contient (5,6,0,1,2,3,4,5)
$y=shift @table     # @table contient (6,0,1,2,3,4,5), $y vaut 6
```

Interpolation de listes en chaînes

- Comme les scalaires, les éléments d'une liste peuvent être interpolés au sein d'une chaîne entre guillemets.
- Les éléments sont automatiquement séparés par des espaces lors de l'interpolation.

```
@beatles=qw/ john paul george ringo /;
print "Les Beatles sont : @beatles";
```

Accès individuel aux éléments

- Les éléments d'une liste peuvent être accédés individuellement par leur indice. Il y a équivalence en Perl entre liste et tableaux
- Attention:
  - le premier indice est 0 (comme en C)
  - pour accéder à un élément, l'identificateur de liste est précédé de $
- Accès aux éléments d'une liste :

```
$beatles[0]              # "john"
$beatles[3]              # "ringo"
```

- Taille d'un tableau :

```
$#beatles        # vaut 3
                 # (indice du dernier élément)
```

Une nouvelle structure de contrôle : foreach

- La structure de contrôle **`foreach`** permet de produire une boucle qui visite chaque élément d'une liste.
- Exemples

```
foreach $musicien (@beatles) {
  print "$musicien est membre des Beatles!\n";
}
foreach $i (1..10) {
  print "Je sais compter jusqu'à $i !\n";
}
```

La variable par défaut : $_

- Il n'est pas nécesaire de préciser le nom de la variable de boucle qui parcourt la liste. Par défaut, celle-ci s'appelle **$_**.
- Remarque : **les variables par défaut sont beaucoup utilisées en Perl**.
  - Elles évitent la recherche de noms de variables.
  - Elles raccourcissent l'écriture.
- Exemples :

```
foreach (@beatles) {
  print "$_ est membre des Beatles!\n";
}
foreach (1..10) {
  print "Je sais compter jusqu'à $_ !\n";
}
```

Trier une liste

- Le tri d'une liste est obtenu avec l'opérateur **`sort`**.

```
@beatles_tri=sort @beatles;
    # donne (george,john,paul,ringo)
```

- Remarque : les éléments de la liste sont triés selon l'ordre des caractères ASCII :
  - Les chiffres précèdent les majuscules,
  - Les majuscules précèdent les minuscules.

Contexte de liste

- Attention! L'action de certains opérateurs change selon que le format de réception est un scalaire ou une liste.
- Exemple : l'opérateur **`reverse`**

```
@inverse = reverse qw/ john paul george ringo /;
           # donne ringo, george, paul, john
$inverse = reverse qw/ john paul george ringo /;
           # donne nhojluapegroegognir
```

Utilisation de <STDIN> dans un contexte de liste

- Dans un contexte scalaire, <STDIN> renvoie la ligne courante.
- Dans un contexte de liste, <STDIN> renvoie la liste de toutes les lignes jusqu'à la fin du fichier d'entrée (ou si l'utilisateur tape ctrl-D sur l'entrée clavier).
- Exemple :

```
@lignes = <STDIN>;
          # lit toutes les lignes de l'entrée
chomp(@lignes);
          # supprime le caractère "Entrée" à la
          # fin de chaque ligne
```

6. Sous-programmes et fonctions

La portée des variables

- Par défaut une variable Perl est globale, elle est donc *visible* dans l'ensemble du programme.
- De plus Perl n'oblige pas à déclarer les variables avant de les utiliser. Ces deux propriétés sont sources de bien des problèmes de mise au point,
- Perl permet donc de déclarer des **variables locales** et d'utiliser un mécanisme obligeant le programmeur à déclarer les variables avant de les utiliser.

Variable locale

- Il est possible de rendre locale une variable en précédant sa déclaration par `my`.
- elle ne sera alors connue que du bloc ou de la fonction (voir plus loin) qui contient sa déclaration.
- Les variables locales sont par défaut initialisées à `undef`.

```
#!/usr/local/bin/perl
$i = 10;
    {
    my $i = 2;
        {
        $i++;
            {
            my $i = 4;
            print "i = $i \n"; # Affiche i = 4
            }
        print "i = $i \n";    # Affiche i = 3
        }
    print "i = $i \n";        # Affiche i = 3
    }
print "i = $i \n";           # Affiche i = 10
```

Sous-programme et fonction

- Un sous-programme est un morceau de programme référencé par un nom,
  - A l'appel de ce nom, l'exécution du sous-programme est activée (rupture de séquence).
  - Un sous-programme peut recevoir un ou plusieurs arguments.
- Une fonction est un sous programme qui fournit une valeur de retour (son résultat).
- En Perl, on utilise le mot-clé **`sub`** pour définir sous-programmes et fonctions.

```
sub ma_fonction {
  bloc;
  }
```

passage des paramètres

- La variable par défaut `@_` contient la **liste** des arguments d'entrée de la fonction.
- Il est fortement conseillé d'utiliser des **variables locales** à l'intérieur d'une fonction!
- Ecriture de la fonction :

```
sub max {
  my $res;              # $res variable locale
   $res = shift @_;     # premier élément
  foreach (@_) {
       if($_ > $res){
              $res = $_;
              }
 return $res;           # la valeur retournée
}                       # (le mot-clé return
                        # est facultatif)
```

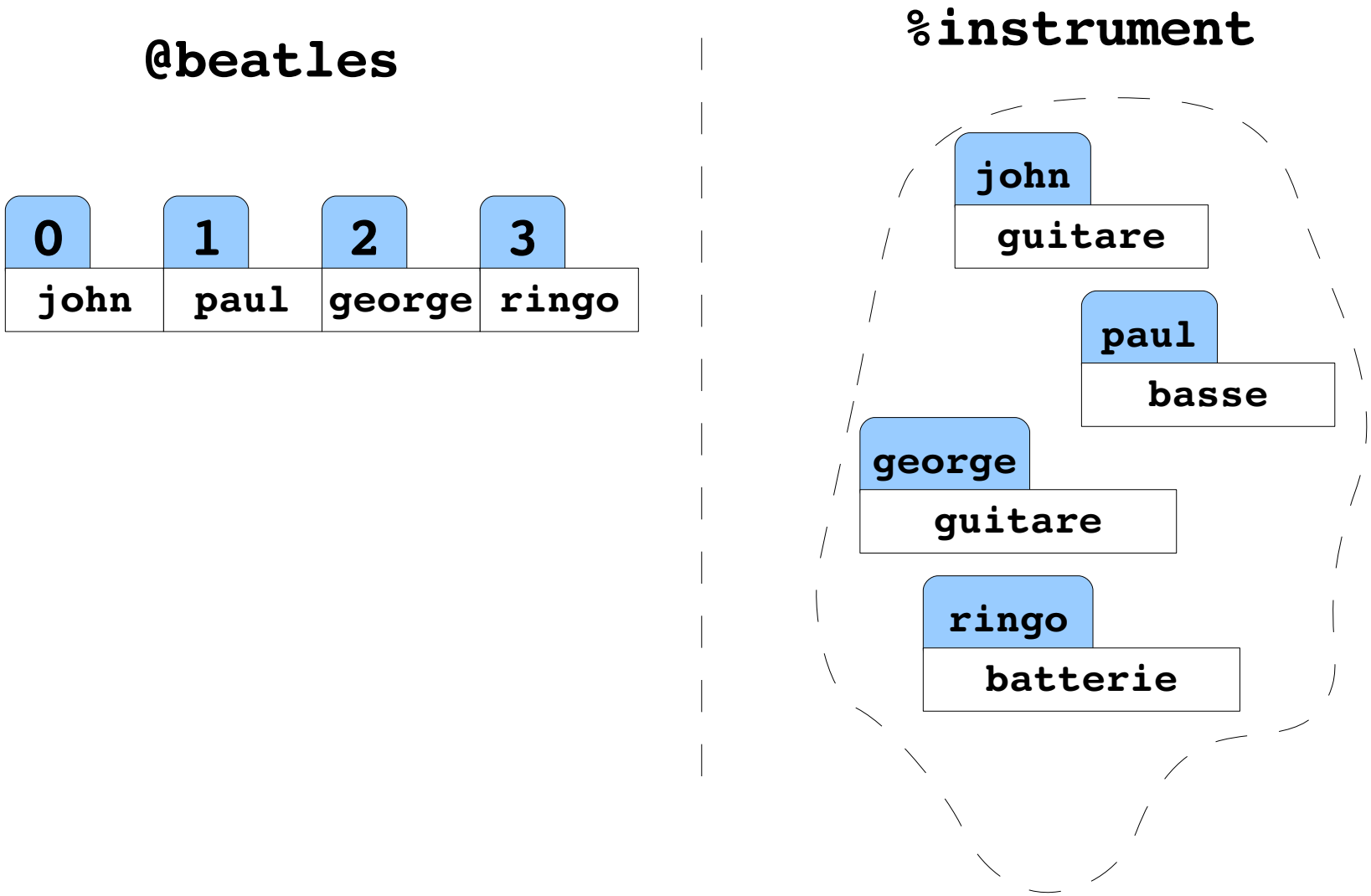
Appel de fonction

- On a défini une fonction **`max`** qui retourne l'élément maximal d'une liste fournie en argument.
- Le descripteur d'une fonction personnelle (non prédéfinie) doit être précédé du signe **`&`** lors de l'appel.
- Voici comment la fonction max peut être appelée :

```
@beatles=qw/ john paul george ringo/;
$premier = &max (@beatles);
$grand = &max (4,6,3,8,2,1)
```

7. Les tableaux associatifs

Tableaux et Hachages



Définition

- Différence tableaux/hachages
  - tableaux = ensembles ordonnés,
  - hachages = ensembles non ordonnés.
- Un hachage est un ensemble :
  - de couples (clé, valeur)
  - dont le premier élément (clé) détermine le second (valeur scalaire).
- Cette définition impose :
  - l'unicité des valeurs de clés.

Désignation

- Les identificateurs de tableaux associatifs (hachages) sont précédés du symbole `%`
- Soit :
  - le hachage `%instrument`
  - et l'une de ses clés `$cle= "ringo"`,
  - alors sa valeur correspondante s'obtient par `$instrument{$cle}`.
  - Si la clé n'existe pas, on obtient la valeur `undef`

Déclaration d'un Hachage

```
%instrument = (
    "john" => "guitare",
    "paul" => "basse",
    "george" => "guitare",
    "ringo" => "batterie"
    );
```

Accès à un élément

- Comme dans le cas des tableaux,
  - un élément isolé est un scalaire :
  - le signe `$` précède la désignation de l'élément
- Attention : accès par accolades `{...}` (et non crochets `[...]`).
- Exemple :

```
print $instrument{$ringo};
```

Exemple 1

```
#!/usr/bin/perl -w
%valeur = ("pi" , 3.14 ,
        "c", "300000 km.s-1",
        "e" , 2.72,
        "q", "1.6e-19 C" );
print "La charge électrique d'un électron est environ
$valeur{q}\n";
$expo= "e";
print "La constante $expo vaut : $valeur{$expo}\n";
```

Exemple 2

```
#!/usr/bin/perl -w

%capitale = (
        'France' => 'Paris',
        'Italie' => 'Rome',
        'Belgique' => 'Bruxelles'
        );
print "la capitale de l'Italie est $capitales{Italie}\n";

$capitale{'Allemagne'}='Bonn';

  # mise à jour de table car cette capitale a changé ...
if (exists $capitale{Allemagne} ) {
        delete $capitale{'Allemagne'};
        $capitale{'Allemagne'}='Berlin';
        }

  # création du meme hachage par mise en correspondance de 2
listes
@pays= qw(France Italie Belgique Allemagne);
@cap = qw(Paris Rome Bruxelles Berlin);
@capitale{@pays} = @cap;
```

Test d'existence

- Pour tester l'existence d'une clé ou d'une valeur utiliser respectivement les fonctions booléennes **`exits`** et **`defined`** :

```
if (exists $hach{$cle} ) {
  ...
if (defined $hach{$cle} ) {
...
```

Récapitulatif

- Variables scalaires :
  - désignation par **`$`**
- Tableaux :
  - déclaration par **`(...)`** ou **`qw`**
  - désignation de l'ensemble par **`@`**
  - accès à un élément par **`[...]`**
- Hachages :
  - déclaration par **`(...)`** ou **`qw`**
  - désignation de l'ensemble par **`%`**
  - accès à un élément par **`{...}`**

8. Manipulation de fichiers sous Unix

Les systèmes de fichiers

- Tous les systèmes d'exploitation proposent un système de fichiers
- Un fichier correspond traditionnellement à une zone de stockage sur un périphérique de stockage de l'ordinateur.
  - Désigné par un identificateur, une taille, des droits d'accès et d'exécution…
  - Localisation unique au sein d'un d'un système hiérarchique de répertoires.

Système de fichiers Unix

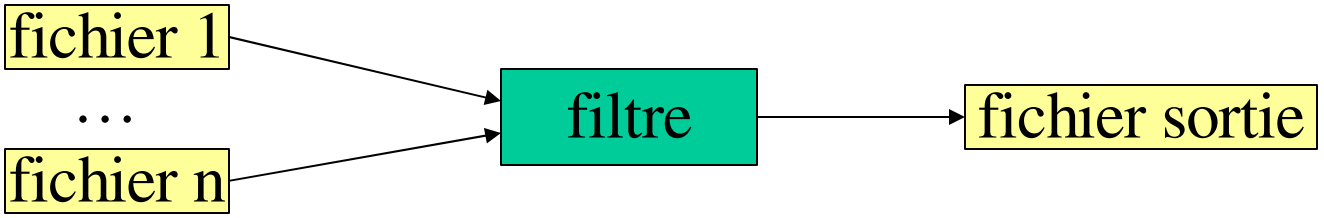
- En Unix, ***tous*** les périphériques (imprimante, écran,clavier) sont considérés comme des fichiers.
- Un fichier peut être accessible en :
  - Lecture : périphérique d'entrée
  - Ecriture : périphérique de sortie
  - Exécution : commande
- De nombreuses commandes (**`cat`**, **`grep`**, **`tail`**, **`split`**, **`cut`**, …) permettent de manipuler les fichiers. Les fichiers sont considérés comme des *arguments*.
- Exemples :
  - Ecran, Imprimante : fichiers accessibles en écriture seule
  - Clavier, souris, … : fichiers accessibles en lecture seule
  - Commande, répertoire : fichier accessible en exécution…

Commandes Unix sur les fichiers

- Commandes opérant sur les répertoires le positionnement des fichiers :
  - ls, pwd, find,… lecture de l'arborescence
  - cp,mv,rm…modification de l'arborescence
  - chown,chmod,…changement des droits d'accès
- Commandes de manipulation de fichiers (filtres)
  - cat, grep, head, tail, cut… lecture et extraction d'informations…
  - cmp,diff : comparaison de fichiers
  - join,split : fusion ou découpage de fichiers

Utilisation des filtres

- Un filtre est une commande d'extraction et de manipulation d'information prenant des fichiers (texte) comme arguments



- Une commande s'appelle en général par :

```
$ filtre fichier_1 fichier_2 … fichier_n
```

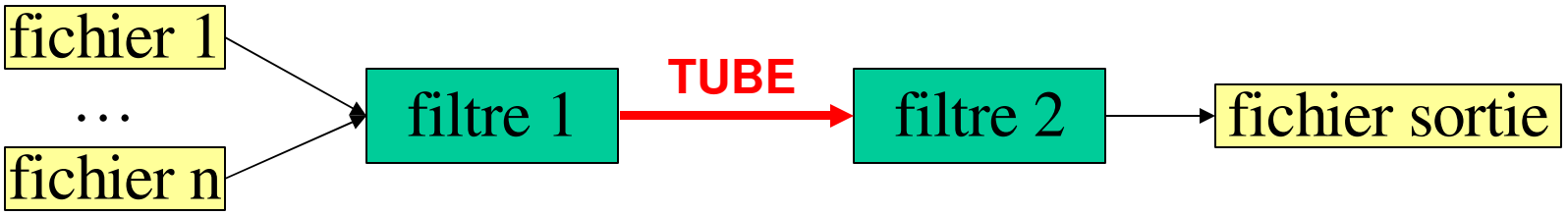
- Par défaut le résultat du traitement est affiché sur la sortie standard (l'écran).
- Pour récupérer ou sauvegarder le résultat du traitement, il faut en général ,il faut réaliser une **redirection**.

Redirections

`commande > fichier_sortie`	**Ecriture** Redirection de la sortie standard en écrasant le contenu de fichier_sortie
`commande >> fichier_sortie`	**Ajout** Redirection de la sortie standard ajoutée en fin de fichier_sortie
`commande >& fichier_sortie`	**Erreur** Redirection de la sortie standard en écrasant le contenu de fichier_sortie
`commande < fichier_entree`	**Lecture** Redirection de l'entrée standard
`commande_1 | commande_2`	**Tube** La sortie standard de commande_1 est redirigée vers l'entrée standard de commande_2

Les tubes

- L'utilisation des tubes permet d'enchaîner les filtres pour réaliser un traitement en une seule ligne de commande



- Exemple : soit un fichier donnees.txt organisees en tableau (300 lignes) :

```
$ head -100 donnees.txt | tail - 100 | cut -c31-39 > extrait.txt
```

Organisation d'un fichier texte

- Une séquence de chaînes de caractères séparées par des passages à la ligne (caractère \n)

	col 1	col 2	col 3	col 4	…							
ligne 1	b	l	a	b	l	a	\n					
ligne 2	\n											
ligne 3	b	l	a	b	l	a	b	l	a	b	l	a	\n
…	b	l	a	\n								
	b	l	a	b	l	a	b	l	a	\n		
	b	l	a	\n								

La commande cat

- permet la concaténation des fichiers fournis en argument

```
$ cat chapitre1.txt chapitre2.txt chapitre3.txt > tome1.txt

$ cat chapitre*.txt > tome1.txt
```

La commande head

- Extrait les premières lignes d'un fichier:

head [-*nombre*] *nom_fichier[s]*

- Exemple :

```
$ head -100 exemple.txt > extrait.txt
```

(Extrait les 100 premières lignes)

La commande tail

- Extrait les dernières lignes d'un fichier

tail [-*nombre*] *nom_fichier[s]*

- Exemple :

```
$ tail -100 exemple.txt > extrait.txt
```

(Extrait les 100 dernières lignes)

La commande split

- Permet de découper un fichier

split -l *nb_lignes nom_fichier*

- Exemple :

```
$ split -l 100 exemple.txt
```

(produit par défaut **xaa**, **xab**,...)

La commande csplit

- Idem, mais plus évolué

csplit –f *préfixe* –n *longueur nom_fichier /motif/*

- Exemple

```
$ csplit –f extrait –n 2 exemple.txt /\n/
```

La commande cut

- Extrait des colonnes d'un fichier

`cut -c` *`col_ini-col_fin nom_fichier`*

Organisation d'un fichier de données texte

- Dans un fichier de données texte, les donnees sont structurées sous forme de tableau, où les champs sont en général séparés par des tabulations

	champ 1		champ 2		champ 3	
ligne 1	info11	\t	info12	\t	info13	\n
ligne 2	info21	\t	…	\t	…	\n
ligne 3	info31	\t	…	\t	…	\n
	…	\t	…	\t	…	\n
	…	\t	…	\t	…	\n
	…	\t	…	\t	…	\n

La commande cut

- Extrait des champs d'un fichier (délimités par des tabulations)

`cut -f` *`champ_ini-champ_fin nom_fichier`*

La commande join

- Fusionne les contenus de plusieurs fichiers de donnes texte sur la base d'un champ de jointure

```
join -1 numero_champ nom_fichier_1 nom_fichier_1
```

- numero_champ indique le champ servant pour la jointure

La commande sort

- Trie les lignes d'un fichier
  - Ordre lexicographique

```
sort nom_fichier(s)
```

  - Ordre numérique

```
sort -n nom_fichier(s)
```

9. Traitement des fichiers en Perl

Perl : un langage orienté vers le traitement des fichiers

- Perl a été conçu en premier lieu pour réaliser des commandes et des filtres opérant sur des fichiers.
- Il existe de nombreux raccourcis dans ce langage pour faciliter ce traitement.
- Un programme Perl peut facilement remplacer les commandes Unix prédéfinies.

```
$ filtre.pl < donnees > resultat
$ filtre1.pl <donnees | filtre2.pl >  resultat
$ filtre.pl donnees_1 donnees_2 donnees_3
```

Descripteurs de fichiers

- Un descripteur de fichiers sert à désigner une *connexion* avec un fichier (ou un périphérique d'E/S) au sein d'un programme Perl.
  - Les descripteurs de fichiers ne comportent pas de préfixe.
  - Il existe plusieurs descripteurs prédéfinis (mots réservés):
    - STDIN : flux d'entrée standard
    - STDOUT : flux de sortie standard
    - STDERR : flux de sortie des messages d'erreur
    - ARGV : fichier(s) fourni(s) en argument de la commande

Parcours des fichiers

- Un fichier est parcouru grâce à l'opérateur **<…>**
  - Chaque appel à cet opérateur permet de *poursuivre la lecture*, c'est à dire de lire l'élément qui suit l'élément précédemment lu.
  - À chaque nouvel appel à **<…>**, la tête de lecture est déplacée sur l'élément suivant.
  - Exemple :

```
while ($ligne=<F>) {
  print "voici la ligne : $ligne";
}
```

- Selon le format de réception, l'appel à **<…>** fournit
  - une chaîne de caractères (une ligne) : **`$ligne = <F>;`**
  - un tableau de chaînes de caractères (contenant la totalité du fichier): **`@total=<F>;`**

Utilisation de l'entrée et de la sortie standard

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

Avec variable implicite :

```
while (<STDIN>) {
     print "Voici $_";
}
```

Opérateur Diamant

- L'opérateur <> sans argument correspond à la lecture sur les arguments de la commande :

```
$ filtre.pl donnees_1 donnees_2 donnees_3
```

- Le contenu des arguments d'entrée est lu comme un seul fichier (les fichiers d'entrée sont mis bout à bout).

```
while (<>) {
  chomp;
  print "ligne lue : $_";
}
```

Quelques commandes Unix vite programmées

- Commande cat :

```
#!/usr/bin/perl
print <>;
```

- Commande sort :

```
#!/usr/bin/perl
print sort <>;
```

Ouverture d'un fichier

- En lecture :

```
open (F_IN, "entree.txt");
```

- En écriture

```
open (F_OUT, ">resultat.txt");
```

- En écriture/ajout

```
open (F_OUT, ">>suite_de_resultats.txt");
```

- Ecriture en sortie :

```
print F_OUT "bonjour tout le monde!"
```

- Fermeture d'un fichier :

```
close F;
```

Traitement des erreurs

```
$nom_fichier="donnees.txt";

unless (-e $nom_fichier) {
   print "Le fichier $nom_fichier n'existe pas !
   \n";
   exit;
}

unless ( open F_IN, $nom_fic ) {
   print "impossible d'ouvrir le fichier
   $nom_fichier !\n";
}
```

Liste des fichiers

```
foreach (<*>){
    print $_, "\n";
    }
```

10. Les expressions régulières

Expressions régulières

- Les expressions régulières (ou expressions rationnelles) sont
  - une famille de notations compactes et puissantes
  - pour décrire certains ensembles de chaînes de caractères.
- Ces notations sont utilisées :
  - pour parcourir de façon automatique les textes
  - À la recherche de morceaux de textes ayant certaines formes
  - Et éventuellement remplacer ces morceaux de textes par d'autres

Origine mathématique

- Théorie des automates et des langages formels
- Un langage formel est décrit comme un ensemble (dit « régulier ») de chaînes de caractères.

Description

- Soient les notations :

  Σ (ensemble fini de lettres). L'ensemble des mots que l'on peut former est noté Σ^*.

  - ε : désigne un mot vide
  - R,S,… des sous-ensembles de Σ^* (des ensembles de mots)
- Les opérateurs autorisés :
  - Concaténation : RS
    - {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}
  - Union : R U S
  - Fermeture : R* :l'ensemble de toutes les chaînes de caractères qui peuvent êtrre formées en concaténant zéro, une ou plusieurs chaaîes de *R*
    - {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "abababab", ... }

Mise en oeuvre

- a U b* désigne {"a", ε, "b", "bb", "bbb", ...};
- (a U b)* désigne l'ensemble des chaînes qui ne contiennent que des 'a' et des 'b', y compris la chaîne vide ε;
- (a* b*)* désigne exactement le même ensemble (que des 'a' et des 'b', le mot vide compris);
- (ab*)* décrit tous les mots ne contenant que des 'a' et des 'b', commençant par 'a', y compris la chaîne vide ε;
- ab*(c U ε) désigne l'ensemble des chaînes qui commencent par 'a', suivi de zéro ou plus 'b', et se terminent éventuellement par un 'c' optionnel;

Propriété

- Les expressions rationnelles :
  - sont capables de décrire exactement les mêmes langages que ceux exprimés par les **automates finis** (déterministes ou non)
  - ces langages sont appelés **langages réguliers**.

Réalisation algorithmique

- Une expression rationnelle permet de décrire un ensemble de motifs
- Les expressions rationnelles sont implémentées sous la forme d'algorithmes de recherche de motifs (ou de mots)
- Problème de la recherche de motifs :
  - Trouver toutes les occurrences d'un motif M dans un texte T.

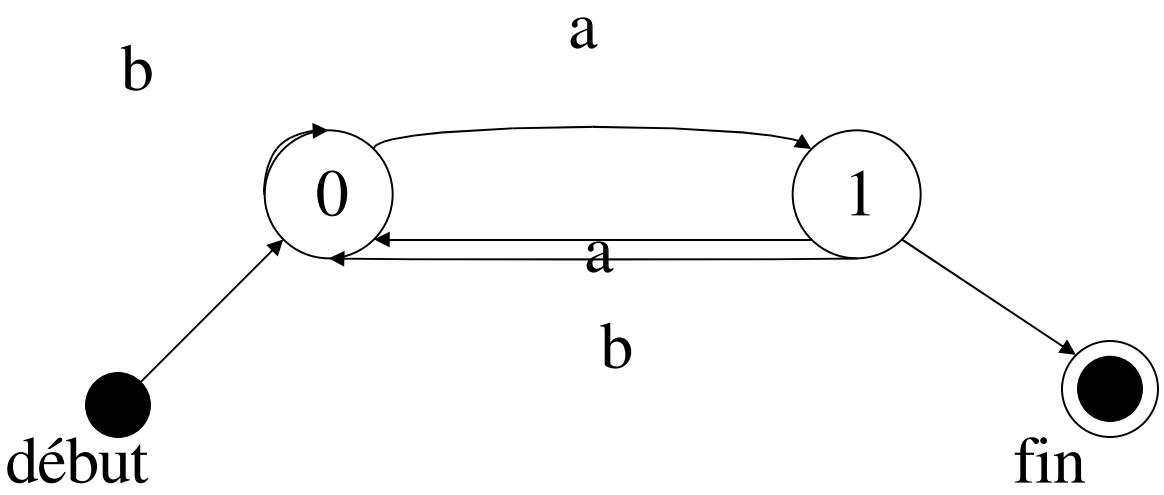
Recherche au moyen d'automates finis

- Exemple :

  Σ={a,b}

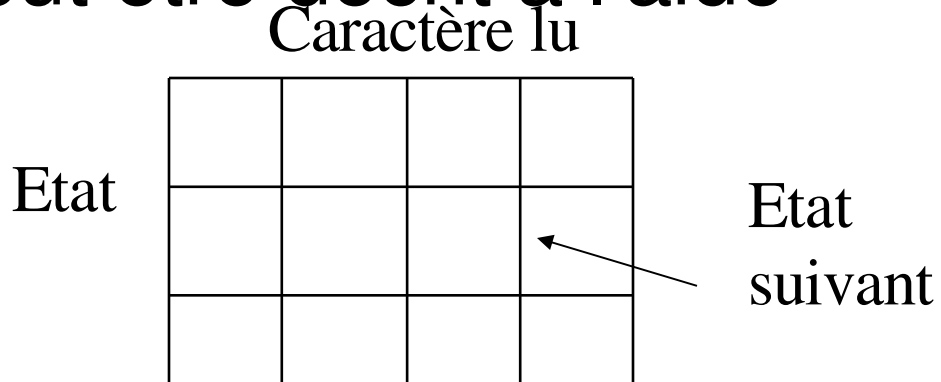
  – chaîne se terminant par un nombre impair de 'a'.

  – Soit b* (aab* U abb*)* a



Définition d'un automate fini

- Définition :
  - Un ensemble fini d'états S={0,1,…,n}
  - Un état initial (0 par défaut)
  - Un ensemble S' d'états terminaux, S' $\subset$ S.
  - Un alphabet Σ
  - Une fonction de transition A : S x $\Sigma \rightarrow$ S
- Un automate peut être décrit à l'aide d'une table :



algorithme

- Soit T le texte considéré et A un automate

```
N = longueur (T)
s = 0
pour i de 1 à n répéter
   s = A (s,T[i])
   si s ∈ S'
            afficher «  motif trouvé à l'emplacement  », i
   fin si
fin pour
```

Remarques

- Cas d'échec :
  - Transition impossible

  Dans ce cas : on redémarre
  - dans l'état initial ($s=s_0$)
  - À partir du deuxième caractère
  - Etc…
- Algorithme glouton :
  - S'arrête sur le premier motif trouvé
  - Accepte le motif de taille maximale
  - S'arrête sur l'occurrence située la plus à gauche

Sous Unix

- Une expression régulière est une **méthode de description des motifs** qui peuvent apparaître dans une chaîne de caractères.
- Si le motif est présent dans la chaîne de caractères, on dit qu'il y a **correspondance** (*matching)* entre la chaîne de caractères et le motif.
- Les expressions régulières sont présentes dans plusieurs langages ou commandes tels que *awk,sed*,*grep*

Sous Unix

- Notations de base pour les expressions rationnelles sous Unix :
  - '.' : un joker, qui représente un caractère unique quelconque (sauf caractères de contrôle et fin de ligne).
  - '[ ]': des classes de caractères entre crochets [ ].
  - '^': au début d'une classe de caractères entre crochets, signifie qu'on considère le complément de cette classe (l'ensemble des caractères qui ne sont pas dans la classe).
  - '^' et '$': représentent respectivement un début et une fin de ligne.

Exemples

- ".ac" représente les mots de trois lettres qui se terminent par "ac"
- "[a-z]" correspond à n'importe quelle lettre minuscule (non-accentuée)
- "[^a-z]" correspond à n'importe quel caractère qui n'est pas une lettre minuscule non-accentuée
- "[st]ac" représente entre autres "sac" et "tac"
- "[^f]ac" représente les mots de trois lettres qui se terminent par "ac" et ne commencent pas par "f"
- "^[st]ac" représente les mots "sac" et "tac" en début de ligne
- "[st]ac$" représente les mots "sac" et "tac" en fin de ligne
- "^trac$" représente le mot "trac" seul sur une ligne

grep

- L'utilitaire grep du monde Unix étend cette liste avec :
- ’+’ : spécificateur de quantité�: ce qui précède, répété� une fois ou plus.
- ’*’ : spécificateur de quantité�: "zéro, une fois ou plus ce qui précède. »
- ’?’ : spécificateur de quantit�é: "zéro ou une fois ce qui précède".
- ’|’ : l'opérateur de choix, c'est-à�-dire l'union ensembliste.

Exemples

- "chat|chien" : représente les mots "chat" et "chien" (et seulement eux).
- "([cC]hat|[cC]hien)" : représente "chat", "Chat", "chien" et "Chien".
- "ch+t": représente "cht", "chht", "chhht", etc.
- "a[ou]+" : représente "aou", "ao", "auuu", "aououuuoou" etc.
- "peu[xt]?" : représente "peu", "peux" et "peut".

Expressions rationnelles en Perl

- Perl, offre un ensemble d'extensions aux expressions rationnelles particulièrement riche.
- Le traitement des expressions rationnelles est une spécialitté de Perl :
  - Les opérateurs d'expressions rationnelles sont inclus dans le langage lui-même.

Exemples de motifs

- `/jour/` : motif simple présent dans la chaîne `"bonjour tout le monde"`
- `/par.i/` : motif présent deux fois dans `"est-il parti ou parmi nous?"`(le `.` remplace tout caractère)
- `/(la)*l/` : motif présent une fois dans `"lalalalalère!"et aussi dans "la"` (le `*` signifie la répétition un nombre indéfini de fois, éventuellement 0 fois)
- `/par.*ait/` : motif présent deux fois dans `"on parlait qu'il partirait"`
- `/(sacri|ori)fice/` : apparaît dans `"sacrifice"` ou `"orifice"` mais pas dans `"édifice"`.

Caractères réservés et échapement

- La syntaxe des expressions régulières utilise un certain nombre de caractères réservés (ou métacaractères), par exemple `.`,`*`, `(`,`)`,`/`,...
- Pour ignorer un caractère réservé, on utilise le caractère d'échappement `\`.

  `/3\*2\? 5\./` correspond à la chaîne `"3*2? 5."`
- Pour ignorer le caractère d'échappement, on applique l'échappement au caractère d'échappement, soit \\.

  `/antislash : \\/` correspond à `"antislash : \"`

Caractères réservés

.	désigne n'importe quel caractère sauf le caractère de fin de ligne.
+	désigne le caractère qui précède répété **au moins une fois** (ainsi **/a+/** correspond aux séquences **"a"**, **"aa"**, **"aaa"**, etc.)
?	désigne le caractère qui précède répété **au plus une fois** (ainsi, **/a?/** correspond aux séquences **""** (chaîne vide) et **"a"**)
*	désigne le caractère qui précède répété **un nombre quelconque de fois**, y compris zéro (ainsi,**/a*/** fait correspond aux séquences **""**, **"a"**, **"aa"**, **"aaa"**, etc.)
^	désigne le ou les caractères qui précèdent s'ils sont situés **en début de ligne** (ainsi, **/^L/** désignera le L de **"La vie…"**, mais pas le L de **"ceci est un L"**)
$	désigne le ou les caractères qui précèdent s'ils sont situés **en fin de ligne** (ainsi, **/L$/** désignera le L de **"ceci est un L"**, mais pas le L de **"La vie…"**)
|	opérateur **OR** (ainsi, **a|b** désigne le caractère a **ou** le caractère b)
\	Caractère d'échappement.

Caractères réservés (suite)

( )	• désigne un **groupe** de caractères (ainsi, **/a(bcd)+/** fait correspondre les séquences **"abcd"**, **"abcdbcd"**, etc.) et • permet également de **mémoriser** une séquence (ainsi, **/<(.+)>/** mémorise le texte contenu entre **<** et **>** dans la variable $1)
[ ]	Désigne un **ensemble** de caractères (ainsi, **/[abc]/** correspond à un caractère au choix parmi a, b et c)
[^]	Désigne un **ensemble complémentaire** (ainsi, **/[^abc]/** correspond à un caractère quelconque **sauf** a, b ou c.)
–	**Intervalle** : par exemple, **/[a-z]/** désigne l'ensemble des caractères alphabétiques minuscules.
{n}	désigne le caractère qui précède **exactement n fois** (ainsi, **/a{2}/** correspond à la séquence **"aa"** et non **"a"** ou **"aaa"**)
{n,}	désigne le caractère qui précède **à partir de n fois** (ainsi, **/a{2,}/** correspond à la séquence **"aa", "aaa", "aaaa"** etc…)
{m,n}	désigne le caractère qui précède **répété entre m et n fois** (ainsi, **/a{2,3}/** correspond aux séquences **"aa"** et **"aaa"**)

Utilisation en Perl

- Comparaison avec **=~** :

```
$chaine = "une chaîne de caractères";
if ($chaine =~ /motif/) {
  print "correspondance trouvée!";
}
```

- Formulation équivalente avec variable implicite :

```
$_ = "une chaîne de caractères";
if (/motif/) {
  print "correspondance trouvée!";
}
```

- Non correspondance **!~** :

```
$chaine = " une chaîne de caractères";
if ($chaine !~ /motif/) {
  print "le motif n'est pas présent!";
}
```

Caractères invisibles et raccourcis

`\n`	Caractère de fin de ligne
`\t`	Tabulation
`\f`	Fin de page
`\s`	Espacement (`\n \t \f` ou `" "`)
`\d`	N'importe quel chiffre
`\w`	N'importe quelle lettre ou chiffre
`\U`	Transforme les caractères qui suivent en majuscules
`\L`	Transforme les caractères qui suivent en minuscules

Ancres

- Par défaut, un motif peut "flotter" entre le début et la fin de la chaîne. Pour imposer un positionnement, on peut utiliser une "ancre" :
  - début de ligne : ^
  - fin de ligne : $
  - Début ou fin de mot : \b,sachant qu'un mot est défini comme `/[A-Za-z0-9_]+/`
- Exemples :

  `/\bmichel\b/` correspond à `"michel"` mais pas à `"michelle"`.

  `/^\s*$/` correspond à une ligne vide.

  `/^pomme\b/` correspond à `"pomme d'api"` mais pas `"pommes vapeur"` ni `"goût pomme"`.

Mémorisation

- Chaque expression entre parenthèse est mémorisée au cours de son traitement.

  Expression `/… (…) … (…) … (…) …/`
  Variable `1` `2` `3` …

- Il est possible de faire référence à un motif mémorisé au sein d'une expression. On parle de **référence arrière**. La référence à un élément mémorisé est obtenue avec `\1`, `\2`, `\3`…
- Exemple :

  `/(.)\1/` correspond à n'importe quel caractère suivi du *même* caractère soit `"aa bc dd"` , `"elle"` , mais pas `"toto"`.

  `/<image source = (['"]).*\1>/` : les marques d'ouverture et de fermeture du fichier image doivent être identiques.

Les commandes

- Les commandes correspondent pour l'essentiel à des opérations de recherche et de remplacement.

  `m//` : recherche de motifs, raccourci en **`//`**(déjà vu)

  `s///` : recherche et remplacement.

  `tr///` : opérateur de transposition

- Les options (se placent en fin de commande)

  `/i` : recherche indifférente à la casse (majuscules/minuscules)

  `/s` : permet de prendre en compte le passage à la ligne comme un caractère quelconque.

  `/g` : recherche globale réitérée sur l'ensemble de la chaîne.

- Exemples :

  `m/\boui\b/i` : mot **`"oui"`** , **`"OUI"`** ou **`"Oui"`** …

  `s/john/paul/g` : remplace **`"john"`** par **`"paul"`** dans toute la chaîne.

  `tr/ACGT/TGCA/g` : calcule le complémentaire d'un brin d'ADN.

Utilisation de variables dans les expressions régulières

```
#!/usr/bin/perl
print "Entrez votre motif\n"
chomp (my $motif = <STDIN>);
while(<>){
  if (/$motif/){
     print "motif trouvé";
  } else {
     print "motif absent";
  }
}
```

Variables de correspondance

- Lors de l'évaluation d'une expression régulière, les expressions `(…)` mémorisées au cours du traitement sont automatiquement copiées dans les variables $1, $2, $3…
- Ces variables sont accessibles dans la suite du programme, et remises à jour à chaque nouvelle évaluation.
- Exemples :

```
$_ = "Je te salue, camarade!";

if ( /\s(\w+),/ ) {
   print "le mot qui précède la virgule est $1.\n";
}

if (/(\w+) (\w+) (\w+)/) {
   print "les premiers mots sont $1, $2, $3.\n";
}
```

Autres variables

- Il existe 3 variables spéciales générées automatiquement à chaque évaluation d'expression : **`$`\``**, **`$&`** et **`$'`**.
  - **`$`\``** : chaîne précédant l'expression trouvée.
  - **`$&`** : l'expression trouvée.
  - **`$'`** chaîne suivant l'expression trouvée.

Un programme de test de motif

```
#!/usr/bin/perl
while(<>) {
   chomp;
   if (/MOTIF/) {
       print "correspondance : |$`<$&>$'|\n";
       } else {
       print "Pas de correspondance \n";
   }
}
```

Substitutions

- L'opérateur **s///** permet de réaliser des substitutions :

```
$chaine = "frappez et entrez";
$chaine =~ s/ez/ons/;
print "$chaine\n"; # frappons et entrez
```

- Substitution globale : **s///g**

```
$_ = "frappez et entrez";
s/ez/ons/g;
print "$_\n"; # frappons et entrons
```

- Utilisation des variables de correspondance

```
$chaine = "il fait beau et chaud.";
$chaine =~ s/(\w+) et (\w+)/$2 et $1/;
print "$chaine\n"; # il fait chaud et beau.
```

- Changement de casse **\U**

```
$chaine =~ s/(\w+) et (\w+)/\U$&/;
print "$chaine\n"; # il fait CHAUD ET BEAU.
```

L'opérateur split

- split est un opérateur qui découpe une chaîne d'après un séparateur.

```
@champs = split /séparateur/ $chaîne
```

- Exemples :

```
@champs = split /:/ "abc:def:g:h";
          # donne ("abc","def","g","h")
@champs = split /\s+/ "Ceci est un \t   test.\n";
          # donne ("Ceci","est","un","test.")
```

L'opérateur join

- Inverse de split, il permet de coller les différents éléments d'une liste pour former une chaîne unique.

```
$resultat = join $colle, @morceaux
```

- Exemple

```
my $x = join ":",4,6,8,10,12;
        # $x reçoit "4:6:8:10:12"
```

Systèmes d’exploitation

Définition

- Le **système d'exploitation** (*SE*, en anglais *Operating System* ou *OS*)
  - est un ensemble de programmes responsables de la liaison
    - entre les ressources matérielles d'un ordinateur
    - et les applications de l'utilisateur (traitement de texte, jeu, navigateur...).
- Il s'exécute automatiquement au démarrage de l'ordinateur.
- Il a pour « mission » de gérer :
  - la communication entre le processeur central et les périphériques (stockage, entrées, sorties, réseau…)
  - La gestion des utilisateurs et des droits d'accès
  - La gestion de ressources :
    - L'ordonnancement des processus (attribution de mémoire et de temps processeur)
    - La communication entre les processus (envoi de signaux, mise en attente, réception…)
    - La réservation de mémoire vive et son attribution aux différents processus.

Plusieurs niveaux

- Premier niveau : le noyau
  - Gestion de la mémoire/des entrées-sorties/des processus
- Deuxième niveau : les langage de commandes
  - Offrent un ensemble de commandes permettant d'accéder aux fonctions du noyau.
  - Permettent à l'utilisateur d'interagir avec le système d'exploitation.
  - Exemples : DOS, Shell…
- Troisième niveau : interfaces graphiques
  - Environnement multifenêtré graphique
  - Windows, OSX, KDE, Gnome…

Le système de fichiers

- Un périphérique de stockage
  - est constitué  d' **emplacements** élémentaires (un peu comme des places de parking),
  - Destinés à recevoir des données.
  - Chacune de ces places est bien sûr identifiée par sa position (son "adresse") sur le périphérique en question.
- Un fichier correspond la **réservation** d'une zone de cet espace de stockage.
  - Il est donc propriétaire d'un ou plusieurs de ces emplacements élémentaires sur lesquels il dépose ses données.

Organisation des fichiers

- Au niveau de l'utilisateur, le système d'exploitation fournit la possibilit�é d'organiser les fichiers au sein de répertoires (également appelés dossiers).
- Le système de répertoires est un outil de classement organisé sous forme arborescente:
  - Il existe un répertoire racine (*root*) à partir duquel s'organise l'arborescence.
  - chaque répertoire (à l'exception du répertoire racine) possède un répertoire père.
  - un répertoire peut contenir à la fois des fichiers et d'autres répertoires (qui sont ses répertoires *fils*)
  - L'organisation sous forme de répertoires permet de positionner les différents fichiers du disque. La position d'un fichier au sein de l'arborescence est donnée par un chemin (*path*) qui part de la racine jusqu'au répertoire contenant le fichier.

En Perl

- Pour accéder à **la liste des fichiers** présents dans un répertoire donné :

```
my @liste_fichiers_textes = <*.txt>

my @liste_applis = </usr/bin/*>
```

Caractéristiques

- Un fichier est doté:
  - d'un nom,
  - d'une position (son chemin dans le système de répertoires)
  - d'un certain nombre de droits (son ou ses propriétaires)
  - d'une taille (correspondant à  un certain nombre d'emplacements réservés sur l'espace de stockage)
  - d'une date de création, de modification etc...

Statistiques d'un fichier

- En Perl :

```
@champs_stat = stat $nom_fichier;
```

- On trouve (13 champs en tout):
  - **`@champs_stat[2]`** : mode du fichier (type et permissions)
  - **`@champs_stat[7]`** : taille totale, en octets
  - **`@champs_stat[8]`** : date dernier accès
  - **`@champs_stat[9]`** : date dernière modif
  - **`@champs_stat[10]`** : date dernier déplacement

Bases de données

- Les bases de données sont des méthodes
  - de stockages de l'information sur disque dur
  - Optimisées pour accélérer les temps d'accès
- Elles sont au cœur de nombreuses applications:
  - Recherche par mot-clé
  - Index de disque dur
  - Systèmes de gestion de contenus web
- Elles peuvent manipuler un nombre considérable de données

Hachages DBM

- Il est possible en Perl d'accéder à un type spécial de tableau associatif, appelé hachage DBM.
- Pour associer une base de données à un hachage DBM, on utilise la fonction *dbmopen*

```
dbmopen(%MESSAGE,"ma_base_de_messsages",0644)
or die "Impossible de creer "ma_base_de_messages :$!";
```

- Cette commande crée deux fichiers : *ma_base_de_messsages.dir* et *ma_base_de_messsages.pag*
- Pour fermer la base, on utilise

```
dbmclose(%MESSAGE);
```

Hachages DBM suite

- Pour le reste, l'utilisation de *%MESSAGE* est exactement identique à celle des tableaux associatifs simples déjà vus! exemple :

```
$MESSAGE{"fred"}="bonjour!";

delete $MESSAGE{"jack"};
```

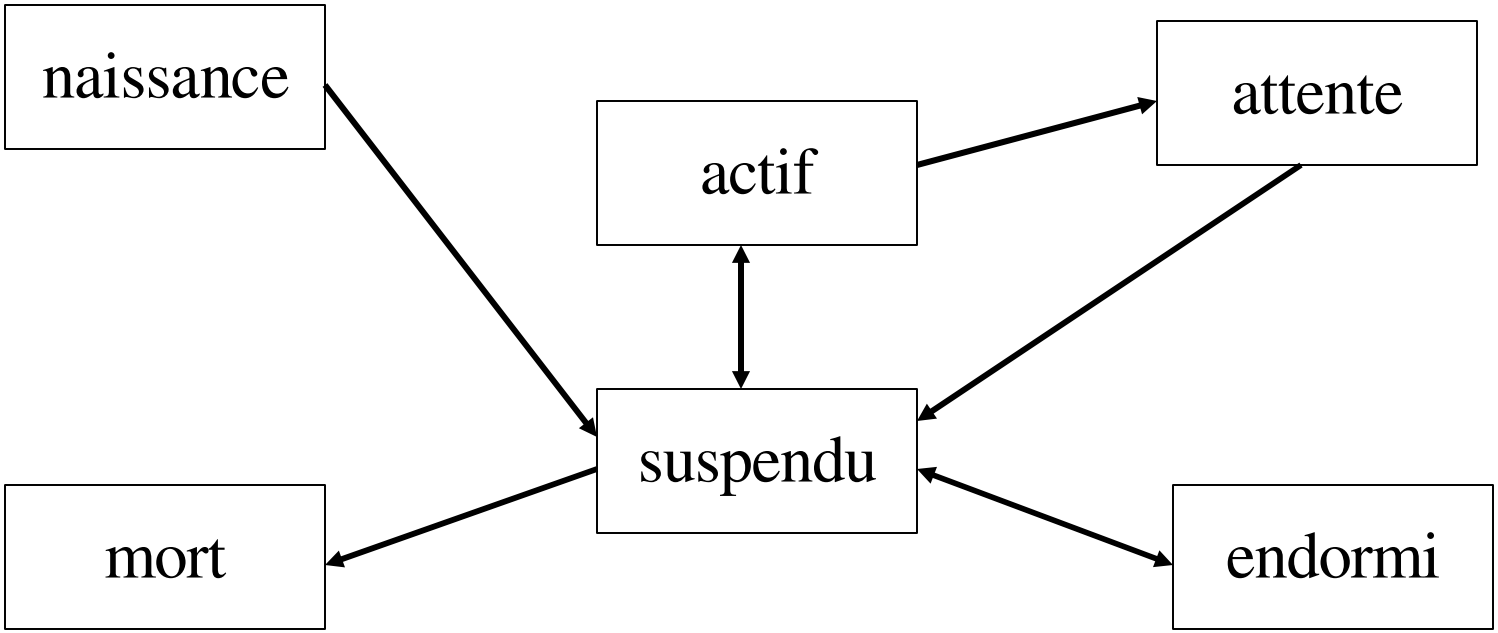
Les processus sous Unix

Les processus

- un processus est un programme en cours d'exécution, avec
  - son propre environnement d'exécution fourni par le système
  - un quota de temps processeur (géré par le système).
- Chaque processus est caractérisé par :
  - Son identificateur (PID)
  - Son processus père (PPID)
  - Son propriétaire
  - Sa priorité d'exécution
  - Son terminal associé (fenêtre d'exécution)
- Le système se charge d'attribuer aux processus de la mémoire et du temps d'accès au processeur en fonction de leur priorité

Les différents états d'un processus

- Un processus peut atteindre différents états au cours de son "existence":
- Le passage d'un état à un autre est géré par des signaux.



Communication inter-processus

- Les processus communiquent :
  - Par signaux
  - Par les fichiers
  - Par les tubes
  - Par les sockets
- Importance des interruptions
  - Ordonnancement
  - Suspension d'un processus (signal, attente d'entrée-sortie…)
  - Basculement de contexte

Mécanisme d'ordonnancement

- Pour effectuer ces tâches, l'ordonnanceur procède de la manière suivante :
  - A intervalle régulier, le système appelle une procédure d'ordonnancement qui
    - *élit* le processus à exécuter.
  - Si le nouveau processus est différent de l'ancien alors survient un **changement de contexte**, opération qui consiste à:
    - sauvegarder le contexte d'exécution de l'ancienne tâche, comme par exemple, les zones mémoires du processus, le pointeur de programme.
    - Le système d'exploitation restaure le contexte de la nouvelle tâche.

La fenêtre de commandes (shell)

- Une fenêtre de commande (ou terminal)
  - Est une interface de communication entre l'utilisateur et la machine
  - Offre un certain nombre de commandes prédéfinies
  - Permet de lancer :
    - Des opérations de manipulation de fichiers
    - Des envois de signaux vers les processus
    - Des applications

Processus père/processus fils

- Certains processus ont la possibilité de lancer d'autres processus.
  - Le processus initial (processus système) d'identité 1.
  - Les fenêtres de commande
  - Etc…
- Chaque processus (hormis le processus système) est fils du processus qui l'a lancé.

Shell suite

- Le système de processus est organisé de manière arborescente depuis le processus initial (numéroté 1)
- Chaque processus possède un numéro (PID)
- Lorsqu'une commande est lancée depuis le shell :
  - la fenêtre de commande se bloque (suspendu) jusqu'à la fin de l'exécution de la commande
  - Le processus lancé est « fils » du shell qui l'a lancé

La commande ps

- La commande **`ps`** permet de connaître les processus en cours d'exécution sur le système
- Exemple : **`ps -ef`**

```
  UID      PID    PPID TTY      STIME COMMAND
SYSTEM    1328       1   ?    Nov 15 /usr/bin/cygrunsrv
SYSTEM    1596    1328   ?    Nov 15 /usr/sbin/sshd
   Mau    1976       1   0    Nov 15 /usr/bin/bash
   Mau    3816    1976   0  10:24:40 /usr/bin/ps
```

La commande top

- Permet de superviser en temps réel l'activité du processeur.
- Donne la liste des processus en cours selon leur taux d'utilisation de la CPU.

```
10:28:39 up 1 day,  3:11,  3 users,  load average: 0.00, 0.00, 0.00
4 processes: 3 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  12.1% user,  13.1% system,   0.0% nice,  74.8% idle
Mem:    359920K total,   270692K used,    89228K free,        0K buffers
Swap:   121196K total, -3915296K used,  4036492K free,        0K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 3712 Mau        8   0  2056 3756    96 R     4.8  1.0   0:00 top
 1328 SYSTEM     8   0  1172  200     0 S     0.0  0.0   0:03 cygrunsrv
 1596 SYSTEM     8   0  1888  472    12 S     0.0  0.1   0:00 sshd
 1976 Mau        8   0  1704 1864    72 S     0.0  0.5   0:40 bash
```

Les signaux

- Un signal Unix est un message élémentaire transmis à un processus.
- Les signaux sont transmis lorsqu'un événement significatif se produit.
- Exemple : la touche ctrl-C envoie un signal d'interruption à tous les processus connectés à un terminal.
- Les différents signaux possibles sont identifiés par un nom et un numéro (de 1 à 16, ou de 1 à 32)
- Un signal peut être envoyé
  - Par le système
  - Par un autre processus
  - Par un utilisateur depuis une fenêtre de commande.

Liste des signaux les plus courants

2	INT	Ctrl - C
3	QUIT	Ctrl - Q
9	KILL	
17	STOP	Ctrl - Z
19	CONT	$ fg

Envoi d'un signal depuis une fenêtre de commande

- Pour envoyer un signal, il suffit de préciser :
  - L'identité du destinataire (PID)
  - La nature du message (No du signal)
- La commande Unix : kill

  `kill -s signal ident_processus`

  ou

  `kill -n ident_processus`

  (où n est un No de signal)
- Exmples :

  `kill -s STOP 1234`

  `kill -9 1234`

Les priorités

- La priorité d'un processus définit son taux d'élection au niveau de l'ordonnanceur
- Commande unix :

  `nice -n` *`num commande`*

- num dans 0..20
  - 0 : priorité normale
  - 20 priorité la plus basse

Communication entre processus

- Pour communiquer autrement que par des signaux, deux processus peuvent ouvrir un tube de communication :
- Exemple de tube lancé explicitement dans une fenêtre de commande :

  `$ commande_1 | commande_2`

  Le processus associé à la commande 1 ouvre un tube de communication avec le processus de la commande 2.

  Le processus 2 se positionne en attente de lecture tant qu'une nouvelle information n'est pas envoyée par le processus 1.
- On dit que :
  - Le processus 2 est connecté au processus 1
  - Les processus 1 et 2 sont synchronisés (le processus 2 a besoin du processus 1 pour s'exécuter, en cas de retard sur le procesus 1, le processus 2 prend aussi du retard)

Problèmes classiques

- L'accès aux ressources :
  - Si plusieurs processus veulent accéder à la même ressource (temps processeur, fichier disque,…) comment gérer au mieux ces accès?
  - Différencier les accès en lecture seule (plusieurs clients) et les accès en écriture (un seul client).
  - Eviter les interblocages (deux processus qui se bloquent lorsque l'un et l'autre attendent une information en provenance de l'autre)

Gestion des processus en Perl

Généralités

- Il existe plusieurs méthodes pour gérer les processus en Perl
  - Appel de commandes Unix
  - Exécution de processus
  - Ouverture/fermeture d'un canal de communication
  - Duplication de processus avec fork…

La fonction system

- Permet d'exécuter une commande Unix à partir d'un programme Perl
- Équivalent à l'exécution d'une commande Unix depuis la fenêtre de commandes (même syntaxe)
- Exemples :

```
print "Voici la date : ";
system "date";
print "\n";
```

  - Un processus enfant exécute la commande date
  - Le processus Perl est suspendu pendant son exécution
  - Le résultat de cette commande est écrit sur la sortie standard.

La fonction exec

- Exécute une commande externe sans revenir au programme initial!!

```
print "Voici la date : ";
exec "date";
print  "J'ai bien travaillé! \n";
          # la derniere ligne ne s'exécute pas
```

Autre exemple

```
print "voici la liste du répertoire personnel";
system 'ls -l $HOME';
Print "\n";
```

- La variable $HOME est une variable d'environnement Unix (elle n'appartient pas à Perl).
- Utilisation des apostrophes simples pour éviter l'interprétation de $HOME.
- Toutes les commandes Unix sont ainsi accessibles depuis un programme Perl.

Utilisation des apostrophes inverses

- Les apostrophes inverses permettent de capturer le résultat de l'appel à une commande Unix (en évitant de la voir apparaître sur la sortie standard).
- Exemple :

```
my $maintenant = `date`;
print "L'heure actuelle est : $maintenant";
```

L'apostrophe inverse permet d'interprétation de variables :

```
my $fonctions = qw { int rand sleep length hex
  sqrt umask};
my %documentation;

foreach (@fonctions){
  $documentation{$_}= `perldoc -t -f $_`;
}
```

Contexte de liste

```
my @liste_fichiers = `ls -l`;
```

Utilisation dans une boucle foreach :

```
@liste = `ls -l`;
#print @liste;

foreach (`ls -l`){
  my ($n,$fichier)= /[rwx\-\+]+\s+\d+\s+\w+\s+\w+\s+(\d+)
   \s+\w+\s+\d+\s+\d+\:\d+\s+(\w+.\w*.\w*)/;
}

foreach (sort keys %taille){
  print "Fichier : $_  \t\t taille : $taille{$_} \n";
  }
```

Particularités du système Unix

- Sur un système Unix, un fichier est plus généralement une interface de communication, c'est
  - dire un "interlocuteur" avec qui le programme peut dialoguer.
  - Fichiers réguliers
  - Fichiers spéciaux = répertoires ou interfaces
- Cet interlocuteur peut :
  - fournir des informations au programme (opération de "lecture")
  - recevoir des informations de la part du programme (opération d' "écriture")
- Pour établir la communication avec un fichier, il faut l'ouvrir (open)
- Pour interrompre la communication, il faut le fermer (close)

Ouverture d'un tube

- La fonction open permet d'ouvrir un processus enfant concurrent avec lequel le programme communique avec un tube (en lecture ou en écriture)
- En lecture :

```
open DATE, "date|";
```

- En écriture :

```
open MAIL, "|mail joe@egim-mrs.fr";
```

- Gestion des erreurs :

```
open DATE, "date|" or die "ouverture impossible! : $!";
```

Lecture/ecriture sur un tube

```
my $maintenant = <DATE>;
print MAIL, "Il est : $maintenant";
close MAIL;
```

Notion de démon et de client/serveur

- Un processus démon est un processus :
  - qui s'exécute en tâche de fond sur le système (il n'est pas associé à un terminal)
  - Le processus attend des signaux en provenance d'autres processus (les clients)
  - Lorsqu'un client cherche à se connecter, le processus active un tube de communication (socket) permettant d'échanger des informations avec le client.
- On parle d'architecture client/serveur.
- Un serveur peut établir des communications avec plusieurs clients. A chaque nouveau client, il produit un processus fils identique à lui même chargé de la communication avec un client particulier.

Socket

- Un socket est un tube permettant de communiquer à travers le réseau
- Deux méthodes de communication
  - Datagramme (à sens unique)
  - Flot (stream) : à double sens
- Les sockets permettent de mettre en place des architectures de type client/serveur

client

```
#!/usr/bin/perl

use IO::Socket;

$socket = IO::Socket::INET->new(     PeerAddr =>  "capucine",
                                PeerPort => 1234,
                                Proto    => "tcp",
                                Type     => SOCK_STREAM)
   or die "Impossible de se connecter : $@\n";
$socket ->autoflush(1);

$socket->send ("Hello!");

$socket->recv($reponse,300);

print "voici la reponse du serveur : $reponse \n";

close($socket);
```

serveur

```
#!/usr/bin/perl

use IO::Socket;
use IO::Handle;

$serveur = IO::Socket::INET->new(     LocalPort => 1234,
                                      Type      => SOCK_STREAM,
                                      Reuse     => 1,
                                      Listen    => 10 )
   or die "Impossible de devenir serveur : $@\n";

while ($client=$serveur -> accept()) {
   $client->recv($message,300);
   print "je viens de lire : $message \n";
   $client->send("Merci de votre message\n");
}

close($serveur);
```

Duplication de processus

- La duplication de processus est souvent utile dans les applications client/serveur
- La commande **`fork`** réalise la duplication :

```
defined (my $pid = fork) or die "fork impossible!";
if ($pid != 0) {
   print "Je suis l'enfant, mon identificateur est $$\n";
} else {
   print "Je suis le père, mon identificateur est $$\n";
}
```

- Pour distinguer le père du fils pendant l'exécution, on regarde la valeur de retour : 0 chez le père, no du père chez le fils.
- La variable $$ contient le no du processus.

Envoi de signaux

- Syntaxe:

  kill 2, 4321 or die "impossible d'envoyer SIGINT au processus 4321 : $!";

- Exemple : infanticide

```
defined (my $pid = fork) or die "fork impossible!";
if ($pid == 0) {
   print "Je suis l'enfant, mon identificateur est $$\n";
} else {
   print "Je suis le père, mon identificateur est $$\n";
   print "Mon fils est : $pid \n";
   kill 9, $pid;
}
```

Capture des signaux

- Permet de "résister" à l'envoi de certains signaux (ou d'exécuter une portion de code associée à certains signaux)

```
sub gestionnaire_int {
   die "je suis interrompu … argh!\n";
}
$SIG{INT} = 'gestionnaire_int';
$r=<STDIN>;
```