Programmation Shell

1 Shell et environnement

Il n'existe pas un shell mais plusieurs qui se différencient par la syntaxe des commandes et par leurs possibilités (historique, complétion de commandes, commandes d'édition...). on peut citer :

- Bourne shell (*sh*) 1970 (Unix version 1, puis version 7)
- C-shell (*csh*) 1982 (Unix BSD)
- Korn shell (*ksh*) 1983 (Unix system V)
- Turbo C shell (*tcsh*) 1989 (FreeBSD, premier Unix en *open source*)
- Bourne again shell (*bash*) 1990 (Linux)

Chaque utilisateur a un shell par défaut qu'il peut changer dans ses fichiers de configuration. *bash* est le plus classique car c'est le shell par défaut sur Linux et MacOS.

Un shell est en même temps un interpréteur interactif de commandes et un interpréteur d'un langage de programmation de commandes. Un programme shell est appelé « script ». C'est un fichier texte contenant des variables, des commandes et des structures de contrôle.

- **Comment exécuter un script shell ?**

1. `sh nom_script`
   Lance une copie du programme `sh` et lui fait exécuter le script. On peut bien sûr indiquer n'importe quel shell disponible sur le système.
2. `. nom_script`
   Contrairement au cas précédent, cela ne lance pas de nouveau shell mais exécute le script dans le shell courant (il n'a alors pas besoin d'être exécutable). Cela permet de modifier des variables d'environnement du shell courant (et devrait être limité à cet usage).
3. `./nom_script`
   Si le fichier script est **exécutable** (par exemple avec `chmod +x nom_script`), ceci lance un copie du shell actuel et exécute le script. C'est la manière la plus classique d'exécuter un script. Attention, la syntaxe du script ne correspond pas forcément à celle du shell si elles sont de deux types différents.
   On peut forcer le type de shell à lancer en mettant en première ligne du script le nom de l'interpréteur : `#!/bin/sh` ou `#!/bin/csh`... On est alors sûr que c'est le bon shell qui sera lancé (c'est-à-dire que la syntaxe du script correspondra) sous réserve que celui-ci soit disponible sur la machine...
   De façon plus générale, une telle commande en début de fichier texte (appelée *shebang*) permet, lors de son « exécution », de lancer le bon programme qui va interpréter les commandes dans le fichier. On peut le faire avec un shell comme ci-dessus mais aussi avec d'autres langages : `#!/usr/bin/perl`, `#!/usr/bin/python`...
   On peut remarquer qu'on ne lance pas le script simplement en indiquant son nom relatif mais bien avec son chemin absolu (grâce au `./` placé devant son nom). L'explication, liée à la variable PATH, se trouve un peu plus loin.

- **Variables**
  Un shell utilise des variables qui peuvent être définies en ligne de commande ou dans un script.
  - Variables locales (au shell)

    `x=truc` (sh/ksh) `set x=truc` (csh)

  - Variables globales

    `x=truc`
    `export x` (sh/ksh) `setenv x truc` (csh)

    Les variables globales sont accessibles par le shell et par tout processus lancé par le shell.
    Le préfixe `$` permet de récupérer la valeur de la variable :

    `x=$truc` (x prend la valeur de la variable `truc` et pas la valeur *truc* directement)
    `echo $x` (permet d'afficher la valeur de la variable x)
    La commande `printenv` (sh/csh) ou `setenv` (csh) sans argument permet d'afficher toutes les variables globales connues du shell. Voici quelques variables classiques :

    SHELL Contient le nom du shell de départ.

    HOME Contient le chemin d'accès au répertoire de départ de l'utilisateur

    HOSTNAME Contient le nom de la machine ; HOST sous *csh*.

    USER/LOGNAME Contient le nom de l'utilisateur.

    PWD Contient le chemin d'accès du répertoire courant.

    PATH Contient la liste des répertoires dans lesquels le shell cherche les programmes à exécuter.

Ce dernier ensemble de répertoires est important puisque c'est grâce à lui que le shell va trouver un programme à partir de son nom relatif. Il est classiquement formé des répertoires `/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin` et autres.

On remarque que le répertoire courant « `.` » n'apparaît pas dans la liste ci-dessus. Cela veut dire que le shell ne pourra pas exécuter un programme qui se trouve dans le répertoire courant même si on le lance en tapant son nom ! En effet, le nom étant un nom relatif, le shell va parcourir l'ensemble des répertoires du PATH pour le trouver et donc ne parcourt pas le répertoire courant. On peut remédier à ce problème en lançant systématiquement les programmes par `./nom_programme`, ce qui revient à donner le nom absolu mais ce n'est pas pratique. On peut aussi changer les fichiers de configuration (`.bash_profile` par exemple) pour inclure « `.` » dans le PATH (par la commande `PATH=$PATH:.`).

Attention, le shell parcourt les répertoires du PATH dans l'ordre indiqué. Ainsi les commandes `PATH=$PATH:.` et `PATH=.:$PATH` n'ont pas le même effet. Dans cette dernière, le répertoire courant est parcouru en premier pour trouver le nom de la commande : imaginons que votre programme s'appelle `test` (ou tout autre nom identique au nom d'une commande Unix) ; dans le dernier cas, votre répertoire étant parcouru d'abord, c'est bien votre programme qui est exécuté (mais dans ce répertoire, vous n'aurez plus facilement accès à la commande Unix de même nom) ; dans le premier cas, ce sera toujours la commande Unix qui sera exécutée et jamais votre programme (à moins de donner son nom absolu) !

2 Écriture de scripts shell

Un script shell permet de faire une extension des commandes de base. Il s'agit d'un fichier texte que le script va exécuter. On va pouvoir y mettre des variables, des commandes et des structures de contrôle pour gérer l'exécution du script. Pour lancer l'exécution d'un script `nom_Scr`, on peut soit taper `sh nom_Scr` (qui lance une copie de `sh` [ou d'un autre shell indiqué] et exécute le script), soit écrire `. nom_Scr` qui l'exécute dans le shell courant mais le plus pratique est de pouvoir lancer le script simplement en indiquant son nom `nom_Scr`. Il faut alors que le script soit exécutable en modifiant explicitement les droits d'accès du fichier texte contenant le script et que le chemin d'accès soit dans la variable `PATH` (sinon, il faut le lancer avec la commande `./nom_Scr`).

- **Arguments**

Un script étant une commande, on peut le lancer en ajoutant des arguments après son nom. Il faut alors pouvoir les récupérer dans le script. Des variables spéciales permettent de faire ça :

`$#` : renvoie le nombre d'arguments passés avec la commande.
`$n` : renvoie le n^{e} argument.
`$*` : renvoie une liste composée de tous les arguments (utilisable par exemple dans un `for`).

- **Commentaires**

Des commentaires peuvent être mis dans le script en les faisant précéder de `#`. Un `#!` sur la première ligne indique le nom du shell à lancer pour exécuter le script.

- **Chaînes de caractères**

Une chaîne de caractères indiquée entre apostrophes (`'`) est prise de façon littérale.

Une chaîne de caractères entre guillemets (`"`) est utilisée après substitution des variables : `"Bonjour $1"` correspondra à une chaîne où le `$1` aura été remplacé par le premier argument de la commande.

Une chaîne de caractères entre apostrophes inverses (`` ` ``) est remplacée par le résultat de l'exécution de la commande placée dans la chaîne : `echo` `` `ls | wc -l` `` affiche le nombre de fichiers du répertoire courant.

Si la variable `$R` contient la valeur `nom_rep` :
`echo 'ls $R | wc -l'` affiche la chaîne `ls $R | wc -l` *(affichage littéral)* ;
`echo "ls $R | wc -l"` affiche la chaîne `ls nom_rep | wc -l` *(remplacement)* ;
`echo` `` `ls $R | wc -l` `` affiche le nombre d'éléments du répertoire `nom_rep` *(exécution).*

- **Gestion des espaces**

Le shell est très strict sur la gestion des espaces dans un script. Chaque commande a des arguments et ceux-ci doivent être séparés par des espaces. De même, le caractère `[` (resp. `]`), indiquant un test doit être suivi (resp. précédé) d'une espace. En revanche, lors d'une affectation à une variable, le symbole = ne doit être ni précédé ni suivi par des espaces.

- **Conditions**

Une condition est encadrée par les caractères `[` et `]` (séparés de l'expression par des espaces). En voici quelques exemples :

– Conditions sur les fichiers

`-e nom`	vraie si `nom` est un fichier ou répertoire existant
`-f nom`	vraie si `nom` est un fichier ordinaire
`-d nom`	vraie si `nom` est un répertoire ordinaire
`-r nom`	vraie si `nom` est accessible en mode lecture
`-w nom`	vraie si `nom` est accessible en mode écriture

– Conditions sur les chaînes

`ch1`	vraie si la chaîne `ch1` n'est pas la chaîne nulle
`-z ch1`	vraie si la chaîne `ch1` est nulle
`ch1 = ch2`	vraie si les deux chaînes sont égales
`ch1 != ch2`	vraie si les deux chaînes sont différentes

– Conditions sur les nombres

`nbr1 op nbr2` compare les deux nombres indiqués

`op` peut prendre comme valeurs possibles :

`-eq`	*(equal)* égal
`-ne`	*(not equal)* différent
`-gt`	*(greater than)* strictement plus grand
`-ge`	*(greater or equal)* plus grand ou égal
`-le`	*(less or equal)* plus petit ou égal
`-lt`	*(less than)* strictement plus petit

– Combinaisons logiques

`! cond`	négation de la condition
`cond1 -a cond2`	*(and)* et logique entre deux conditions
`cond1 -o cond2`	*(or)* ou logique entre deux conditions
`(cond)`	regroupe une condition

Ainsi, la condition `[ -f $1 -o -d $1 ]` est vraie si l'argument est un fichier ou un répertoire ; la condition `[$# -ge 1 -a -f $1]` est vraie s'il y a au moins un argument et que le premier est un fichier.

- **Tests et boucles**

Un test simple ou une boucle s'écrit :

```
if [ expr ]
then
  commandes
fi
```

```
while [ expr ]
do
  commandes
done
```

```
for var in liste
do
  commandes
done
```

Ainsi, le script suivant teste si un fichier (dont le nom est passé en paramètre) existe et le déplace en le renommant avant de créer un fichier vide de même nom.

```
#!/bin/bash

if [ -e $1 ]
then
    mv $1 $1.backup
fi
touch $1
```

Et le script suivant ne fait rien d'autre que réafficher chacun des arguments passés à la commande.

```
#!/bin/bash

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

2.1 Script compteur

On a vu qu'avec `ls | wc -l`, on pouvait compter le nombre de fichiers dans le répertoire courant. On souhaite maintenant restreindre ce comptage aux fichiers ayant une extension donnée (par exemple `ls *.c | wc -l` va compter touts les fichiers ayant `.c` comme extension).

Exercice 1
Écrire un script qui prend un argument (l'extension) et renvoie le nombre de fichiers ayant cette extension.

2.2 Script Verlan

Les commandes `head` et `tail` extraient les premières et dernières lignes d'un fichier :
`head -n i fichier` extrait les *i* premières lignes du fichier (on écrit aussi `head -i`).
`tail -n i fichier` extrait les *i* dernières lignes du fichier (on écrit aussi `tail -i`).
`tail -n +i fichier` extrait les lignes du fichier en partant de la i^e.

Par exemple, `tail -n +2` extrait les lignes à partir de la deuxième, c'est-à-dire affiche le fichier sans la première ligne.
On peut les combiner pour extraire une ligne quelconque :
`head -n 5 fichier | tail -n 1` va isoler la cinquième ligne du fichier.

On peut alors définir une fonction qui a pour but de renvoyer la i^e ligne d'un fichier dont le nom est dans une variable globale :

```
lire_ligne() {
    ligne=`head -n $i $file | tail -n 1`
}
```

Exercice 2 *Écrire maintenant un script qui liste un fichier à l'envers en extrayant la dernière ligne, puis l'avant-dernière...*

Il faut faire attention que `wc` compte les lignes en comptant les *retours-chariot*. Si le fichier que l'on veut lister ne se termine pas par un retour-chariot, le script va compter une ligne de moins et ne va pas afficher la dernière ligne. On peut donc le modifier en incrémentant `i` de 1 avant la boucle.

2.3 Script liste

Exercice 3
Écrire un script qui prend un répertoire en argument et liste son contenu en indiquant le type pour chaque élément.

2.4 Script liste2

Exercice 4
Étendre le script précédent à un nombre quelconque d'arguments, chacun devant être listé. On pourra utiliser la variable `$*` *qui renvoie une liste formée de tous les arguments.*

2.5 Script décryptage MD5

Exercice 5 *Écrire un script qui compare la version cryptée d'une liste de mots, avec un mot de passe crypté grâce à la commande* `md5sum`*. Utilisez la liste mots disponible dans* `/usr/share/dict/words` *pour décoder le mot de passe suivant: 351a7b5994f2b820a91812722119afca*