INF203

Memo bash

Variables utilisateur

Syntaxe pour la définition :
`variable=valeur` (pas d'espaces autour du `=`)
Pour la consultation, le nom de la variable est précédé de `$`.
Exemple :

```
numTP=42
if [ $numTP -ne 0 ]
then
     mkdir TP$numTP
fi
```

Variables d'environnement

Variables dont la valeur est conservée lors de l'invocation d'un sous shell.
Syntaxe pour la définition :
`export variable=valeur`
ou encore, après avoir défini la variable `toto` (par exemple) :
`export toto`
Un certain nombre de variables d'environement sont définies par le système lors de la connexion au compte (par exemple `$HOME`, `$PATH`, `$USER`, ...). ....

Variables spéciales définies par le shell

`$?`	code de retour de la dernière commande (0 si ok)
`$0`	nom du script
`$1` à `$9`	les éventuels 9 premiers arguments passés au script
`$#`	nombre d'arguments
`$*`	liste des arguments (à partir de `$1`)

Entrées-sorties

echo : affiche ses arguments sur la sortie standard.
Exemple :
`echo Vive INF203`

read : lecture d'une ligne sur l'entrée standard et affectation de chaque mot (resp. du reste de la ligne) à la variable correspondante donnée en argument du **read** (resp. la dernière variable).
Exemple :

```
echo "Nom et prénom ?"
read nom prenom reste
echo "merci $prenom $nom, reste de la ligne : $reste"
```

Développement, protection et substitution

Chaque ligne lue par le shell lors de l'interprétation de commandes est soumise au développement des variables avant son découpage en mots. Ce développement consiste à remplacer les noms de variable précédés d'un `$` par leur valeur.

Une fois le découpage en mots effectué, un développement des chemins est effectué par le shell. Ceci consite à remplacer les mots contenant des symboles spéciaux (qui sont alors appelés des motifs) par la liste des noms de fichier correspondant au motif. Si le shell ne trouve aucune possibilité de remplacement, les symboles spéciaux sont laissés inchangés. Les symboles spéciaux sont :

* peut être remplacé par un nombre quelconque de caractères. quelconques
? peut être remplacé par exactement un caractère quelconque.
[ensemble] peut être remplacé par exactement un caractère faisant partie de l'ensemble donné. L'ensemble est une séquence de caractères et/ou d'intervalles de caractères. Un intervalle de caractères est un caractère, suivi d'un tiret suivi d'un caractère supérieur au premier dans l'ordre des codes ASCII.

Le shell fourni également des moyens de protéger une partie de la ligne de commande de ces différents développements :

— un caractère précédé d'un *backslash* (le symbole \) perd son sens spécial vis-à-vis du développement.
Exemple :
```
echo \*          # affiche *
echo \$HOME      # affiche $HOME
```
— une chaîne de caractères entre simple *quotes* (le symbole ') n'est pas soumise au développement.
Exemple :
```
echo 'Is your home $HOME ?'          # affiche Is your home $HOME ?
```
— une chaîne de caractères entre double *quotes* (le symbole ") n'est pas soumise au développement de chemins.
Exemple :
```
echo "Is your home $HOME ?"          # affiche Is your home /home/toto ?
```

Enfin, le shell fourni un mécanisme appelé substitution de commande qui consiste à remplacer toute chaîne de caractères entre *backquotes* (le symbole `) par l'affichage resultant de l'exécution de la ligne de commande constituée par la chaîne en question.
Exemple :

```
echo `expr 1 + 3`         # affiche 4
toto=`expr $toto + 1`     # incrémentation de la valeur de toto
```

Extraction du nom d'un fichier sans l'extension et/ou sans le chemin complet : `basename`

Exemple :

```
fcomplet=nom.txt
prefixe=`basename $fcomplet .txt`
echo $fcomplet            #affiche nom.txt
echo $prefixe             #affiche nom

prefixe=`basename $fcomplet .c`
echo $prefixe             #affiche nom.txt

fcomplet=/Public/203_INF_Public/TP2/max2.c
suffixe=`basename $fcomplet`
echo $fcomplet            #affiche /Public/203_INF_Public/TP2/max2.c
echo $suffixe             #affiche max2.c
```

Les tests

Syntaxe : `[ expression ]` (ou `test expression`). **test** est une commande dont le code de retour est 0 si l'expression testée est vraie.

Expressions sur les fichiers et répertoires

Syntaxe : `option fich`
(attention aux espaces)

option	signification
`-e`	`fich` existe
`-s`	`fich` n'est pas vide
`-f`	`fich` est un fichier
`-d`	`fich` est un répertoire
`-r`	`fich` a le droit `r` (lecture)
`-w`	`fich` a le droit `w` (écriture)
`-x`	`fich` a le droit `x` (exécution)

Expressions sur les entiers

Syntaxe : $\texttt{n}_1$ `option` $\texttt{n}_2$
(attention aux espaces)

option	signification
-eq	$n_1 = n_2$
-ne	$n_1 \neq n_2$
-lt	$n_1 < n_2$
-gt	$n_1 > n_2$
-le	$n_1 \leq n_2$
-ge	$n_1 \geq n_2$

Expressions sur les chaines

Syntaxe : `option chaine` ou $\texttt{chaine}_1$ `option` $\texttt{chaine}_2$
(attention aux espaces)

option	signification
-z	`chaine` est vide
-n	`chaine` n'est pas vide
=	les 2 chaines sont identiques
!=	les 2 chaines sont différentes

Opérateurs booléens sur les expressions

Expression	Interprétation logique
$\texttt{expr}_1$ `-a` $\texttt{expr}_2$	ET
$\texttt{expr}_1$ `-o` $\texttt{expr}_2$	OU
`! expr`	NON

Exemples :

```
[ -d $dir  -a -x $dir ] # la variable dir représente un répertoire accessible en exécution

[ $x -lt 42 ] # la variable x représente un entier strictement inférieur à 42
```

Conditionnelle

Syntaxe :

```
if <condition>
then
    # à exécuter si la condition est vraie
    <suite de commandes 1>
else
    # à exécuter sinon
    <suite de commandes 2>
fi
```

La partie `else` est facultative. Pour imbriquer des `if` dans la partie `else` :

```
if <condition 1>
then
    <suite de commandes 1>
elif <condition 2>
then
    <suite de commande 2>
else
    <suite de commandes 3>
fi
```

La condition est déterminée par le code de retour d'une suite de commandes. On se contente ici de la commande test : Exemple :

```
mod=‘expr $1 % 2‘
if [ $mod -eq 0 ]
then
   echo $1 est pair
else
   echo $1 est impair
fi
```

Boucle `for`

Syntaxe :

```
for <variable> in <liste>
do
    <suite de commandes>
done
```

Exemples :

```
cd $HOME/INF203
for i in 1 2 3
do
    ls TP$i
done


for fich in *.o
do
    rm $fich
done
```

Case

Syntaxe :

```
case <valeur expression> in
     <valeur 1>) <suite de commandes 1> ;;
     <valeur 2>) <suite de commandes 2> ;;
     <valeur 3>) <suite de commandes 3> ;;
     *) <suite de commandes 4> ;;
esac
```

Exemple :

```
case $appreciation in
     A) echo "Très bien !" ;;
     B) echo "Pas mal ..." ;;
     C) echo "C'est moyen" ;;
     D) echo "Insuffisant, peut mieux fairë ;;
     E) echo "Il est temps de se mettre au travail" ;;
esac
```

Boucle `while`

Syntaxe :

```
while <condition>
do
    <suite de commandes>
done
```

Fonctions

Syntaxe :

```
<nom_fonction>() {
   <suite de commandes>
}
```

Exemple :

```
lire() {
        somme=0
        while read ligne
                do
                    echo "ligne " $ligne
                    somme=‘expr $somme + $lignè
                    echo "somme " $somme

                done
        }
lire < fichier_d_entiers
```