Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

Programmation Shell

Table des matières

Programmation Shell....1
I- Introduction....2
Shell ?....2
Le Bash....2
II- Premiers scripts Shell....3
Syntaxe des scripts Shell....3
Bonjour !....3
Exécuter un script....3
Succession de commandes....4
III- Variables et opérations....4
Déclaration de variables....4
Utilisation des variables :....5
Opérations et opérateurs arithmétiques :....5
Variables prédéfinies par le Bash....5
IV- Saisir une information et la stocker dans une variable....6
V- Structures conditionnelles....6
Tests avec if :....6
Else, elif....7
Plusieurs conditions :....7
Opérateurs de comparaison :....8
Case :....9
Select :....9
VI- Boucles....10
La boucle while : tant que 10
La boucle until : jusqu'à 11
La boucle for :....11
VII- Passage de paramètres....12
VIII- Redirections....13
Commande1| commande2....13
Commande1 `commande2`....13
Redirection vers une autre sortie > :....14
Redirection des erreurs 2> :....14
Redirection des entrées < :....14
IX- Fonctions....15
X- Sur Internet 15
Cours et / ou TP sur la programmation Bash :....15

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

I- Introduction

Shell ?

Le Shell (coquille en anglais) est le composant du système d'exploitation qui permet de interpréter les commandes tapées au clavier (on l'appelle aussi **interpréteur de commandes**). Aujourd'hui le Shell est en grande partie remplacé par les programmes fonctionnant avec une interface graphique mais on s'en sert surtout pour automatiser des tâches grâce à la programmation.

Parmi les Shells les plus connus, il y a sous Windows 9X **command.com**, sous NT et suivants **cmd.exe** et pour la partie graphique **explorer.exe**. Sous Unix il en existe un certain nombre, dont les plus connus sont les suivants : Bourne Shell, **Bourne Again Shell**, Z Shell, C Shell, Korn Shell, ...

A l'heure actuelle, des langages comme **Perl**, **Ruby** ou **Python** prennent la relève des Shells sans toutefois les remplacer totalement.

Le Bash

Le Bash reprend le shell originel de Linux (Bourne Shell) et y intégre des amélioration provenant d'autres Shells comme le Ksh et le Csh.

Ce programme est libre (licence GPL) et a été porté sur Windows (**Cygwin**).

Lors des TP sur la programmation Shell, nous utiliserons toujours le Bash.

Si vous avez besoin une page de manuel existe pour le Bash : man bash !

Ce site en est la traduction française :
http://www.linux-france.org/article/man-fr/man1/bash-1.html

Quel Shell utilisez-vous ?

Dans le fichier /etc/passwd, il y a certaines informations concernant les utilisateurs et notamment le shell qu'ils utilisent, par exemple :
nico:x:1000:1000:nico:/home/nico:**/bin/bash**

Pour savoir tous les shells qui sont disponibles sur votre machine, vous pouvez taper :
ls /bin | grep sh

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

II- Premiers scripts Shell

Syntaxe des scripts Shell

Un script Shell n'est rien d'autre qu'un fichier texte dans lequel sont inscrites un certain nombre de commandes compréhensibles par votre interpréteur de commandes. Etant donné que Linux ne prend pas en compte les extensions des fichiers, vous pouvez nommer vos fichier de script comme vous voulez.

Pour indiquer qu'il s'agit bien d'un script et non pas d'un fichier texte classique, la première ligne doit toujours commencer par le nom de l'interpréteur de commandes :

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

... si vous utilisez bash

Vous pouvez écrire autant de lignes de commentaires que vous voulez, pour cela, faites débuter votre ligne ou votre fin de ligne par #.

```
#!/bin/sh
echo Bonjour # Un commentaire
# Un autre commentaire
```

Bonjour !

Pour créer votre premier script qui affichera Bonjour à l'écran, vous allez éditer un fichier que vous appelez script1 avec vi et vous allez taper ceci à l'intérieur :

```
#!/bin/bash
echo Bonjour
```

Exécuter un script

Pour exécuter le script que vous venez de créer, vous devrez d'abord vérifier que vous avez le droit d'exécuter ce fichier (normalement, non !). Pour cela faites un ls -l du répertoire dans lequel se trouve le fichier.

```
ls -l
-rwxr-xr-x ...          monfichier
```

Vous devriez avoir un x pour avoir le droit d'exécution, si ce n'est pas le cas, donnez à votre fichier le droit d'être exécuté en tapant :

```
chmod +x monfichier
```

Cette opération n'est à faire qu'une fois, si vous avez le droit d'exécution, pas besoin de répéter cette étape.

Vous pourrez alors l'exécuter en tapant :

```
./monfichier
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

Succession de commandes

Maintenant que vous avez essayé un script basique avec la commande echo, vous pouvez ajouter à la suite autant de commandes shell que vous voulez. Par exemple :

```
#!/bin/sh
clear
echo -n Vous utilisez le système
uname
echo
echo -n La version de votre noyau est
uname -r
```

Ce script va afficher quelque chose comme ça à l'écran :

```
Vous utilisez le système Linux
La version de votre noyau est 2.6.8-2-386
```

III- Variables et opérations

Déclaration de variables

Il n'y a pas besoin de déclarer les variables comme en langage C, par exemple. On peut directement affecter une valeur à une variable de cette façon :
variable=valeur
Par exemple :
nom=Dupont

> Les noms de variables doivent suivre les règles suivantes :
> Éviter les caractères spéciaux, ne pas commencer par un chiffre.

Utilisation des variables :

Pour utiliser les variables qui ont été affectées, on utilise le signe $.
echo $nom # Ceci affichera Dupont à l'écran

Opérations et opérateurs arithmétiques :

Pour faire des opérations sur des variables, vous pouvez utiliser la commande expr.
Cette commande s'utilise avec des variables de type numérique, par exemple :

```
expr 3 « + » 5        # Affichera 8 à l'écran
expr $a « * » $b      # Affichera 6 à l'écran si $a vaut 3 et $b vaut 2
```

Elle peut aussi s'utiliser avec des chaînes de caractères :

```
expr index $email @     # cherche s'il y a un @ dans la variable $email
                        # et renvoie l'emplacement de ce caractère
expr length $nom        # renvoie le nombre de caractères de la variable
expr substr $nom 1 5    # renvoie les 5 premiers caractères de $nom
```

> Pour tout savoir sur expr vous pouvez faire man expr !

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

Variables prédéfinies par le Bash

Le bash et le système Linux d'une manière générale utilise ses propres variables pour fonctionner, par exemple le nom de l'utilisateur courant est stocké dans une variable nommée $USER. Vous pouvez taper echo $user pour voir ce qu'elle contient.

```
# echo $USER
root
```

Les variables système sont nommées en majuscules pour éviter d'entrer en conflit avec les variables que vous pourriez créer.

> Ne nommez donc pas vos variables en majuscules !

Les principales variables prédéfinies sont :

HOME (équivaut à ~)	répertoire de l'utilisateur
PATH	répertoires contenant des fichiers exécutables
MAIL	chemin d'accès aux mails de l'utilisateur
MAILCHECK	temps au bout duquel un mail est traité
IFS	caractère de séparation des arguments
TERM	nom du type de terminal
PS1	invite principale du shell en mode interpréteur
PS2	invite secondaire du shell en mode programmation

> Pour voir toutes les variables d'environnement auxquelles vous pouvez accéder :
> echo $ *(et sans appuyer sur Enter)* <TAB> <TAB>

IV- Saisir une information et la stocker dans une variable

Pour créer un programme qui soit interactif, vous pouvez demander à l'utilisateur de saisir des informations et les utiliser par la suite gràce à des variables. Pour cela, vous pouvez utiliser la commande read suivie du nom de la variable dans laquelle vous voulez stocker l'information :

```
echo Veuillez entrer votre nom :
read nom
echo Vous vous appellez $nom
```

```
# Veuillez entrer votre nom :
# Dupont
# Vous vous appellez Dupont
```

Vous pouvez aussi demander à l'utilisateur d'entrer plusieurs valeurs en une seule fois, par exemple :

```
read nom, prenom
echo Vous vous appellez $prenom $nom
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

V- Structures conditionnelles

Tests avec if :

Pour vérifier qu'une ou plusieurs conditions sont remplies avant d'effectuer un traitement, on utilise souvent le if, par exemple :

```
if [ $age -ge 18 ];
      then echo Vous etes majeur(e)
fi
```

Else, elif

Le if sert à imposer une condition pour effectuer un traitement. Le else (sinon) permet de complêter le if pour effectuer un traitement dans le cas ou la condition n'est pas remplie.
Exemple :

```
if [ $age -ge 18 ];
      then echo Vous etes majeur(e)
      else echo Vous etes mineur(e)
fi
```

Le elif (else if) permet d'imbriquer plusieurs if les uns dans les autres pour pouvoir traiter tous les cas possibles :

```
if [ $feu = «rouge» ];
      then echo N'avancez pas
      elif [ $feu = «orange» ];
            then echo Ralentissez
      else echo Allez-y
fi
```

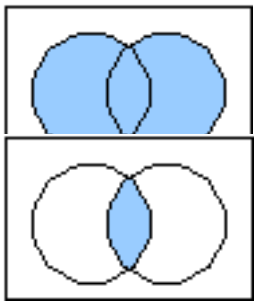
Plusieurs conditions :

Vous pouvez aussi combiner un certain nombre de conditions les unes après les autres, par exemple pour vérifier que l'àge entré par l'utilisateur est situé entre 0 et 100 :

```
if [ $age -le 0 ] || [ $age -ge 100 ];
      then echo l\'age entré n\'est pas correct
fi
```

Pour combiner plusieurs conditions, vous pouvez utiliser soit OU (||), soit ET (&&). Il s'agit des opérateurs logiques :
OU signifie soit l'un, soit l'autre, soit les deux.

ET signifie l'un et l'autre obligatoirement.



Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

Opérateurs de comparaison :

Opérateur	Description	Exemple
Opérateurs sur des fichiers		
-e *fichier*	vrai si *fichier* existe	`[ -e /etc/shadow ]`
-d *fichier*	vrai si *fichier* est un répertoire	`[ -d /tmp/trash ]`
-f *fichier*	vrai si *fichier* est un fichier ordinaire	`[ -f /tmp/glop ]`
-L *fichier*	vrai si *fichier* est un lien symbolique	`[ -L /home ]`
-r *fichier*	vrai si *fichier* est lisible (r)	`[ -r /boot/vmlinuz ]`
-w *fichier*	vrai si *fichier* est modifiable (w)	`[ -w /var/log ]`
-x *fichier*	vrai si *fichier* est exécutable (x)	`[ -x /sbin/halt ]`
fichier1 -nt *fichier2*	vrai si *fichier1* plus récent que *fichier2*	`[ /tmp/foo -nt /tmp/bar ]`
fichier1 -ot *fichier2*	vrai si *fichier1* plus ancien que *fichier2*	`[ /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 ]`

La négation d'une comparaison se fait avec !

Exemple de négation :
Si ce répertoire n'existe pas on le crée :

```
if [ ! -d /repert ];
     then mkdir /repert;
fi
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

Case :

La structure case permet de faire un traitement différent en fonction de la valeur d'une variable, par exemple :

```
#!/bin/bash
echo Quel est votre OS préféré ?
Echo 1- Windows    2- Linux    3- Mac OS    4- Autre
read $choix
case « $choix » in
     1) echo « Vous préférez Windows » ;;
     2) echo « Vous préférez Linux »;;
     3) echo « Vous préférez Mac OS »;;
     4) echo « Vous préférez un autre OS »;;
     else ) echo « Vous devez taper un nombre entre 1 et 4 ! »;;
esac
```

```
# Quel est votre OS préféré ?
2
Vous préférez Linux
```

Select :

Le select est une extension du case. La liste des choix possibles est faite au début et on utilise le choix de l'utilisateur pour effectuer un même traitement.
Exemple :

```
select systeme in "Windows" "Linux" "BSD" "Mac OS" "MS DOS"
do
  echo "Votre système favori est $systeme."
  break
done
```

Ce qui créera automatiquement un menu et donnera à l'écran :

```
1) Windows
2) Linux
3) BSD
4) Mac OS
5) MS DOS
#? 4
Votre système favori est Mac OS.
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

VI- Boucles

Les boucles servent en général à deux choses :
- Vérifier qu'une information saisie par l'utilisateur est correcte et lui faire recommencer la saisie tant que ce n'est pas correct
- Recommencer un certain nombre de fois la même suite de commandes

d'autre part elles fonctionnent toujours avec ces trois critères :
- Une **valeur de départ**
- Une **condition** d'entrée ou de sortie
- Une **incrémentation**

Les boucles While et Until testent une condition **avant** d'effectuer le traitement, ce qui veut dire qu'il se peut qu'on n'entre jamais dans la boucle.

Il n'y a pas d'équivalent de répéter ... jusqu'à ... en langage Shell

La boucle while : tant que …

While signifie tant que, ce qui veut dire que la boucle sera exécutée tant que la condition est respectée.

Exemples :

```
i=1                                 # Valeur de départ 1
while [ $i -le 5 ]; do              # Condition de sortie : i > 5
     echo tour de boucle n° $i
     i = `expr  i  «+»  1`          # Incrément de 1 par tour de boucle
done
```

```
continuer=o                              # Valeur de départ «o»
while [ $continuer = «o» ]; do           # condition de sortie : pas o
     echo Voulez-vous recommencer ? o/n
     read continuer                      # Nouvelle valeur de continuer
done                                     # (qui remplace l'incrément)
```

Cette boucle continue tant que l'utilisateur entre **o**.

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

La boucle until : jusqu'à …

Until signifie jusqu'à, ce qui veut dire que la boucle sera exécutée jusqu'à ce que la condition soit respectée.

Exemples :

```
i=1                                  # Valeur de départ 1
until [ $i -gt 5 ]; do       # Condition de sortie : i > 5
      echo tour de boucle n° $i
      i = `expr  i  «+»  1`        # Incrément de 1 par tour de boucle
done
continuer=o                                 # Valeur de départ «o»
until [ $continuer = «n» ]; do          # condition de sortie : n
      echo Voulez-vous recommencer ? o/n
      read continuer                        # Nouvelle valeur de continuer
done                                        # (qui remplace l'incrément)
```

Cette boucle continue jusqu'à ce que l'utilisateur entre **n**.

La boucle for :

A priori, la boucle for est utilisée quand on veut exécuter un ensemble de commandes un nombre précis de fois.
Exemple :

```
echo Combien voulez-vous d\'étoiles ?
read nombre
for i in `seq $nombre`
do
      echo -n \*
done
```

```
Combien voulez-vous d'étoiles ?
4
****
```

La commande seq sert à afficher une liste de nombre :
seq 3 affichera à l'écran 1 2 3
seq 3 5 affichera à l'écran 3 4 5
seq 1 2 7 affichera à l'écran 1 3 5 7

En shell la boucle for est beaucoup utilisée pour traiter les fichiers, par exemple :

```
echo Ce script va renommer tous les fichiers en y ajoutant votre nom
for fichier in `ls`
do
        if [ ! -d $fichier ]; then
                fichierdest=$USER--$fichier
                mv $fichier $fichierdest
        fi
done
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

VII- Passage de paramètres

En général pour utiliser une commande, on l'appelle par son nom, suivi d'un certain nombre de paramètres, par exemple pour déplacer un fichier :

```
mv      fichier         répertoire/
1       2               3
```

1 : commande
2 : paramètre 1
3 : paramètre 2

L'avantage d'utiliser des paramètres est que lorsqu'on lance le programme, il n'y a pas à faire intervenir l'utilisateur donc le script peut s'exécuter sans que personne ne tape quoi que ce soit au clavier. Pour passer un paramètre à votre programme, il vous faut faire comme avec n'importe quelle commande :

nom_de_votre_script paramètre1 paramètre2

Quand un utilisateur appelle votre programme en y ajoutant des paramètres, vous devez les récupérer dans des variables. Pour cela il existe des variables spéciales :

$1 à $9 : contienne le Xième paramètre passé
$* : contient tous les paramètres qui ont été passés
$# : contient ne nombre de paramètres qui ont été passés
$0 : contient le nom du script lui-même
$$: contient le numéro de processus du shell lui-même

Exemple de script utilisant des paramètres :

```
#!/bin/sh
echo Vous avez passé les $# paramètres suivant :
echo paramètre 1 : $1
echo paramètre 2 : $2
echo paramètre 3 : $3
echo paramètre 4 : $4
echo On peut aussi les afficher tous d\'un coup : $*
```

En appellant ce programme comme ceci, on obtient :

```
# mon_script  10  abc  toto  91

     Vous avez passé les 4 paramètres suivant :
     paramètre 1 : 10
     paramètre 2 : abc
     paramètre 3 : toto
     paramètre 4 : 91
     On peut aussi les afficher tous d'un coup : 10 abc toto 91
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

VIII- Redirections

La sortie standard de la réponse d'une commande est l'écran. Dans certains cas, on a besoin d'utiliser la réponse d'une commande comme paramètre d'une autre commande. Pour cela, il faut utiliser des redirections.

commande1 | commande2

Le pipe permet de renvoyer le résultat d'une commande à une seconde commande la cas le plus fréquent de son utilisation est pour rechercher un texte dans le résultat d'une commande qui nous est fourni.

Par exemple : La commande ps ax liste tous les processus qui tournent. La liste étant longue, on limite l'affichage en passant le résultat de cette commande à grep qui ne va sortir que les lignes contenant le texte apache.

```
# ps  ax  |  grep  apache
     1892 ?        S      0:00 /usr/sbin/apache
     1893 ?        S      0:00 /usr/sbin/apache
```

Commande1 `commande2`

Cette redirection permet de passer la commande de droite comme paramètre de la commande de gauche. C'est donc l'inverse de la redirection |.

Par exemple pour regarder le contenu de tous les fichiers du répertoire courant : On envoie la liste des fichiers (ls) à la commande cat.

```
# cat  `ls`
     contenu des fichiers ...
     ...
```

Redirection vers une autre sortie > :

La sortie par défaut étant l'écran, il se peut que vous vouliez récupérer les sorties pour les écrire soit dans un fichier, soit vers une autre sortie.

D'une manière générale on indique la sortie que l'on veut grâce à > .

Exemple d'écriture d'une sortie dans un fichier :

```
echo Bonjour > /home/toto/bjour # écrit Bonjour dans le fichier bjour
```

Attention, le signe > remplace ce qu'il y avait dans le fichier par la sortie de la commande. S'il s'agit d'un nouveau fichier il le crée.

Pour écrire à la suite d'un fichier existant, il faut doubler le signe : >>

```
echo je suis $USER >> /home/toto/bjour # écrit je suis Toto
                                       # à la suite du fichier bjour
```

A la suite de ces deux commandes, le fichier bjour contient :

```
Bonjour
je suis Toto
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

Redirection des erreurs 2> :

D'autre part, si on veut éviter d'afficher les erreurs sur la sortie standard, on peut les rediriger vers un périphérique inexistant (/dev/null), par exemple :

```
ls -l  2>  /dev/null
```

2> signifie « les messages d'erreur sont redirigés vers ... »

Redirection des entrées < :

A l'inverse, l'entrée standard étant le clavier, vous pouvez récupérer le contenu d'un fichier en entrée. Dans ce cas le signe est inversé : <
Exemple avec le fichier noms.txt qui contient ceci :

```
Dupont
```

```
read $nom < noms.txt
```

La ligne ci-dessus récupère la ligne de noms.txt dans la variable $nom.

IX- Fonctions

Comme pour les autres langages de programmation, les scripts Bash peuvent faire appel à des fonctions.
La déclaration de fonction se fait comme suit :
function nom () {
    commandes ;
}

L'appel d'une fonction se fait en écrivant son nom. On peut lui passer des paramètres comme on le fait pour une commande.

Attention on ne peut appeler une fonction que si elle a été déclarée avant !

Exemple de fonction et de son appel :

```
#!/bin/sh

# Déclaration de la fonction :
function multiplier()
{
 resultat=`expr $1 "*" $2`
 echo  $resultat
}

echo Veuillez entrer deux nombres
read nb1
read nb2

# Appel de la fonction :
multiplier $nb1 $nb2
```

Nicolas Dewaele
nicodewaele@gmail.com
http://nicodewaele.free.fr

X- Sur Internet …

Cours et / ou TP sur la programmation Bash :

http://www-phase.c-strasbourg.fr/inform/linux/cours/user/node1.html

http://www.linux-pour-lesnuls.com/shell.php

http://www.labo-linux.org/cours/module-1/chapitre-11-scripting-bash/