Support de cours n°1

DESS TEXTE

Introduction

Ce support de cours rappelle les notions de base du langage Perl, notions utilisées dans le cadre du DESS TEXTE, module "Outils informatiques". Les divers points abordés seront successivement :

- les types de base de Perl,
- l'ouverture de fichiers en Perl,
- le parcours d'un tableau en Perl,
- le passage d'arguments à la ligne de commande (DOS ou Unix),
- l'utilisation de fonctions en Perl.

1 Les types de base en Perl

En Perl (à partir de la version 5), il existe 3 types de base : les **scalaires**, les **listes** (ou tableaux), et les **tables de hachage** (ou tableaux associatifs). Lors de la définition d'une variable, on utilise pour définir son type les symboles suivants :

- **$** pour désigner une variable scalaire, à voir comme une case mémoire contenant une donnée (un nombre entier ou réel, une chaîne de caractères, ou encore une adresse i.e. une référence),
- **@** pour désigner une liste. Une liste peut être vue comme un tableau indicé, c'est-à-dire qu'une liste contient un ensemble de données que l'on peut atteindre au moyen d'un indice. En perl, les tableaux sont indicé à partir de 0 (1^{er} élément),
- **%** pour désigner une table de hachage. Une table de hachage est un ensemble de couples (clé,valeur). Il est important d'insister sur l'unicité des clés : on ne peut pas utiliser deux fois la même clé, sous peine d'écraser la valeur correspondante.

1.1 Affectation

Comme dans de nombreux langages informatiques, perl utilise le symbole = pour l'affectation. Par affectation, on entend l'assignation d'une valeur à une variable.
Exemple : `$a = 5 ;` a pour conséquence de placer dans la case mémoire dénommée a la valeur entière 5. Si, par la suite, on a l'instruction `$b = $a ;`, on affecte à b la valeur de a, à savoir 5 (copie de valeur). Attention à l'ordre de l'affectation !

Pour affecter des valeurs à un tableau, plusieurs méthodes sont envisageables :

- on peut procéder par assignation directe : `$tab[0] = 1 ; $tab[1] = "oui" ;`
- on peut également affecter un ensemble de valeurs : `@tab = (1, "oui") ;`
- enfin, il existe la fonction bien pratique (lorsque l'ordre des éléments n'est pas imposé) : **push(@tab,X)**, ajoute X en fin du tableau *@tab*. A noter que X peut être un scalaire ($a par exemple), ou encore une liste (@a).

Enfin, en ce qui concerne l'affectation de tables de hachage, deux méthodes sont à retenir :

- on peut, comme pour les listes, procéder par assignation directe : `$hash{'oui'} = 1 ; $hash{'non'} = 0 ;`
- on peut affecter l'ensemble du hachage : `%hash = ("oui" => 1, "non" => 0) ;`

1.2 Accès

Lorsque l'on veut accéder à la valeur d'une variable, c'est immédiat : `print "$a ;"` affiche le contenu de la variable *a*. Par contre pour les listes, il faut désigner directement les éléments à accéder : `print "$tab[3]" ;` affiche le 4^e élément du tableau. Attention : `print @tab ;` provoque l'affichage des éléments du tableau collés les uns aux autres.
Enfin, en ce qui concerne les tableaux associatifs, il existe deux fonctions permettant d'accéder aux clés et valeurs : `keys %hash` renvoie la **liste** des clés du hachage (i.e. *keys* retourne un tableau). De même, pour les valeurs : `values %hash` retourne la **liste** des valeurs du hachage. Attention, ces deux fonctions renvoient un tableau, elles servent à lire les données d'un hachage, mais en aucun cas à les définir.

Remarque : perl dispose de variables prédéfinies, comme par exemple `$#tab` qui contient l'indice du dernier élément du tableau @tab.

2 L'ouverture de fichiers en perl

Il existe plusieurs manières d'ouvrir un fichier, suivant l'utilisation que l'on veut faire de celui-ci. Les différents modes d'ouverture d'un fichier sont principalement : l'ouverture en lecture, en écriture, en lecture/écriture, et en écriture en fin (appelée aussi *append*). L'ouverture d'un fichier se fait au moyen de la fonction **open**. Cette fonction prend en arguments respectifs un *descripteur* (un "nom symbolique" arbitraire) et un nom de fichier précédé d'un symbole décrivant le mode d'ouverture (on suppose que le nom du fichier est contenu dans la variable $fichier) :

Mode	Appel
lecture seule	`open(DESC,"<$fichier") ;`
écriture	`open(DESC,">$fichier") ;`
lecture/écriture	`open(DESC,"+<$fichier") ;`
écriture en fin	`open(DESC,"»$fichier") ;`

Afin de terminer proprement un programme (lors d'une erreur dans le nom d'un fichier par exemple), on utilise la fonction *die* prenant en argument un message (i.e. une chaîne de caractères).
Exemple d'ouverture de fichier en lecture :

```
open(DESC,"<\$fichier") || die "Erreur : \$! \n";
```

Ici, on ouvre le fichier via open **ou**, si une erreur empêchant l'ouverture survient, on termine le programme (fonction die) en affichant le message "Erreur... ;", à noter l'emploi de la variable perl prédéfinie *$!* contenant le message d'erreur renvoyé par le système d'exploitation.

Remarques :

1. en perl, le descripteur créé par la fonction `open` permet un parcours ligne par ligne d'un fichier : `while($ligne = <DESC>){...}`.
2. Après utilisation d'un descripteur pour parcourir un fichier, il convient de fermer le fichier au moyen de la fonction *close* : `close(DESC) ;`. Cela permet d'éviter des erreurs, si par la suite on veut reparcourir le fichier à partir d'une nouvelle boucle *while*.

3 Le parcours d'un tableau en perl

Pour parcourir les éléments d'un tableau, il existe principalement trois méthodes équivalentes, utiliser une boucle *for*, une boucle *foreach*, ou une boucle *while* :

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

foreach $elt (@tab) {
  print "$elt";
}

$i = 0;
while($i <= $#tab) {
  print "$tab[$i]";
  $i++;
}
```

Le choix de la boucle dépend, en règle générale, de ce l'on souhaite faire du tableau (foreach est très utile si l'on souhaite parcourir tous les éléments du tableau, for est adapté au cas où l'on connait les indices limites des éléments que l'on souhaite accéder).

4 Le passage d'arguments à la ligne de commande

Comme dans d'autres langages informatiques, tels que C, perl permet le passage d'informations au programme au moyen de la ligne de commande de l'interpréteur (fenêtre DOS ou shell Unix). Les données que l'on souhaite donner au programme perl doivent être saisies à la suite du nom du programme et séparés d'un espace, exemple : `perl monprog.perl argument1 argument2`. Perl accepte n'importe quel nombre d'arguments.
Que se passe-t-il ensuite ? Lorsque l'interpréteur Perl lit cette ligne, il crée un tableau nommé **@ARGV** (mot réservé), dans lequel il stocke les arguments : `argument1` en 1ère position ($ARGV[0]), `argument2` en 2e ($ARGV[1]), etc. Par simple lecture de ce tableau, vous pouvez récupérer, dans votre programme, les valeurs des arguments que l'utilisateur aura donné. Cela s'avère très utile par exemple pour transmettre des noms de fichiers (c'est ce qui rend votre programme portable).

5 L'utilisation des fonctions en perl

Une fonction est un ensemble d'instructions réalisant un traitement particulier et prenant 0, 1 ou plusieurs arguments. Pour information, on distingue, en informatique, une fonction d'une procédure par le fait qu'une fonction retourne un résultat alors qu'une procédure non (elle peut simplement modifier une variable dont l'adresse est passée en paramètre). En perl, la définition de fonctions (et procédures) se fait de la manière suivante :

```
sub ma_fonction {
  ...
  return @tab;
}
```

Nous définissons ainsi une fonction *ma_fonction* renvoyant le tableau *@tab*. On remarque que, bizarrement, on ne précise pas dans la définition de cette fonction les arguments qu'elle nécessite. En fait Perl traite les arguments d'une fonction de manière analogue aux arguments de la ligne de commande. Voyons ce qu'il se passe à l'appel d'une fonction sur un exemple :

```
$a = &additionne(3,2);

sub additionne {
  my($val1,$val2);
  my $res;
```

```
  $val1 = $_[0];
  $val2 = $_[1];
  $res = $val1 + $val2;
  return $res;
}
```

Tout d'abord, on remarque que l'appel à une fonction se fait via le symbole **&** : `$res = &ma_fonction($arg1,$arg2) ;`
Lorsque perl lit un appel à une fonction, il stocke ses arguments (on dit aussi paramètres) dans un tableau prédéfini nommé **@_**. On peut alors accéder à ces arguments dans le corps de la fonction en lisant les valeurs contenues dans ce tableau @_. Ici on a écrit une fonction `additionne` qui prend en arguments deux entiers et renvoie leur somme. `$val1` et `$val2` vont contenir repectivement les deux arguments donnés à la fonction lors de son appel (connus via `$_[0] et $_[1]`). Enfin le mot-clé `return` indique ce que la fonction renvoie, ici le scalaire `$res`. Bien entendu, dans notre programme, la variable *$a* va recevoir la valeur 5.

Remarques (importantes) :

1. Si l'on passe en arguments d'une fonctions deux listes, la liste des arguments @_ va concaténer ces deux listes et l'on perdra leur limites, d'où la nécessité d'utiliser dans ce cas des références (i.e. des scalaires contenant les adresses des listes à traiter).
2. Une fonction peut retourner divers types de données, un scalaire, une liste, ou encore un tableau associatif.