Mini tutoriel bash[1]

I- Variables et évaluation

Les variables sont stockées comme des chaînes de caractères.
Les variables *d'environnement* sont des variables exportées aux processus (programmes) lancés par le shell. Les variables d'environnement sont gérées par UNIX, elles sont donc accessibles dans tous les langages de programmation (voir plus loin).
Pour définir une variable :

```
$ var='ceci est une variable'
```

Attention : pas 'espaces autour du signe égal. Les quotes (apostrophes) sont nécessaires si la valeur contient des espaces. C'est une bonne habitude de toujours entourer les chaînes de caractères de quotes.
Pour utiliser une variable, on fait précéder son nom du caractère "`$`" :

```
$ echo $var
```

ou encore :

```
$ echo 'bonjour, ' $var
```

Dans certains cas, on doit entourer le nom de la variable par des accolades :

```
$ X=22
$ echo Le prix est ${X}0 euros
```

affiche "Le prix est de 220 euros" (sans accolades autour de X, le shell ne pourrait pas savoir que l'on désigne la variable X et non la variable X0).
Lorsqu'on utilise une variable qui n'existe pas, le bash renvoie une chaîne de caractère vide, et n'affiche pas de message d'erreur (contrairement à la plupart des langages de programmation, y compris csh) :

```
$ echo Voici${UNE_DROLE_DE_VARIABLE}!
```

affiche "Voici!".
Pour indiquer qu'une variable doit être exportée dans l'environnement (pour la passer aux commandes lancée depuis ce shell), on utilise la commande `export` :

```
$ export X
```

ou directement :

```
$ export X=22
```

1 Inspiré d'un document de Emmanuel Viennet

Évaluation, guillemets et quotes

Avant évaluation (interprétation) d'un texte, le shell substitue les valeurs des variables. On peut utiliser les guillemets (`"`) et les quotes (`'`) pour modifier l'évaluation :

- les guillemets permettent de grouper des mots, sans supprimer le remplacement des variables. Par exemple, la commande suivante ne fonctionne pas :

  ```
  $ x=Hauru no
  bash: no: command not found
  ```

  Avec des guillemets c'est bon :

  ```
  $ x="Hauru no"
  ```

  On peut utiliser une variable entre guillemets :

  ```
  $ y="Titre: $x Ugoku Shiro"
  $ echo $y
  Titre: Hauru no Ugoku Shiro
  ```

- les quotes (apostrophes) groupes les mots et suppriment toute évaluation :

  ```
  $ z='Titre: $x Ugoku Shiro'
  $ echo $z
  Titre: $x Ugoku Shiro
  ```

Expressions arithmétiques

Normalement, bash traite les valeurs des variables comme des chaînes de caractères. On peut effectuer des calculs sur des nombres entiers, en utilisant la syntaxe `$(( ... ))` pour délimiter les expressions arithmétiques:

```
$ n=1
$ echo $(( n + 1 ))
2
$ p=$((n * 5 / 2 ))
$ echo $p
2
```

Découpage des chemins

Les scripts shell manipulent souvent chemins (*pathnames*) et noms de fichiers. Les commandes `basename` et `dirname` sont très commodes pour découper un chemin en deux partie (répertoires, nom de fichier) :

```
$ dirname /un/long/chemin/vers/toto.txt
```

```
/un/long/chemin/vers
$ basename /un/long/chemin/vers/toto.txt
toto.txt
```

Évaluation de commandes

Il est courant de stocker le résultat d'une commande dans une variable. Nous entendons ici par "résultat" la *chaîne affichée par la commande*, et non son code de retour.
Bash utilise plusieurs notations pour cela : les *back quotes* (`) ou les parenthèses :

```
$ REP=`dirname /un/long/chemin/vers/toto.txt`
$ echo $REP
/un/long/chemin/vers
```

ou, de manière équivalente :

```
$ REP=$(dirname /un/long/chemin/vers/toto.txt)
$ echo $REP
/un/long/chemin/vers
```

(attention encore une fois, pas d'espaces autour du signe égal).
La commande peut être compliquée, par exemple avec un tube :

```
$ Fichiers=$(ls /usr/include/*.h | grep std)
$ echo $Fichiers
/usr/include/stdint.h /usr/include/stdio_ext.h /usr/include/stdio.h
/usr/include/stdlib.h /usr/include/unistd.h
```

Découpage de chaînes

Bash possède de nombreuses fonctionnalités pour découper des chaînes de caractères. L'une des plus pratiques est basée sur des motifs.
La notation ## permet d'éliminer la *plus longue* chaîne en correspondance avec le motif :

```
$ Var='tonari no totoro'
$ echo ${Var##*to}
ro
```

ici le motif est `*to`, et la plus longue correspondance "tonari no toto"[1]. Cette forme est utile pour récupérer l'extension (suffixe) d'un nom de fichier :

```
$ F='rep/bidule.tgz'
$ echo ${F##*.}
tgz
```

La notation # (un seul #) est similaire mais élimine la *plus courte* chaîne en correspondance :

```
$ Var='tonari no totoro'
```

```
$ echo ${Var#*to}
nari no totoro
```

De façon similaire, on peut éliminer la fin d'une chaîne :

```
$ Var='tonari no totoro'
$ echo ${Var%no*}
tonari
```

Ce qui permet de supprimer l'extension d'un nom de fichier :

```
$ F='rep/bidule.tgz'
$ echo ${F%.*}
rep/bidule
```

`%` prend la plus courte correspondance, et `%%` prend la plus longue :

```
$ Y='archive.tar.gz'
$ echo ${Y%.*}
archive.tar
$ echo ${Y%%.*}
archive
```

Arguments de la ligne de commande

Un script bash est un simple fichier texte exécutable (droit `x`) commençant par les caractères `#!/bin/bash` (doivent être les premiers caractères du fichier).
Voici un exemple de script :

```
#!/bin/bash

if [ "${1##*.}" = "tar" ]
then
        echo $1 est une archive tar
else
        echo $1 n\'est pas une archive tar
fi
```

Ce script utilise la variable `$1`, qui est le premier argument passé sur la ligne de commande.

Lorsqu'on entre une commande dans un shell, ce dernier sépare le nom de la commande (fichier exécutable ou commande interne au shell) des arguments (tout ce qui suit le nom de la commande, séparés par un ou plusieurs espaces). Les programmes peuvent utiliser les arguments (options, noms de fichiers à traiter, etc).
En bash, les arguments de la ligne de commande sont stockés dans des variables spéciales :

`$0, $1, ...`	les arguments
`$#`	le nombre d'arguments
`$*`	tous les arguments

Le programme suivant illustre l'utilisation de ces variables :

```
#!/bin/bash
```

```
echo 'programme  :' $0
echo 'argument 1 :' $1
echo 'argument 2 :' $2
echo 'argument 3 :' $3
echo 'argument 4 :' $4
echo "nombre d'arguments :" $#
echo "tous:" $*
```

Exemple d'utilisation, si le script s'appelle "`myargs.sh`" :

```
$ ./myargs.sh un deux trois
programme  : ./myargs.sh
argument 1 : un
argument 2 : deux
argument 3 : trois
argument 4 :
nombre d'arguments : 3
tous: un deux trois
```

II- Structures de contrôle

Alternative (si alors sinon)

L'instruction `if` permet d'exécuter des instructions si une condition est vraie. Sa syntaxe est la suivante :

```
if [ condition ]
then
    action
fi
```

`action` est une suite de commandes quelconques. L'indentation n'est pas obligatoire mais très fortement recommandée pour la lisibilité du code. On peut aussi utiliser la forme complète :

```
if [ condition ]
then
    action1
else
    action2
fi
```

ou encore enchaîner plusieurs conditions :

```
if [ condition1 ]
then
    action1
elif [ condition2 ]
then
    action2
elif [ condition3 ]
then
```

```
    action3
else
    action4
fi
```

Opérateurs de comparaison

Le shell étant souvent utilisé pour manipuler des fichiers, il offre plusieurs opérateurs permettant de vérifier diverses conditions sur ceux-ci : existence, dates, droits. D'autres opérateurs permettent de tester des valeurs, chaînes ou numériques. Le tableau ci-dessous donne un aperçu des principaux opérateurs :

Opérateur	Description	Exemple
Opérateurs sur des fichiers		
-e *filename*	vrai si *filename* existe	`[ -e /etc/shadow ]`
-d *filename*	vrai si *filename* est un répertoire	`[ -d /tmp/trash ]`
-f *filename*	vrai si *filename* est un fichier ordinaire	`[ -f /tmp/glop ]`
-L *filename*	vrai si *filename* est un lien symbolique	`[ -L /home ]`
-r *filename*	vrai si *filename* est lisible (r)	`[ -r /boot/vmlinuz ]`
-w *filename*	vrai si *filename* est modifiable (w)	`[ -w /var/log ]`
-x *filename*	vrai si *filename* est exécutable (x)	`[ -x /sbin/halt ]`
file1 -nt *file2*	vrai si *file1* plus récent que *file2*	`[ /tmp/foo -nt /tmp/bar ]`
file1 -ot *file2*	vrai si *file1* plus ancien que *file2*	`[ /tmp/foo -ot /tmp/bar ]`
Opérateurs sur les chaînes		
-z *chaine*	vrai si la *chaine* est vide	`[ -z "$VAR" ]`
-n *chaine*	vrai si la *chaine* est non vide	`[ -n "$VAR" ]`
chaine1 = *chaine2*	vrai si les deux chaînes sont égales	`[ "$VAR" = "totoro" ]`
chaine1 != *chaine2*	vrai si les deux chaînes sont différentes	`[ "$VAR" != "tonari" ]`
Opérateurs de comparaison numérique		
num1 -eq *num2*	égalité	`[ $nombre -eq 27 ]`
num1 -ne *num2*	inégalité	`[ $nombre -ne 27 ]`
num1 -lt *num2*	inférieur (<)	`[ $nombre -lt 27 ]`
num1 -le *num2*	inférieur ou égal (< =)	`[ $nombre -le 27 ]`
num1 -gt *num2*	supérieur (>)	`[ $nombre -gt 27 ]`
num1 -ge *num2*	supérieur ou égal (> =)	`[ $nombre -ge 27 ]`

Quelques points délicats doivent être soulignés :

- Toutes les variables sont de type chaîne de caractères. La valeur est juste convertie en nombre pour les opérateurs de conversion numérique.
- Il est nécessaire d'entourer les variables de guillemets (`"`) dans les comparaisons. Le code suivant affiche "OK" si `$var` est égale à "tonari no totoro" :

  ```
  if [ "$myvar" = "tonari no totoro" ]
  ```

```
then
   echo "OK"
fi
```

Par contre, si on écrit la comparaison comme `if [ $myvar = "tonari no totoro" ]` le shell déclenche une erreur si `$myvar` contient plusieurs mots. En effet, la substitution des variables a lieu avant l'interprétation de la condition.

Boucle for

Comme dans d'autre langages (par exemple python), la boucle for permet d'exécuter une suite d'instructions avec une variable parcourant une suite de valeurs. Exemple :

```
for x in un deux trois quatre
do
    echo x= $x
done
```

affichera :

```
x= un
x= deux
x= trois
x= quatre
```

On utilise fréquemment for pour énumérer des noms de fichiers, comme dans cet exemple :

```
for fichier in /etc/rc*
do
   if [ -d "$fichier" ]
   then
       echo "$fichier (repertoire)"
   else
       echo "$fichier"
   fi
done
```

Pour faire une boucle `for` un certain nombre de fois (comme en C par exemple) :

```
for i in $(seq 1 1 10)
do
   echo "le nombre$i"
done
```

Ou encore, pour traiter les arguments passés sur la ligne de commande :

```
#!/bin/bash

for arg in $*
do
  echo $arg
done
```

Boucle while

La boucle while permet d'exécuter une suite d'instructions tant qu'une condition est vérifiée.

```
while [ condition ]
do
  action1
  action2 ...
done
```

Exemple :

```
x=1
while [ $x -le 5 ]
do
  echo "x= $x"
  x=$(( $x + 1 ))
done
```

Instruction case

L'instruction case permet de choisir une suite d'instruction suivant la valeur d'une expression :

```
case "$x" in
  go)
      echo "demarrage"
      ;;
  stop)
      echo "arret"
      ;;
   *)
      echo "valeur invalide de x ($x)''
esac
```

Noter les deux `;` pour signaler la fin de chaque séquence d'instructions.

III- Définition de fonctions

Il est souvent utile de définir des fonctions. La syntaxe est simple :

```
mafonction() {
   echo "appel de mafonction..."
}

mafonction
mafonction
```

qui donne :

```
appel de mafonction...
appel de mafonction...
```

Voici pour terminer un exemple de fonction plus intéressant :

```
tarview() {
    echo -n "Affichage du contenu de l'archive $1 "
    case "${1##*.}" in
        tar)
           echo "(tar compresse)"
           tar tvf $1
        ;;
        tgz)
           echo "(tar compresse gzip)"
           tar tzvf $1
        ;;
        bz2)
           echo "(tar compresse bzip2)"
           cat $1 | bzip2 -d | tar tvf -
        ;;
        *)
           echo "Erreur, ce n'est pas une archive"
        ;;
    esac
}
```

Plusieurs points sont à noter :

- `echo -n` permet d'éviter le passage à la ligne;
- La fonction s'appelle avec un argument (`$1`)

  ```
  tarview toto.tar
  ```

Table des matières