المعهد العالي للإعلاميّة والرّياضيّات بالمنستير
l'Institut Supérieur d'Informatique et de Mathématiques de Monastir

Chapitre 3
programmation shell

Mme Sirine Bchir

Le shell : l'interface en ligne de commande

Le shell = la couche logicielle qui constitue l'interface utilisateur d'un système d'exploitation

Très souvent le mot « shell » fait référence à une interface en ligne de commande.

erface texte : **avantages** et **désavantages**

- L’interface consiste juste en l’affichage de la lecture du texte.
- Peut être utilisé sur des systèmes avec peu de ressources ou par réseau.
- Intégration facile de nouvelles fonctions.
- Automatisation facile.

- Représentation non intuitive d’objets. (tout est texte)
- Aide contextuelle relativement pauvre.

Les shells : quelques titres

sh : Bourne Shell

- ▶ était le shell UNIX par défaut
- ▶ beaucoup de shells modernes sont compatibles

bash : Bourne Again Shell

- ▶ le shell par défaut sous la plupart de distributions Linux et sous MacOS

zsh : Z Shell

- ▶ un shell avec des fonctionnalités avancées cmd : la ligne de commande Windows
- ▶ offre des fonctionnalités pareils aux autres shells, mais est moins utilisé

PowerShell : un shell Windows qui se fonde sur .NET

Script shell = un programme pour un shell

- permet de vérifier des conditions, de faire des boucles et des fonctions
- est exécuté (interprété) directement par le shell (n'est pas compilé)

Pourquoi faire des scripts ?

- encore un langage de programmation ?!
- on pourrait écrire des programmes en Java (ou en un autre, vrai langage)

Les langages de scripts shell sont beaucoup mieux adaptés aux tâches de gestion d'un système.

Les scripts bien faits sont donc plus clairs et plus faciles à maintenir.

Le Hello World

Le fichier sera interprété par /bin/bash

Le shebang →

```
#!/bin/bash
echo Hello
World
```

Afficher Hello World

Pour lancer :

- sauvegarder dans script.sh
- donner les droits à l'exécution : chmod +x script.sh
- lancer : ./script.sh

Les Variables

```
#!/bin/bash
message="Hello World"
echo $message
```

- $ pour utiliser la variable
- toutes les variables sont des chaînes de caractères
- les noms de variable sont sensibles

Variables d'environnement

Variables partagées par tout le système.

- par toutes les instances de shell

La commande env affiche la liste de toutes les variables d'environnement.

Pour définir une variable d'environnement :

```
variable=valeur
export variable
```

ou bien

```
export
variable=valeur
```

La variable d'environnement PATH

Donne la liste de dossiers où se trouvent les fichiers exécutables.

- ls, cp, mv, etc.

Les noms de dossier sont séparés par « : » :

```
echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/bin
```

Calculs numériques

```
variable=$((10 * 20 - 1))
```

ou bien

```
let "variable = 10 * 20 - 1"

echo $variable
```

-4

Seulement les calculs avec les nombres entiers sont possibles :

```
echo $((3/2))
```

1

Structures de contrôle

if : l'instruction conditionnelle

```
x="hello"
if [ $x == "hello" ]
then
    echo True
 else
    echo False
fi
```

Les retours à la ligne sont importants.

On utilise != pour l'inégalité.

Ces opérateurs ne marchent que pour les chaînes de caractères.

if : comparer les nombres

```
if [ $value -eq 1 ]
if [ $value -ne 1 ]

if [ $value -gt 1 ]
if [ $value -ge 1 ]

if [ $value -lt 1 ]  if
[ $value -le 1 ]
```

value = 1 ? (equals)

value ≠ 1 ? (not equals)

value > 1 ? (greater than)

value ≥ 1 ? (greater or equal)

value < 1 (less than)

(less or equal)

- if
- case

vérifier plusieurs alternatives

```
x=4
if [ $x -eq 3 ]
then
   echo Three
elif [ $x -eq 4 ]
then
   echo
Four  else
   echo
   Somethi
```

```
x=3
case $x in   3 )
   echo Three
   ;;
 4 )
   echo Four
   ;;
* )
   echo
   Something
   else
   esac
```

if : vérifier si une variable est définie

```
if [ -z $x ]
then
   echo x
   n\'ést pas
   encore
   défini
fi

x=3

if [ -n $x ]
then
   echo x
```

if : vérifier qu'une condition n'est pas vraie

```
if [ ! -n $x ]
then
   echo x
   n\'ést pas
   encore
   défini
fi

x=3

if [ ! -z $x ]
then
```

if : vérifier qu'une condition n'est pas vraie

```
if [ ! -n "$x" ]
then
   echo x n\'ést
   pas encore
   défini
fi

x=3

if [ ! -z "$x" ]
then
   echo x est
   défini
   maintenant
```

if : connecteurs logiques

```
x=4
y=5

if [ "$x" -eq 4 -a "$y" -eq 5 ]
then
  echo '$x est 4 et $y est 5'
fi

if [ "$x" -eq 5 -o "$y" -eq 5 ]
then
  echo '$x est 5 ou $y est 5'
fi
```

```
x=4
y=5

if [[ "$x" -eq 4 ]] && [[ "$y" -eq 5 ]]
then
  echo '$x est 4 et $y est 5'
fi

if [[ "$x" -eq 5 ]] || [[ "$y" -eq 5 ]]
then
  echo '$x est 5 ou $y est 5'
fi
```

for : traverser une liste

Traiter chaque ligne de la sortie d'une commande :

```
for i in $(ls)
do
   echo "J'ai vu $i !"
done
```

Faire une boucle for à la Java :

```
for i in $(seq 1 10)
do
   echo "Et
$i !"  done
```

Même chose que for(i =

while : boucle while

```
i=0
while [[ $i -lt 10 ]]
do
   echo "et $i"
   i=$(($i + 1))
done
```

La syntaxe pour les conditions est la même que pour if.

until : boucle while avec la condition inverse

```
i=0
until [[ $i -ge 10 ]]
do
   echo "et $i"
   i=$(($i + 1))
done
```

while : lire un fichier ligne par ligne

La commande read renvoie les lignes de l'entrée standard une par une.

```
while read line  do
   echo $line
done
```

Tableaux

Tableaux : déclaration et indexation

Créer un tableau en utilisant la syntaxe liste :

```
tableau=( "salade" "tomate" "oignon"
)
```

Accéder aux éléments d'après leurs indexes :

```
echo ${tableau[1]}
```

Donner les valeurs aux éléments directement :

```
tableau[0]="salad
e"
tableau[1]="tomat
```

Tableaux : longueur et traversée

Pour savoir la longueur d'un tableau :

```
echo ${#tableau[@]}
```

ou bien

```
echo ${#tableau[*]}
```

Pour traverser un tableau :

```
for i in ${tableau[@]}
```

ou bien

```
for i in ${tableau[*]}
```

Les fonctions

Fonctions : syntaxe de base

Pour déclarer une fonction :

```
function helloWorld
  {  echo Hello
  World
}
```

Pour appeler une fonction :

```
helloWorld
```

Où sont les paramètres ?

Fonctions : les paramètres

On accède aux paramètres par leur numéro :

```
function premier {
   echo "premier paramètre = $1"
}
```

La commande shift oublie le premier argument et décale tous les numéros des autres arguments.

```
function premier {  echo
   "premier = $1"  shift
   echo "deuxième = $1"
}
```

Fonctions : accéder à tous les paramètres

$@ le tableau de tous les paramètres

$* une chaîne de caractères qui contient tous les paramètres (aplatis)

$# le nombre de paramètres

Pour énumérer tous les paramètres :

```
function tous {  for
   i in "$@"  do
      echo $i
   done
}
```

Fonctions : variables locales et globales

Par défaut, toutes les variables sont globales.

Pour déclarer une variable locale, utiliser local.

```
x=1
y="hello"

function modif
   {  x=2
   local y="bonjour"
}

modif
   echo $x          # affiche 2
   echo $y          # affiche "hello"
```

Fonctions : valeurs de retour

Les fonctions n'ont pas de valeur de retour, mais elles ont une sortie standard.

```
function func {
   echo "hello"
}
x=$(func)
echo $x
```

On peut aussi communiquer via les variables globales.

Fonctions : codes de retour

Une fonction peut renvoyer un code de retour.

```
Function plusgrand{if [[ $1 -gt $2 ]]
  then
      return 1          # code de retour
  else
      return 0
  fi }
```

La variable spéciale $? contient le code de retour du dernier appel.

Un code de retour est toujours une valeur numérique.

Fonctions = scripts

(presque)

> Une fonction se comporte comme un script.

Les deux ont une entrée standard et une sortie standard. Les façons d'accéder aux paramètres sont les mêmes.

Pour sortir d'un script, utiliser `exit`

- peut prendre en argument le code de retour.

Exercice 1
écrire un script qui lit un entier à partir du clavier puis affiche son signe (négatif, positif ou nul)

```
#!/bin/bash
echo "saisir un entier"
read n
if [ $n -gt 0 ];then
echo "$n est positif"
elif [ $n -lt 0 ];then
echo "$n est négatif" else
echo "$n est nul"
fi
```

Tests sur les valeurs numériques

Option	Signification
-lt	nb1 inférieur à nb2 (less than)
-le	nb1 inférieur ou égal à nb2 (less or equal)
-**gt**	nb1 supérieur à nb2 (greater than)
-ge	nb1 supérieur ou égal à nb2 (greater or equal)

Exercice 2:
écrire un script qui lit 2 chaînes de caractères comme paramètres puis compare leurs valeurs et affiche le résultat sous forme d'un message

```
#!/bin/bash
if [ $1 = $2 ];then
echo "$1 et $2 sont identiques" else
echo "$1 et $2 sont différentes"
fi
```

Commande read

La commande **read** lit une ligne à partir d'une entrée standard et affecte les valeurs de chaque zone de la ligne d'entrée à une variable shell en utilisant les caractères de la variable IFS (Internal Field Separator) comme séparateurs.

Article	Descriptif
-P	Lit l'entrée à partir de la sortie d'un processus exécuté par le shell à l'aide de | & (barre verticale, perluète).
-r	Indique que la commande de lecture traite un caractère \ (barre oblique inversée) comme faisant partie de la ligne d'entrée et non comme un caractère de contrôle.
-S	Sauvegarde l'entrée en tant que commande dans le fichier d'historique du shell.

Exercice 3 :

Écrire un script qui affiche le menu suivant :
1 – Windows?
2 – BeOs?
3 – Linux?
4 – Unix?
Réponse ?
Si vous répondez 1 alors le programme affiche « Dommage! », 2 il affiche « Peut mieux faire! », 3 « Pas mal! », 4 « Super! ».

```
#!/bin/bash
echo "1-Windows"
echo "2-BeOs"
echo "3-Linux"
echo "4-Unix"
read -p "réponse?" rep *case $rep in
1)echo "Dommage!";;
2)echo "peut mieux faire";
3)echo "pas mal";;
4)echo "super";; *)
echo "choix invalid";; esac
//Source : www.exelib.net
```

Exercice 4 :Écrire un script qui dit si le premier paramètre passé en ligne de commande est un fichier, un répertoire, ou autre type.

```
/#!/bin/bash
if [ -f $1 ];then
echo "$1 est un fichier" elif [ -d $1 ];then
echo "$1 est un dossier" else
echo "$1 est un autre type de fichier"
fi
```

Exercice 5 : Écrire un script permettant de lister uniquement les répertoires se trouvant dans un emplacement donné comme premier par

```
#!/bin/bash
cd $1
for i in *;do
if [ -d $i ];then
ls -ld $i
fi
done
```

Exercice 6 :

Écrire une commande qui prend en argument un nom de fichier et affiche:

- son nom si c'est un fichier régulier non exécutable suivi de la mention « est un fichier non exécutable »;
- son nom si c'est un fichier régulier exécutable suivi de la mention « est un fichier exécutable »,
- la liste de tous les fichiers réguliers exécutables qu'il contient si c'est un répertoire.

```
#!/bin/bash
if [ -f $1 ]&& ! [ -x $1 ];then
echo "$1 est un fichier non
executable" elif [ -f $1 ]&&[ -x
$1 ];then
echo "$1 est un fichier executable"
elif [ -d $1 ];then
cd $1 for i in *;do
if [ -f $i ]&&[ -x $i ];then
ls -l $i
fi
done
fi
```

Exercice 7 :

Écrire une commande recycle qui permet de manipuler une corbeille de fichiers (un répertoire)nommée corbeille et située à votre répertoire personnel.

La commande accepte trois options :

- recycle -l pour lister le contenu de la corbeille;
- recycle -r pour vider la corbeille ;
- recycle fichier1 fichier2 ... pour déplacer les fichiers considérés vers la corbeille.

Si la corbeille n'existe pas, elle est créée

```
#!/bin/bash
if ! [ -e "Corbeille" ];then
mkdir Corbeille
fi
if [ -z $1 ];then
echo "usage:"
echo "recycle -l:lister la corbeille" echo
"recycle -r:vider la corbeille" echo "recycle
fichiers:déplacer les fichiers vers la
corbeille"
elif [ $1 = "-l" ];then ls -l Corbeille elif
[ $1 = "-r" ];then
rm -r Corbeille/* else
mv $* Corbeille
fi
```

Exercice 8 :

1. Écrire un script qui concatène puis trie deux fichiers file1 et file2 dans un nouveau fichier file3 et qui affiche le nombre total de lignes. Les noms des trois fichiers doivent être passés en paramètre.
2. Modifier le script précédent pour demander à l'utilisateur de saisir au clavier le (ou les) nom(s) de fichiers qu'il aurait oublié d'indiquer en lançant le script

```
#!/bin/bash
if [ $# -lt 3 ];then
echo "usage:concat file1 file2 file3"
else echo "concaténation de $1 et $2
dans $3 ..." cat $1 $2>$3 echo "tri de
$3 ..." sort $3 echo "le nombre de
lignes que contient $3 est $(cat $3 |wc
-l)" fi
```

```
if [ $# -lt 3 ];then
echo "usage:concat file1 file2 file3"
echo "saisir 3 fichiers... "
read f1 f2 f3
echo "concaténation de $f1 et $f2 dans $f3 ..."
cat $f1 $f2>$f3
echo "tri de $f3 ..."
sort $f3
echo "le nombre de lignes que contient $f3 est $(cat $f3
|wc -l)"
else
echo "concaténation de $1 et $2 dans $3 ..."
cat $1 $2>$3
echo "tri de $3 ..."
sort $3
echo "le nombre de lignes que contient $3 est $(cat $3 |
wc -l)"
fi
```

Exercice 8 :

Écrire un script calculatrice permettant d’afficher un menu :
calculer la somme
calculer la multiplication
calculer la soustraction
calculer la division

```
#!/bin/bash
#définition des fonctions
function somme
{
let s=$1+$2
return $s
}
function multiplication
{
let p=$1*$2
return $p
}
```

```
function soustraction
{
let diff=$1-$2
return $diff
}
function division
{
if [ $2 -eq 0 ];then
return 1
else
let div=$1/$2
return $div
fi
}
```

```
#script principal
read -p "saisir deux entiers:" a b
echo "a-calculer la somme"
echo "b-calculer la multiplication"
echo "c-calculer la soustraction"
echo "d-calculer la division"
read -p "faites votre choix?" c
case $c in
a)somme $a $b
echo "la somme est $?";;
b)multiplacation $a $b
echo "le produit est $?";;
c)soustraction $a $b
echo "la soustraction est $?";;
d)division $a $b
if [ $? -eq 1 -a $b -eq 0 ];then
echo "erreur:division par 0"
else
echo "la division est $?"
fi;;
*)echo "choix invalid";;
esac
```