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)
```

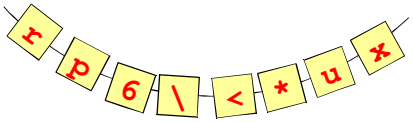
Expressions numériques

- 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
```


3. Les entrées/sorties

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)
```

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

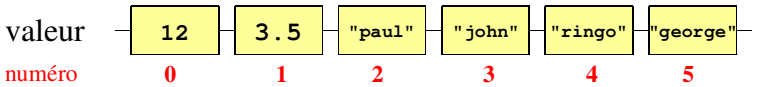
- Lorsque la dernière ligne est atteinte, <STDIN> retourne undef et la boucle while s'arrête

```
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

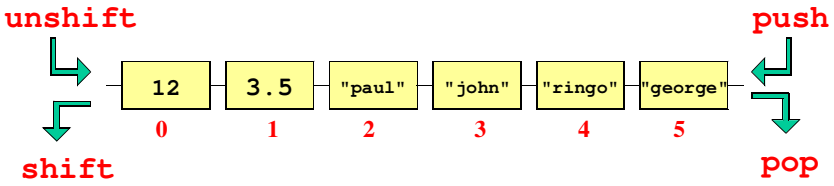
- Affectation en série sur variables scalaires :

```
($x,$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)
```

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
```


6. Sous-programmes et fonctions

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
```


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
```

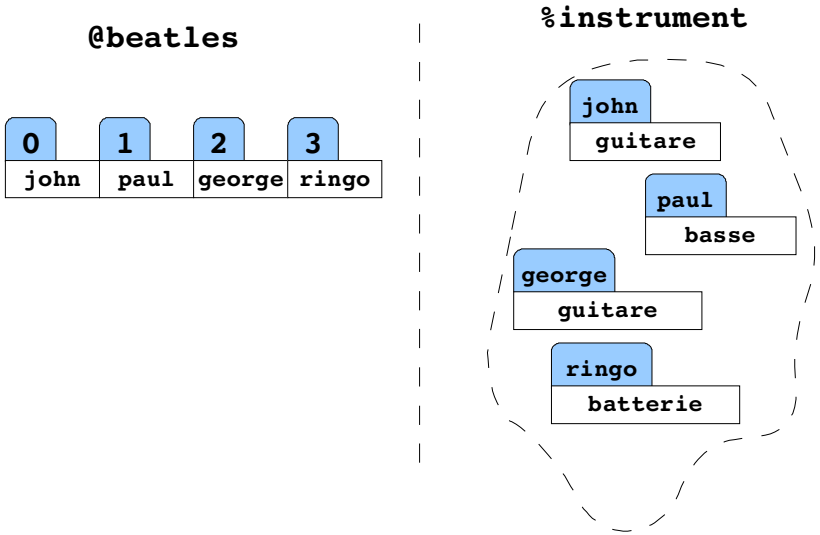
Appel de fonction

- 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
%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

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