CHAPITRE 3 :

LA PROGRAMMATION SHELL

Objectifs spécifiques

- Se familiariser avec l'écriture de Script Shell
- Résolution de problèmes aves des scripts shell

Eléments de contenu

I. Introduction

II. Initiation d'un shell

III. Les variables

IV. Les scripts shell

Volume Horaire :

Cours : 3 heures

TD : 1 heure 30 mn

3.1 Introduction

Le shell est l'interface Homme/Machine des systèmes UNIX/LINUX. Il en existe plusieurs qui diffèrent par la syntaxe mais aussi par les services offerts. Le shell a un double rôle, celui d'interpréteur de commandes mais aussi celui d'un langage de programmation et à ce titre, il gère des variables et des fonctions et dispose de structures de contrôle (boucles, conditionnelles ….). Ces fonctionnalités servent essentiellement pour l'écriture de procédures (scripts shell) qui permettent à chacun de paramétrer une session et développer ses propres outils.

3.2 Initiation d'un shell

Lors de l'ouverture d'une session, le shell exécute des fichiers de configuration, qui peuvent contenir des commandes quelconques et sont généralement utilisées pour définir des variables d'environnement et des alias.

- csh exécute le fichier ~/.cshrc
- tcsh exécute le fichier ~/.cshrc
- sh exécute le fichier ~/.profile
- bash exécute le fichier ~/.bash_profile ou à défaut le fichier ~/.profile
- On remarque bien que ces fichiers de configuration sont des fichiers cachés.

Remarque : Chaque utilisateur peut ajouter des commandes shell au profil personnel ~/.bash_profile.

On peut par exemple ajouter les lignes suivantes :

```
gedit .bash_profile
clear
salut= ‘’bonjour $USER ! ‘’
# ceci est un commentaire, $USER contient le nom de connexion
echo $USER
echo ‘’Nous sommes le $(date) ‘’
# $(nomCommande) permet d'obtenir le résultat de l'exécution de la commande mentionnée entre ( )
```

3.3 Les scripts Shell

Un script bash ou shell est un simple fichier texte exécutable (avec un droit d'exécution x) commençant par les caractères #!/bin/bash (doivent être les premiers caractères du fichier) et qui peut contenir des commandes ou des instructions. En effet, on peut avoir la dans du shell, des variables, des structures de contrôles, des structures répétitives…etc d'où l'appellation script shell

Exemple

```
#!/bin/bash
…
echo “voici un script bash“
…
```

De même que tout autre programme, Shell peut également prendre des arguments dans un fichier. Les scripts (programmes Shell) peuvent avoir des arguments.

3.3.1 Commentaire

Le symbole # sert à indiquer une ligne commentaire dans les fichiers de commandes.

```
# cette commande permet ……
# Auteur :
```

3.3.2 Commande echo

La commande **echo** permet d'afficher l'expression donnée en paramètre. Cette dernière peut être :

- Soit une variable
- Soit une chaîne
- Soit une expression composée de chaines et de variables.

3.3.3 Commande read

La commande **read** lit la saisie de l'utilisateur à partir du canal d'entrée standard (clavier) et stocke ces données dans des variables du shell. Read lit une ligne entrée, la découpe en mots séparés par des espaces et affecte chaque mot aux variables de la liste-de-variables Les noms de ces variables sont transmis comme paramètres de read dont la syntaxe est la suivante : **read var1 [var2 …]**. Si la ligne lue comporte plus de mots que la liste-de-variables comporte, il affecte à la dernière variable l'ensemble des mots restant à affecter.
Lorsque read est traité, le shell attend une entrée de la part de l'utilisateur.

Exemple:

```
echo "saisir le contenu des variables a et b "
read a b
echo "a = $a"
echo "b = $b"
```

3.3.4 Les variables

3.3.4.1 Les noms de variables

Les noms des variables peuvent être composés:

- soit d'une suite de lettres, de chiffres et du caractère _ : ce cas correspond aux variables créées par l'utilisateur
- soit d'un chiffre : ce cas correspond aux paramètres des fichiers de commandes
- soit de l'un des caractères quelconques * @ # ? - $! ce cas correspond à un ensemble de variables gérées par le Shell

3.3.4.2 Variables d'environnement (prédéfinies)

Le système UNIX définit pour chaque processus une liste de variables d'environnement, qui permettent de définir certains paramètres. Parmi ces variables nous citons :

- **HOME** : contient le chemin absolu du répertoire de connexion de l'utilisateur
- **LOGNAME** : contient le nom de connexion de l'utilisateur
- **PATH** : contient la liste des répertoires contenant des exécutables séparés par ' :'. Ces répertoires seront parcourus par ordre à la recherche d'une commande externe
- **SHELL** : contient le chemin d'accès absolu des fichiers programmes du shell

Remarque :

La commande **env** permet d'afficher l'ensemble des variables d'environnement pour le shell actif.

Exemple :

```
$ echo "le répertoire de connexion est : $HOME"
```

```
Le répertoire de connexion est : /home/imene
```

3.3.4.3 Les variables utilisateur

3.3.4.3.1 Declaration

Il n'est pas nécessaire de déclarer une variable avant de l'utiliser. Les objets possédés par les variables sont d'un seul type des chaînes de caractères.

3.3.4.3.2 Référencement du contenu d'une variable

Pour référencer la valeur d'une variable, on utilise la notation consistant à écrire le signe $ suivi du nom de la variable.

```
i=2

echo $i

2
```

3.3.4.3.3 Affectation de valeurs aux variables

Pour affecter une valeur à une variable, il suffit de taper le nom de variable, suivi du signe égal, suivi de la valeur que l'on désire affecter à la variable. La syntaxe d'une affectation est la suivante :

```
nom-de-variable = chaîne-de-caractères
```

Exemple:

```
Var1=bonjour
Var2=madame
REP=/usr/monDossier
PERSONNE=$USER
```

Il est possible de stocker le résultat d'une commande dans une variable. Le résultat c'est la chaîne affichée par la commande, et non son code de retour (qui peut etre 0 ou 1). On utilise soit :

- les back quotes (anti-apostrophes qu'on obtient en tapant Altgr+7) ou
- les parenthèses

Exemple :

```
DATE=`date`
NousSommesLe=$(date)
```

La commande peut être compliquée, par exemple avec un tube :

```
Fichiers=$(ls /usr/include/*.h | grep std)
#on envoie le flux de sortie de la commande ls comme flux d'entrée pour la commande grep
```

```
echo $Fichiers
/usr/include/stdint.h /usr/include/stdio_ext.h /usr/include/stdio.h
/usr/include/stdlib.h /usr/include/unistd.h
```

On peut affecter une chaîne vide à une variable de 3 manières différentes:

```
V1=
V2=' '
V3=""
```

Si la chaîne-de-caractères à affecter à la variable comporte des blancs, il faut l'entourer à l'aide des quotes et il ne faut surtout pas laisser des espaces autour du signe =.

```
MESSAGE='hello everybody'
```

3.3.4.4 Les variables spéciales $

$? : référence la valeur retournée par la dernière commande exécutée.

$$: référence le numéro du processus (PID) du shell actif

$! : référence le numéro du processus (PID) du dernier processus lancé en arrière plan

$0 : référence le nom de procédure de commande

$1….$9 : référence la valeur de nième paramètre

$# : référence le nombre de paramètres d'un fichier de commandes.

$* : référence la liste de tous les paramètres $1……$9, est utilisé si l'on veut substituer tous les arguments sauf le nom de commande ($0).

$@ : référence la liste des paramètres sous la forme "$1", "$2",…., "$9"

Remarque :

La commande **set** sans paramètre permet d'afficher la liste des variables définies dans le shell. Elle ne montre pas seulement les variables définies par l'utilisateur mais aussi les variables systèmes. Elle visualise la variable ainsi que son contenu.

Exemple 1

On suppose que l'utilisateur toto n'est pas défini dans le système :

```
grep  toto /etc/passwd
echo $?
1
grep imene /etc/passwd
```

```
imene:x:77:227:utilisateur imene:/users/ imene:/bin/bash
echo $?
0
```

Si le code retour de la commande **est nul**, ça signifie que la commande ou le programme a été bien exécuté sans erreur. Si le code retour est **non nul**, ceci signifie que la commande a rencontré des erreurs lors de son exécution.

Exemple 2

```
#!/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:" $*

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

3.3.5 Commande shift

La commande interne **shift** a pour effet d'effectuer un décalage de paramètres qui permet d'affecter la valeur de **$2** à **$1**, la valeur de **$3** à **$2**, etc... Et d'affecter la valeur du dixième argument qui est inaccessible à **$9**. La commande shift met à jour la variable **$#** en diminuant sa valeur de 1. La commande shift permet donc d'accéder aux paramètres qui sont au-delà du neuvième paramètre.

3.3.6 Les tableaux

Il existe deux types de tableaux : les tableaux indicés et les tableaux associatifs.

3.3.6.1 Création de tableaux

Un tableau peut être créé de façon explicite à l'aide du mot-clef ***declare (***et peut aussi être initialisé en même temps***)***, suivi de l'option -a pour un tableau indicé, et -A pour un tableau associatif

```
declare -a tableauIndice=( "un" "deux" "trois" "quatre" )

declare -A tableauAssociatif=( ['un']="one" ['deux']="two" ['trois']="three" )
```

La création d'un tableau indicé peut se faire simplement par initialisation, en fournissant ses éléments entre parenthèses :

```
tableauIndice=( "un" "deux" "trois" "quatre" )
```

Les indices sont ici assignés automatiquement, en commençant par 0. On peut aussi l'initialiser avec des indices imposés :

```
tableauIndice =( "toto" "titi" [42]="tata" "tutu" )
```

Dans cet exemple, l'indice 0 vaut toto, le 1 titi, le 42 tata et le 43 tutu (les valeurs des autres indices sont la chaîne vide).

Les tableaux associatifs peuvent également être créés de la même manière, mais une clef doit bien sûr être précisée pour chaque élément :

```
tableauAsso=( ['un']="one" ['deux']="two" ['trois']="three" )
```

3.3.6.2 Affichage d'un tableau

L'affichage d'un tableau se fait avec la syntaxe **${montableau[*]}** , ou **${montableau[@]}**.

La différence entre l'utilisation de @ et * est identique à celle entre les variables spéciales $@ et $* du shell.

```
declare -a tableauIndice=( "un" "deux" "trois" "quatre" )

echo ${ tableauIndice[@]}

un deux trois quatre

declare -A tableauAsso=( ['un']="one" ['deux']="two" ['trois']="three" )

echo ${tableau_asso[@]}

one two three
```

L'utilisation des accolades est nécessaire pour éviter les conflits avec les développements de chemin (dans lesquels les crochets ont une signification spéciale).

Il est possible d'obtenir la liste des clefs d'un tableau à l'aide de la syntaxe **${!tableau[@]}**. Dans le cas d'un tableau indicé, on obtient ses indices

```
declare -a tableauIndi=( "un" "deux" "trois" "quatre" )
```

```
echo ${!tableauIndi[@]}

0 1 2 3

declare -A tableauAsso=( ['un']="one" ['deux']="two" ['trois']="three" )

echo ${!tableau_asso[@]}

un deux trois
```

Ceci permet de vérifier les indices assignés aux valeurs lors de l'initialisation du tableau :

```
declare -a tableau_indi( "fry" "leela" [42]="bender" "flexo" )

echo ${!tableau_indi[@]}

0 1 42 43
```

Quant à la taille du tableau on peut l'obtenir avec la syntaxe **${#tableau[@]}** :

```
declare -a tableau_indi=( "un" "deux" "trois" "quatre" )

echo ${#tableau_indi[@]}

declare -A tableau_asso=( ['un']="one" ['deux']="two" ['trois']="three" ) echo ${#tableau_asso[@]}

3
```

3.3.6.3 Lecture d'un élément d'un tableau

La lecture d'un élément se fait selon la syntaxe **tableau[indice]**, mais nécessite toujours les accolades :

```
montableau=("un" "deux" "trois" "quatre")
echo ${montableau[2]}
trois
declare -A tabAsso=( ['un']="one" ['deux']="two" ['trois']="three" )
echo ${tabAsso["deux"]}
two
```

Pour les tableaux indicés, un indice négatif correspond à un indice commençant à l'indice maximal du tableau plus un (le dernier élément est donc *-1*). Si aucun indice n'est fourni pour le tableau, on accède à l'indice 0.

3.3.6.4 Modification d'un élément

Un élément peut être assigné selon la syntaxe **tableau[indice]=valeur**. Si le tableau n'existe pas, il sera créé comme un tableau indicé :

```
tab[1]="uno"

echo ${tab[1]}
```

```
uno
```

Il n'est pas possible de créer un tableau associatif en lui assignant un élément, il faut le déclarer explicitement avant l'assignation (ou l'initialiser en lui fournissant les clefs entre crochets).

```
declare -A tabAsso
tabAsso["couleur"]="jaune"
echo ${tab_asso["couleur"]}
jaune
```

Pour supprimer un élément d'un tableau on utilise la commande interne ***unset tableau[indice]*** :

```
declare -a tableauIndi=( "un" "deux" "trois" "quatre" )
echo ${tableauIndi[@]}
un deux trois quatre
unset tableauIndi[1]
echo ${tableauIndi[@]}
un trois quatre
declare -A tableauAsso=( ['un']="one" ['deux']="two" ['trois']="three" )
echo ${tableauAsso[@]}
one two three
unset tableauAsso['deux']
echo ${tableauAsso[@]}
one three
```

La commande unset permet aussi de détruire tout un tableau, en lui passant son nom, nom[*] ou nom[@]. Ainsi, en supposant un tableau de nom *tableau*, les trois lignes suivantes sont équivalentes :

- unset tableau
- unset tableau[@]
- unset tableau[*]

3.3.7 Commande exit

La commande interne **exit** permet de terminer un Shell en transmettant un code de retour. Si exit est invoqué avec un paramètre, c'est ce paramètre qui est pris comme code de retour. Sinon, le code de retour sera le code de retour de la dernière commande exécutée.

3.3.8 Commande expr

Cette commande évalue les arguments comme une expression. Le résultat est envoyé sur la sortie standard. *arguments* est une expression comprenant des opérateurs.

• **exp1 \ | exp2**: retourne exp1 si elle est non nulle et non vide, sinon retourne exp2 si exp2 est non nulle et non vide, sinon retourne 0.

• **exp1 \ & exp2**: retourne exp1 si aucune des expressions n'est vide ou nulle, 0 sinon.

• **exp1 op_comp exp2**: où **op_comp** est un opérateur de comparaison (<, <=, =, !=, >=,>); retourne 0 si vrai, 1 sinon.

• **exp1 + exp2**: addition.

• **exp1 - exp2**: soustraction.

• **exp1 * exp2** : multiplication.

• **exp1 / exp2**: division.

• **exp1 \% exp2**: modulo.

• **chaîne**: renvoie chaîne sur la sortie standard si la chaîne est non vide, 0 sinon.

Exemple :

```
a=1
a=‘expr $a + 1‘
echo $a
```

3.3.9 Commande let et (( ))

Les commandes let et les (( )) sont équivalentes, elles servent à :

- Affecter des variables numériques.

```
let x=1

(( x=4 ))
```

- Faire des calculs

```
let x=x+1

(( i +=1 ))
```

- Evaluer les tests numériques

```
(( (x=x-1) ‹ 0 ))
```

3.3.10 Les structures conditionnelles

3.3.10.1 Commande test

La commande test permet d'exprimer des prédicats sur les chaînes de caractères, sur les entiers et sur les fichiers. La valeur de ces prédicats peut ensuite être testée dans une structure **if**, **while** ou **until**.

La syntaxe de la commande :

test prédicat ou **test [prédicat]**

Remarque :

Il ne faut pas oublier les espaces de part et d'autres des crochets.

- **Prédicat sur les chaines de caractères**

on note ch1 et ch2 deux chaînes de caractères.

test ch1 = ch2 vrai si ch1 est égale à ch2.

test ch1 != ch2 vrai si ch1 est différente de ch2.

test -n ch1 vrai si ch1 est non vide

test -z ch1 vrai si ch1 est vide

- **Prédicat sur les entiers**

on note n1 et n2 deux entiers décimaux.

test n1 -eq n2 vrai si n1 est égal à n2

test n1 -ne n2 vrai si n1 est diffèrent de n2

test n1 -gt n2 vrai si n1 est strictement supérieur à n2

test n1 -ge n2 vrai si n1 est supérieur ou égal à n2

test n1 -lt n2 vrai si n1 est strictement inférieur à n2

test n1 -le n2 vrai si n1 est inférieur ou égal à n2

- **Prédicat sur les fichiers**

test -r fichier vrai si fichier existe et on a le droit read

test -w fichier vrai si fichier existe et on a le droit write

test -x fichier vrai si fichier existe et on a le droit eXecute

test -e fichier vrai si le fichier existe

test -f fichier vrai si fichier existe et c'est un fichier ordinaire

test -d fichier vrai si fichier existe et c'est un répertoire

test -L fichier vrai si fichier existe et c'est un lien symbolique

test -b fichier vrai si fichier existe et c'est fichier spécial en mode bloc

test -c fichier vrai si fichier existe et c'est un fichier spécial en mode caractère

test -s fichier vrai si fichier existe et a une taille non nulle

fichier1 -nt fichier2 (newer than) vrai si fichier1 a été modifié plus récemment que fichier2.

fichier1 -ot fichier2 (older than) vrai si fichier1 a été modifié moins récemment que fichier2.

- **Prédicat sur logiques**

! (Négation logique)

-a (ET logique)

-o (OU logique)

3.3.10.2 Structure if

Syntaxe

if commande
then liste_commandes1
[**else** liste_commandes2]
fi

La commande est évaluée. Si elle est vraie (code de retour nul), liste_commandes1 sont exécutées (et liste_commandes2 ne le sont pas) et si elle est fausse (code de retour non nul), liste_commandes2 sont exécutées (liste_commandes2 ne le sont pas). La partie **else** de cette instruction est optionnelle.
Il existe une variante pratique de if:

if commande1
then ligne_commandes1
elif commande2
then ligne_commandes2
elif commande3
then ligne_commandes3
else
ligne_commandes4
fi

Exemples

```
if test −d $1
then echo "$1 est un répertoire"
elif test −f $1
then echo "$1 est un fichier"
```

```
elif test −L $1
then echo "$1 est un lien symbolique"
else echo "$1 autre ..."
fi

if (test $langue = "francais")
then echo "bonjour"
else echo "hello"
fi
```

3.3.10.3 Structure case

Cette structure de contrôle permet d'effectuer un branchement conditionnel sur une séquence de commandes en fonction de la valeur d'une variable. La syntaxe est la suivante :

```
case mot in

        liste-de-modèles ) liste-de-commandes ;;
        liste-de-modèles ) liste-de-commandes ;;
esac
```

Dans les listes-de-modèles les modèles sont sépares par le caractère **|**.

La sémantique des modèles

Les modèles sont les mêmes que ceux qui servent de modèles de noms de fichiers. Les méta caractères sont * ? [ ] avec la même sémantique. Un mot est dit conforme à une liste-de-modèles si il est conforme à l'un quelconque des modèles de la liste (le caractère | a donc le sens d'un "ou").

La sémantique du case

Le mot est comparé successivement à chaque liste-de-modèles. A la première liste-de-modèles pour laquelle le mot correspond, on exécute la liste-de-commandes correspondante.

Si mot ne correspond à aucun modèle, l'instruction se poursuit jusqu'à la condition *) et si celle-ci n'existe pas, jusqu'au esac.

Exemples

```
case $langue in
        francais) echo "Bonjour" ;;
        anglais ) echo "Hello" ;;
        italien ) echo "Bongiorno" ;;
esac
```

```
case $param in
        0|1|2|3|4|5|6|7|8|9 ) echo "$param est un chiffre";;
        [0-9]*              ) echo "$param est un nombre";;
        [a-zA-Z]*           ) echo "$param est un nom";;
        *                   ) echo "$param de type non défini";;
esac

case $1 in
        one)X=un;;
        two)X=deux;;
        three) X=trois;;
        *) X=$1;;
 esac

 echo "voulez vous continuer :"
read reponse
case "$reponse" in
        [nN] | no | NO) echo "Quitter";;
        [yY] | yes | YES) echo "Continuer";;
        *) echo "Réponse non valide";;
esac
```

3.3.11 Structures itératives

3.3.11.1 Boucle for

Pour la boucle for, il ne s'agit pas de fixer une valeur de départ et une valeur de fin contrôlant le nombre d'itérations mais d'une répétition d'un traitement pour des valeurs différentes d'une variable.
La syntaxe est la suivante :

```
for nom in liste_mots
do
        liste_commandes
done
```

A chaque itération de la boucle for, la variable Shell « nom » prend comme valeur le mot suivant de la liste_mots. Si la liste_mots est omise, la boucle est exécutée pour chacun des paramètres positionnels (des arguments donnés à la commande). Ainsi :

for i est équivalent à dire **for i in $***

Exemples

Cette séquence Shell recopie tous les fichiers *.c du répertoire courant dans le répertoire *rep*.

```
for i in *.c
do
    cp $i rep/
done
```

Cet exemple permet d'afficher le type de chaque fichier dans l'ensemble

```
for fichier in essai1 essai2 essai3
do
        file $fichier
done
```

La liste de mots à parcourir peut être aussi le résultat d'une commande. Cet exemple permet de parcourir le fichier /etc/passwd ligne par ligne et affiche le nom d'utilisateur correspond à chaque ligne.

```
for ligne in $(cat /etc/passwd)
do
    USER=$(echo $ligne|cut –f1 –d ":")
    echo $USER
done
```

3.3.11.2 Boucle while

Cette structure permet de boucler sur une séquence de commandes tant que la condition est vraie. La boucle est interrompue si la valeur de retour est différente de zéro.

La syntaxe est la suivante:

```
while commande
 do
        liste-de-commandes
 done
```

La structure **while** itère:

- ⇨ Exécution de la commande du while
- ⇨ Si celle ci rend un code de retour nul (Vrai) il y a exécution de la liste-de-commandes du do, sinon la boucle se termine.

De la même manière que pour la structure if, la liste-de-commandes de la partie while est généralement réduite à une seule commande.

Exemples

- ✈ L'exemple ci-dessous réalise une boucle de 10 itérations et incrémente la variable i à chaque instruction.

```
i=1
while (test $i  -le  10)
do
        i=`expr $i + 1`
done
```

- ✈ Tant que l'utilisateur passé en argument est dans la liste de la commande who (donc parmi les utilisateurs connectés), on boucle après avoir attendu 60 secondes. S'il n'est plus connecté, la commande du while va retourner un code retour non nul (Faux), on signale qu'il n'est plus connecté.

```
while who | grep "$1"
do
        sleep 60
done
echo "\n$1 n'est plus connecté"
```

3.3.11.3 Boucle until

La boucle est exécutée jusqu'à ce que la dernière commande de la suite_commande renvoie la valeur 0, à ce moment la boucle est interrompue. La syntaxe est la suivante:

```
until commande
do
        liste-de-commandes
done
```

La structure until itère l'exécution de la commande du until, si celle ci rend un code de retour non nul (Faux) il y a exécution de la liste-de-commandes du do, sinon la boucle se termine quand l'exécution de la commande du until rend un code retour nul (Vrai).

Exemples

```
until who | grep "$1"
do
```

```
        sleep 180
done
echo "\n$1 est arrivé"
```

Jusqu'à ce que l'utilisateur passé en argument se trouve dans la liste des utilisateurs (who | grep $1), on fait une pause (sleep 180). Lorsque l'on sort de la boucle (alors l'utilisateur est connecté), on affiche le message "utilisateur est arrivé".

3.3.11.4 Débranhements break et continue

Il est possible de sortir prématurément d'une boucle sans effectuer toutes les itérations prévues, par le mot clé **break**. Cela permet de pouvoir sortir d'une boucle infinie, ou que la sortie soit déclenchée par un événement autre que le test de boucle.

Le mot **continue** permet au contraire de faire une itération supplémentaire.

Exemple:

Ce script boucle en demandant un nom de fichier jusqu'à ce que la réponse désigne un fichier existant.

```
#!/bin/bash
while
do
        echo "nom de fichier: \n"
        read fic
        if test −f "$fic"
        then
                echo $fic fichier trouvé
                break
        else
                echo $fic fichier inconnu
        fi
done
echo "fin de traitement"
```

L'exécution de ce script permet les résultats suivants :

```
$. /script
nom de fichier : toto
toto fichier inconnu
nom de fichier : fich1
fich1 fichier inconnu
nom de fichier : monfichier
monfichier fichier existant
fin de traitement
```

Exercice 1 :

Créer un script qui demande à l'utilisateur de saisir une note et qui affiche un message en fonction de cette note :

- "très bien" si la note est entre 16 et 20 ;
- "bien" lorsqu'elle est entre 14 et 16 ;
- "assez bien" si la note est entre 12 et 14 ;
- "moyen" si la note est entre 10 et 12 ;
- "insuffisant" si la note est inférieur à 10.

Faites en sorte que le programme se répète tant que l'utilisateur n'a pas saisi une note négative ou 'q' (pour quitter).

```
#!/bin/bash

note=0
moyenne=0
i=0

until [ "$note" -lt 0 ]; do
   echo "Entrez votre note (q pour quitter) :"
   read -r note
   if [ "$note" = "q" ]; then
      note=-1
      echo "au revoir !"
   elif [ "$note" -ge 16 ]; then
      echo "très bien"
   elif [ "$note" -ge 14 ]; then
      echo "bien"
   elif [ "$note" -ge 12 ]; then
      echo "assez bien"
   elif [ "$note" -ge 10 ]; then
      echo "moyen"
   elif [ "$note" -ge 0 ]; then
      echo "insuffisant"
   else
      echo "au revoir !"
   fi

   if [ "$note" -ge 0 ]; then
      let moyenne=$moyenne+$note
      let i=$i+1
   fi
done

if [ "$i" -le 0 ]; then
   let i=1
fi

let moyenne=$moyenne/$i
```

```
echo "La moyenne est de $moyenne ($i notes)"
```

Exercice 2 :

Ecrire un script shell qui affiche la liste des sous répertoires d'un répertoire donné (par défaut le répertoire courant).

```
#!/bin/bash
if (test $# -eq 0) then
    #Le répertoire par défaut est le répertoire courant
    Folder=.     #ou Folder=$PWD     ou   Folder=`pwd`
else
    if (test -d $1) then
            Folder=$1
    else
            echo "Le répertoire donné est invalide."
            exit
    fi
fi

for   i   in   `ls $Folder`
do
    if (test -d  $Folder/$i) then
            echo $i
    fi
done
```

Exercice 3 :

Ecrire un script shell qui précise le type du fichier passé en paramètre et ses permissions pour l'utilisateur.

```
#!/bin/bash
# script test-fichier
if [ $# -ne 1 ]; then
  echo "Usage : $0 <nom_fichier>"
  exit 1
fi

if [ ! -e "$1" ]; then
  echo "Le fichier '$1' n'existe pas"
  exit
fi

if [ ! -s "$1" ]; then
  echo "$1 a une taille nulle"
  exit
fi
```

```
# droits et type d'utilisateur
a=$(stat -c%A "$1")
p=$(stat -c%U "$1")
g=$(stat -c%G "$1")

n=1
echo "'$1' est accessible par :"
for u in "le propriétaire ($p)" "le groupe ($g)" "les autres"
do
   echo -n "* $u en"
   d=${a:n:3}
   if [ $d == --- ]
   then echo -n " rien"
   else
      [[ $d == r?? ]] && echo -n " lecture"
      [[ $d == ?w? ]] && echo -n " écriture"
      [[ $d == ??x ]] && echo -n " exécution"
   fi
  fi
   n=$((n+3))
   echo
done
# type de fichier
case ${a::1} in
-) echo "fichier standard" ;;
d) echo "dossier" ;;
l) echo "lien symbolique" ;;
esac
```

Exercice 4 :

Créer un script qui permet de calculer et d'afficher la factorielle d'un nombre donné en paramètre (ou saisi en cas d'absence de paramètres).

```
#!/bin/bash

if [ "$#" -eq 0 ]; then
   echo "Saisir une valeur : "
   read -r val
else
   val=$1
fi

# Dans le cas où c'est négatif, on rend la valeur positive
if [ "$val" -lt 0 ]; then
   let val=-1*$val
fi

result=1
val2="$val"

while [ "$val" -ne 0 ]; do
```

```
    printf "$val "
    let result=$result*$val
    let val=$val-1
    if [ "$val" -ne 0 ]; then
        printf "* "
    fi
done

echo "= $result"
```