Programmation shell

1. Structure et exécution d’un script

Le shell n’est pas qu’un simple interpréteur de commandes, mais dispose d’un véritable langage de programmationavec notamment une gestion des variables, des tests et des boucles, des opérations sur les variables, des fonctions...

Toutes les instructions et commandes sont regroupées au sein d’un script. Lors de son exécution, chaque ligne sera lue une à une et exécutée. Une ligne peut se composer de commandes internes ou externes, de commentaires ou être vide. Plusieurs instructions par lignes sont possibles, séparées par le **;** ou liées conditionnellement par **&&** ou **||**. Le **;** est l’équivalent d’un **saut de ligne**.

Par convention les noms des shell scripts se terminent généralement (pas obligatoirement) par «.sh» pour le Bourne Shell et le Bourne Again Shell, par «.ksh» pour le Korn Shell et par «.csh» pour le C Shell.

Pour rendre un script exécutable directement :

```
$ chmod u+x monscript
```

Pour l’exécuter :

```
$ ./monscript
```

Pour éviter le ./ :

```
$ PATH=$PATH:.
$ monscript
```

Notez que le point est placé en dernier dans le PATH. Le mettre en premier est un risque pour la sécurité: une nouvelle commande **ls** modifiée est placée dans votre répertoire. Imaginez les dégâts avec une commande **passwd**.

Quand un script est lancé, un nouveau shell fils est créé qui va exécuter chacune des commandes. Si c’est une commande interne, elle est directement exécutée par le nouveau shell. Si c’est une commande externe, dans le cas d’un binaire un nouveau fils sera créé pour l’exécuter, dans le cas d’un shell script un nouveau shell fils est lancé pour lire ce nouveau shell ligne à ligne.

Une ligne de commentaire commence toujours par le caractère **#**. Un commentaire peut être placé en fin d’une ligne comportant déjà des commandes.

```
# La ligne suivante effectue un ls
ls # La ligne en question
```

La première ligne a une importance particulière car elle permet de préciser quel shell va exécuter le script :

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

Dans le premier cas c'est un script Bourne Again, dans l'autre un script Korn. Vous pouvez aussi passer des options aux commandes.

2. Arguments d'un script (paramètres)

a. Paramètres de position

Les paramètres de position sont aussi des variables spéciales utilisées lors d'un passage de paramètres à un script.

Variable	Contenu
$0	Nom de la commande (du script).
$1-9	$1,$2,$3... Les neuf premiers paramètres passés au script.
$#	Nombre total de paramètres passés au script.
$*	Liste de tous les paramètres au format "$1 $2 $3 ...".
$@	Liste des paramètres sous forme d'éléments distincts "$1" "$2" "$3" ...

```
$ cat param.sh
#!/bin/bash

echo "Nom : $0"
echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3"
echo "Liste : $*"
echo "Elements : $@"

$ param.sh riri fifi loulou
Nom : ./param.sh
Nombre de parametres : 3
Parametres : 1=riri 2=fifi 3=loulou
Liste : riri fifi loulou
Elements : riri fifi loulou
```

La différence entre **$@** et **$*** ne saute pas aux yeux. Reprenez l'exemple précédent avec une petite modification:

```
$ param.sh riri "fifi loulou"
Nom : ./param.sh
Nombre de parametres : 2
Parametres : 1=riri 2=fifi loulou 3=
Liste : riri fifi loulou
Elements : riri fifi loulou
```

Cette fois-ci il n'y a que deux paramètres de passés. Pourtant les listes semblent visuellement identiques. En fait si la première contient bien :

"riri fifi loulou"

La deuxième contient :

"riri" "fifi loulou"

Soit bien deux éléments. Dans le premier exemple vous aviez :

"riri" "fifi" "loulou"

b. Redéfinition des paramètres

Outre le fait de lister les variables, l'instruction **set** permet de redéfinir le contenu des variables de position.Avec :

set valeur1 valeur2 valeur3 ...

$1 prendra comme contenu valeur1, $2 valeur2 et ainsi de suite.

```
$ cat param2.sh
#!/bin/bash
echo "Avant :"
echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3 4=$4"
echo "Liste : $*"
set alfred oscar romeo zoulou
echo "apres set alfred oscar romeo zoulou"
echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3 4=$4"
echo "Liste : $*"

$ ./param2.sh riri fifi loulou donald picsou
Avant :
Nombre de parametres : 5
Parametres : 1=riri 2=fifi 3=loulou 4=donald
Liste : riri fifi loulou donald picsou
apres set alfred oscar romeo zoulou
Nombre de parametres : 4
Parametres : 1=alfred 2=oscar 3=romeo 4=zoulou
Liste : alfred oscar romeo zoulou
```

c. Réorganisation des paramètres

La commande **shift** est la dernière commande permettant de modifier la structure des paramètres de position. Un simple appel décale tous les paramètres d'une position en supprimant le premier : $2 devient$1, $3 devient $2 et ainsi de suite. Le $1 originel disparaît. $#, $* et $@ sont redéfinis en conséquence.

La commande **shift** suivie d'une valeur n effectue un décalage de n éléments. Ainsi avec shift 4 $5 devient $1, $6 devient $2, ...

```
$ cat param3.sh
#!/bin/bash
set alfred oscar romeo zoulou
echo "set alfred oscar romeo zoulou"
echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3 4=$4"
echo "Liste : $*"
shift
echo "Après un shift"
echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3 4=$4"
echo "Liste : $*"

$ ./param3.sh
set alfred oscar romeo zoulou
Nombre de parametres : 4
Parametres : 1=alfred 2=oscar 3=romeo 4=zoulou
Liste : alfred oscar romeo zoulou
Après un shift
Nombre de parametres : 3
Parametres : 1=oscar 2=romeo 3=zoulou 4=
Liste : oscar romeo zoulou
```

d. Sortie de script

La commande **exit** permet de mettre fin à un script. Par défaut la valeur retournée est 0 (pas d'erreur) mais n'importe quelle autre valeur de 0 à 255 peut être précisée. Vous récupérez la valeur de sortie par la variable $?.

```
$ exit 1
```

3. Environnement du processus

Rappel:Seules les variables exportées sont accessibles par un processus fils. Si vous souhaitez visualiser l'environnementlié à un fils (dans un script par exemple) utilisez la commande **env**.

```
$ env
LESSKEY=/etc/lesskey.bin
NNTPSERVER=news
INFODIR=/usr/local/info:/usr/share/info:/usr/info
MANPATH=/usr/local/man:/usr/share/man
KDE_MULTIHEAD=false
SSH_AGENT_PID=28012
HOSTNAME=slyserver
DM_CONTROL=/var/run/xdmctl
XKEYSYMDB=/usr/share/X11/XKeysymDB
HOST=p64p17bicb3
SHELL=/bin/bash
TERM=xterm
PROFILEREAD=true
HISTSIZE=1000
XDM_MANAGED=/var/run/xdmctl/xdmctl-
:0,maysd,mayfn,sched,rsvd,method=classic
XDG_SESSION_COOKIE=16af07a56781b4689718210047060300-
1211264847.394692-546885666
```

```
TMPDIR=/tmp
GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/usr/share/themes//QtCurve/gtk-
2.0/gtkrc:/home/driss/.gtkrc-2.0-qtengine:/home/driss/.gtkrc-
2.0:/home/driss/.kde/share/config/gtkrc-2.0
KDE_NO_IPV6=1
GTK_RC_FILES=/etc/gtk/gtkrc:/home/driss/.gtkrc:/home/driss/.kde/share/co
nfig/gtkrc
GS_LIB=/home/driss/.fonts
WINDOWID=71303176
MORE=-sl
QTDIR=/usr/lib/qt3
XSESSION_IS_UP=yes
KDE_FULL_SESSION=true
GROFF_NO_SGR=yes
JRE_HOME=/usr/lib/jvm/jre
USER=driss
...
```

La commande **env** permet de redéfinir aussi l'environnement du processus à lancer. Cela peut être utile lorsque le script doit accéder à une variable qui n'est pas présente dans l'environnement du père, ou qu'on ne souhaite pas exporter. La syntaxe est :

```
env var1=valeur var2=valeur ... commande
```

Dans le cas de bash (c'est-à-dire hors script), env n'est pas indispensable.

```
var1=valeur var2=valeur ... commande
```

4. Substitution de commande

Le mécanisme de substitution permet de placer le résultat de commandes simples ou complexes dans une variable. Vous placez les commandes à exécuter entre des accents graves «`» ([Alt Gr] **7**).

```
$ mon_unix=`uname`
$ echo ${mon_unix}   # ou echo $mon_unix
Linux
$ machine=`uname -a | cut -d" " -f5`
echo $machine
SMP
```

Attention, seul le canal de sortie standard est affecté à la variable. Le canal d'erreur standard sort toujoursvers l'écran dans ce cas.

5. Tests de conditions

La commande **test** permet d'effectuer des tests de conditions. Le résultat est récupérable par la variable $? (code retour). Si ce résultat est 0 alors la condition est réalisée.

a. Tests sur une chaîne

- **test -z "variable"** : zero, retour OK si la variable est vide (ex : test -z "$a").
- **test -n "variable"** : non zero, retour OK si la variable n'est pas vide (texte quelconque).
- **test "variable"** = chaîne : OK si les deux chaînes sont identiques.
- **test "variable" !** = chaîne : OK si les deux chaînes sont différentes.

```
$ a=
$ test -z "$a" ; echo $?
0
$ test -n "$a" ; echo $?
1
$ a=Jules
$ test "$a" = Jules ; echo $?
0
```

Attention à bien placer vos variables contenant du texte entre guillemets. Dans le cas contraire un bug se produira si la variable est vide:

```
$ a=
$ b=toto
$ test $a = $b  && echo "ok"
bash: [: =: operateur unaire attendu
```

Alors que :

```
test "$a" = "$b" && echo "ok"
```

ne produit pas d'erreur.

b. Tests sur les valeurs numériques

Les chaînes à comparer sont converties en valeurs numériques. Bash ne gère que des valeurs entières. La syntaxe est :

```
test valeur1 option valeur2
```

et les options sont les suivantes :

Option	Rôle
-eq	Equal : égal
-ne	Not Equal : différent
-lt	Less than : inférieur
-gt	Greater than : supérieur
-le	Less or equal : inférieur ou égal
-ge	Greater or equal : supérieur ou égal

```
$ a=10
$ b=20
$ test  "$a" -ne "$b" ; echo $?
0
$ test "$a" -ge "$b" ; echo $?
1
$ test "$a" -lt "$b" && echo "$a est inferieur a $b"
10 est inferieur a 20
```

c. Tests sur les fichiers

La syntaxe est :

test option nom_fichier

et les options sont les suivantes :

Option	Rôle
-f	Fichier normal.
-d	Un répertoire.
-c	Fichier en mode caractère.
-b	Fichier en mode bloc.
-p	Tube nommé (named pipe).
-r	Autorisation en lecture.
-w	Autorisation en écriture.
-x	Autorisation en exécution.
-s	Fichier non vide (au moins un caractère).
-e	Le fichier existe.
-L	Le fichier est un lien symbolique.
-u	Le fichier existe, SUID-Bit positionné.
-g	Le fichier existe SGID-Bit positionné.

```
$ ls -l
-rw-r--r--   1 driss   users     1392 Aug 14 15:55 dump.log
lrwxrwxrwx   1 driss   users        4 Aug 14 15:21 lien_fic1 -> fic1
lrwxrwxrwx   1 driss   users        4 Aug 14 15:21 lien_fic2 -> fic2
-rw-r--r--   1 driss   users      234 Aug 16 12:20 liste1
-rw-r--r--   1 driss   users      234 Aug 13 10:06 liste2
-rwxr--r--   1 driss   users      288 Aug 19 09:05 param.sh
-rwxr--r--   1 driss   users      430 Aug 19 09:09 param2.sh
-rwxr--r--   1 driss   users      292 Aug 19 10:57 param3.sh
drwxr-xr-x   2 driss   users     8192 Aug 19 12:09 rep1
-rw-r--r--   1 driss   users     1496 Aug 14 16:12 resultat.txt
-rw-r--r--   1 driss   users     1715 Aug 16 14:55 toto.txt
-rwxr--r--   1 driss   users       12 Aug 16 12:07 voir_a.sh
$ test -f lien_fic1 ; echo $?
1
$ test -x dump.log ; echo $?
1
```

```
$ test -d rep1 ; echo $?
0
```

d. Tests combinés par des critères ET, OU, NON

Critère	Action
-a	AND, ET logique
-o	OR, OU logique
!	NOT, NON logique
(. . .)	groupement des combinaisons. Les parenthèses doivent être verrouillées \(...\).

Vous pouvez effectuer plusieurs tests avec une seule instruction. Les options de combinaison sont les mêmes que pour la commande **find**.

```
$ test -d "rep1" -a -w "rep1" && echo "rep1: repertoire, droit en  ecriture"
rep1: repertoire, droit en ecriture
```

e. Syntaxe allégée

Le mot test peut être remplacé par les crochets ouverts et fermés **[...]**. Il faut respecter un espace avant et après les crochets.

```
$ [ "$a" -lt "$b" ] && echo "$a est inferieur a $b"
10 est inferieur a 20
```

Le bash (et le ksh) intègre une commande interne de test qui se substitue au binaire test. Dans la pratique,la commande interne est entièrement compatible avec la commande externe mais bien plus rapide car il n'y a pas de création de nouveau processus. Pour forcer l'utilisation de la commande interne,utilisez les doubles crochets **[[...]]**.

```
$ [[ "$a" -lt "$b" ]] && echo "$a est inferieur a $b"
10 est inferieur a 20
```

La syntaxe bash permet d'utiliser l'opérateur ==, mais aussi && (-a, et) et || (-o, ou) au sein des crochets:

```
$ i=1; j=2; [[ $i == 1 && $j == 2 ]] && echo OK
OK
```

6. if ... then ... else

La structure **if then else fi** est une structure de contrôle conditionnelle.

```
if <commandes_condition>
then
   <commandes exécutées si condition réalisée>
elif <condition>
```

```
then
   <commandes exécutées si condition réalisée>else
   <commandes exécutées si dernière condition pas réalisée>
fi
```

Vous pouvez préciser **elif**, en fait un else if. Si la dernière condition n'est pas réalisée, on en teste une nouvelle.

```
$ cat param4.sh
#!/bin/bash
if [ $# -ne 0 ]
then
       echo "$# parametres en ligne de commande"
else
       echo "Aucun parametre; set alfred oscar romeo zoulou"
       set alfred oscar romeo zoulou
fi

echo "Nombre de parametres : $#"
echo "Parametres : 1=$1 2=$2 3=$3 4=$4"
echo "Liste : $*"

$ ./param4.sh titi toto
2 parametres en ligne de commande
Nombre de parametres : 2
Parametres : 1=toto 2=titi 3= 4=
Liste : toto titi

$ ./param4.sh
Aucun parametre; set alfred oscar romeo zoulou
Nombre de parametres : 4
Parametres : 1=alfred 2=oscar 3=romeo 4=zoulou
Liste : alfred oscar romeo zoulou
```

La syntaxe proche du C permet d'utiliser des conditions plus pratiques. Par exemple, vous voulez savoir si un fichier contient un motif de recherche en utilisant grep, au sein d'une condition if. Voici deux manièresde procéder:

```
grep -q 20 input
if [[ $? -eq 0 ]]
then
   echo "trouvé"
fi
```

Vous testez le code retour de la commande **grep**, qui va retourner 0 si 20 est présent dans input. Vous êtes obligé de le faire en deux lignes, et en utilisant -q (quiet) pour masquer la sortie standard. Maintenant, même chose avec la nouvelle syntaxe:

```
if ( $(grep -q 20 input ))
then
   echo "trouvé"
fi
```

L'expression retourne 0 et est interprétée comme vraie par le shell. N'est-ce

pas plus simple?

7. Choix multiples case

La commande **case ... esac** permet de vérifier le contenu d'une variable ou d'un résultat de manière multiple.

```
case Valeur in
   Modele1) Commandes ;;
   Modele2) Commandes ;;
   *) action_defaut ;;
esac
```

Le modèle est soit un simple texte, soit composé de caractères spéciaux. Chaque bloc de commandes lié au modèle doit se terminer par deux points-virgules. Dès que le modèle est vérifié, le bloc de commandes correspondant est exécuté. L'étoile en dernière position (chaîne variable) est l'action par défaut si aucun critère n'est vérifié. Elle est facultative.

Caractère	Rôle
*	Chaîne variable (même vide)
?	Un seul caractère
[...]	Une plage de caractères
[!...]	Négation de la plage de caractères
|	OU logique

```
$ cat case1.sh
#!/bin/bash
if [ $# -ne 0 ]
then
	echo "$# parametres en ligne de commande"
else
	echo "Aucun parametre; set alfred oscar romeo zoulou"
	exit 1
fi

case $1 in
	a*)
		echo "Commence par a"
		;;
	b*)
		echo "Commence par b"
		;;
	fic[123])
		echo "fic1 fic2 ou fic3"
		;;
	*)
		echo "Commence par n'importe"
		;;
esac

exit 0
```

```
$ ./case1.sh "au revoir"
Commence par a
$ ./case1.sh bonjour
Commence par b
$ ./case1.sh fic2
fic1 ou fic2 ou fic3
$ ./case1.sh erreur
Commence par n’importe
```

8. Saisie de l’utilisateur

La commande **read** permet à l’utilisateur de saisir une chaîne et de la placer dans une ou plusieurs variable.La saisie est validée par [Entrée].

```
read var1 [var2 ...]
```

Si plusieurs variables sont précisées, le premier mot ira dans var1, le second dans var2, et ainsi de suite. S’il y a moins de variables que de mots, tous les derniers mots vont dans la dernière variable.

```
$ cat read.sh
#!/bin/bash
echo "Continuer (O/N) ? "
read reponse
echo "reponse=$reponse"
case $reponse in
        O)
                echo "Oui, on continue"
                ;;
        N)
                echo "Non, on s’arrête"
                exit 0
                ;;
        *)
                echo "Erreur de saisie (O/N)"
                exit 1
                ;;
esac
echo "Vous avez continue. Tapez deux mots ou plus :"
read mot1 mot2
echo "mot1=$mot1 et mot2=$mot2"
exit 0
$ ./read.sh
Continuer (O/N) ? O
reponse=O
Oui, on continue
Vous avez continue. Tapez deux mots ou plus :salut les amis
mot1=salut
mot2=les amis
```

9. Les boucles

Elles permettent la répétition d’un bloc de commandes soit un nombre limité

de fois, soit conditionnellement. Toutes les commandes à exécuter dans une boucle se placent entre les commandes **do** et **done**.

a. Boucle for

La boucle **for** ne se base pas sur une quelconque incrémentation de valeur mais sur une liste de valeurs,de fichiers...

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

La liste représente un certain nombre d'éléments qui seront successivement attribués à var.

Avec une variable

```
$ cat for1.sh
#!/bin/bash
for params in $@
do
       echo "$params"
done


$ ./for1.sh test1 test2 test3
test1
test2
test3
```

Liste implicite

Si vous ne précisez aucune liste à for, alors c'est la liste des paramètres qui est implicite. Ainsi le script précédent aurait pu ressembler à :

```
for params
do
       echo "$params"
done
```

Avec une liste d'éléments explicite

Chaque élément situé après le «in» sera utilisé pour chaque itération de la boucle, l'un après l'autre.

```
$ cat for2.sh
#!/bin/sh
for params in liste liste2
do
       ls -l $params
done
```

```
$ ./for2.sh
-rw-r--r--   1 oracle   system      234 Aug 19 14:09 liste
-rw-r--r--   1 oracle   system      234 Aug 13 10:06 liste2
```

Avec des critères de recherche sur nom de fichiers

Si un ou plusieurs éléments de la liste correspond à un fichier ou à un motif de fichiers présents à la position actuelle de l'arborescence, la boucle for considère l'élément comme un nom de fichier.

```
$ cat for3.sh
#!/bin/bash
for params in *
do
        echo "$params \c"
        type_fic=`ls -ld $params | cut -c1`
        case $type_fic in
                -)      echo "Fichier normal" ;;
                d)      echo "Repertoire" ;;
                b)      echo "mode bloc" ;;
                l)      echo "lien symbolique" ;;
                c)      echo "mode caractere" ;;
                *)      echo "autre" ;;
        esac

done


$ ./for3.sh
case1.sh Fichier normal
dump.log Fichier normal
for1.sh Fichier normal
for2.sh Fichier normal
for3.sh Fichier normal
lien_fic1 lien symbolique
lien_fic2 lien symbolique
liste Fichier normal
liste1 Fichier normal
liste2 Fichier normal
param.sh Fichier normal
param2.sh Fichier normal
param3.sh Fichier normal
param4.sh Fichier normal
read.sh Fichier normal
rep1 Repertoire
resultat.txt Fichier normal
toto.txt Fichier normal
voir_a.sh Fichier normal
```

Avec un intervalle de valeurs

Il existe deux méthodes pour compter de 1 à n avec une boucle for. La première consiste à utiliser une substitution de commande avec la commande **seq**. Sa syntaxe de base prend un paramètre numérique et compte de 1 à ce paramètre. Le manuel vous apprendra qu'il est possible de démarrer à n'importe quelle valeur, avec n'importe quel incrément.

```
$ seq 5
1
2
3
4
5
```

Avec for, voici ce que cela donne :

```
$ for i in $(seq 5); do echo $i; done
1
2
3
4
5
```

La seconde méthode consiste à utiliser une syntaxe proche du langage C :

```
$ for ((a=1 ; a<=5 ; a++)); do echo $a; done
1
2
3
4
5
```

Avec une substitution de commande

Toute commande produisant une liste de valeurs peut être placée à la suite du «in» à l’aide d’une substitutionde commande. La boucle for prendra le résultat de cette commande comme liste d’éléments sur laquelle boucler.

```
$ cat for4.sh
#!/bin/bash
echo "Liste des utilisateurs dans /etc/passwd"
for params in `cat /etc/passwd | cut -d: -f1`
do
        echo "$params "
done


$ ./for4.sh
Liste des utilisateurs dans /etc/passwd
root
nobody
nobodyV
daemon
bin
uucp
uucpa
auth
cron
lp
tcb
adm
ris
carthic
ftp
stu
```

...

b. Boucle while

La commande **while** permet une boucle conditionnelle «tant que». Tant que la condition est réalisée, le bloc de commande est exécuté. On sort si la condition n'est plus valable.

```
while condition
do
   commandes
done
```

ou :

```
while
bloc d'instructions formant la condition
do
   commandes
done
```

Par exemple :

```
$ cat while1.sh
#!/bin/bash
while
        echo "Chaine ? \c"
        read nom
        [ -z "$nom" ]
do
        echo "ERREUR : pas de saisie"
done
echo "Vous avez saisi : $nom"
```

Lecture d'un fichier ligne à ligne

```
#!/bin/bash
cat toto.txt | while read line
do
          echo "$line"
done
```

ou :

```
#!/bin/bash
while read line
do
          echo "$line"
done < toto.txt
```

Il y a une énorme différence entre les deux versions. Dans la première, notez la présence du tube (pipe): la boucle est exécutée dans un second processus. Aussi toute variable modifiée au sein de cette boucle perd sa valeur en sortie!

c. Boucle until

La commande **until** permet une boucle conditionnelle «jusqu'à». Dès que la condition est réalisée, on sort de la boucle.

```
until condition
do
   commandes
done
```

ou :

```
until
bloc d'instructions formant la condition
do
   commandes
done
```

d. true et false

La commande **true** ne fait rien d'autre que de renvoyer 0. La commande **false** renvoie toujours 1. De cette manière il est possible de réaliser des boucles sans fin. La seule manière de sortir de ces boucles est un exit ou un break.

Par convention, tout programme qui ne retourne pas d'erreur retourne 0, tandis que tout programme retournant une erreur, ou un résultat à interpréter, retourne autre chose que 0. C'est l'inverse en logique booléenne.

```
while true
do
   commandes
   exit / break
done
```

e. break et continue

La commande **break** permet d'interrompre une boucle. Dans ce cas le script continue après la commande **done**. Elle peut prendre un argument numérique indiquant le nombre de boucles à sauter, dans le cas de boucles imbriquées (rapidement illisible).

```
while true
do
   echo "Chaine ? \c"
   read a
   if [ -z "$a" ]
   then
      break
   fi
done
```

La commande **continue** permet de relancer une boucle et d'effectuer un

nouveau passage. Elle peut prendre un argument numérique indiquant le nombre de boucles à relancer (on remonte de n boucles). Le script redémarre à la commande **do**.

f. Boucle select

La commande **select** permet de créer des menus simples, avec sélection par numéro. La saisie s'effectue au clavier avec le prompt de la variable PS3. Si la valeur saisie est incorrecte, une boucle s'effectue et le menu s'affiche à nouveau. Pour sortir d'un select il faut utiliser un **break**.

```
select variable in liste_contenu
do
   traitement
done
```

Si in liste_contenu n'est pas précisé, ce sont les paramètres de position qui sont utilisés et affichés.

```
$ cat select.sh
#!/bin/bash
PS3="Votre choix :"
echo "Quelle donnee  ?"
select reponse in Jules Romain Francois quitte
do
        if [[ "$reponse" = "quitte" ]]
        then
                break
        fi
        echo "Vous avez choisi $reponse"
done
echo "Au revoir."
exit 0

$ ./select.sh
Quelle donnee  ?
1) Jules
2) Romain
3) Francois
4) quitte
Votre choix :1
Vous avez choisi Jules
Votre choix :2
Vous avez choisi Romain
Votre choix :3
Vous avez choisi Francois
Votre choix :4
Au revoir.
```

10. Les fonctions

Les fonctions sont des bouts de scripts nommés, directement appelés par leur nom, pouvant accepter des paramètres et retourner des valeurs. Les noms de fonctions suivent les mêmes règles que les variables sauf qu'elles ne peuvent

pas être exportées.

```
nom_fonction ()
{
   commandes
   return
}
```

La commande **return** permet d'affecter une valeur de retour à une fonction. Il ne faut surtout pas utiliser la commande **exit** pour sortir d'une fonction, sinon on quitte le script.

```
$cat mesFonctions
methode1 ()
{
      ls -l $@
}
 methode2 ()
{
      ls -i $@
}
methode1
```

```
$./mesFonctions
 51794715 apache_mod_h264_streaming-2.2.7.tar.gz
   75914 Arduino
51773445 Bureau
  176011 debugs
51787215 dessin.png
51773449 Documents
51773451 Images
51773447 Modèles
51773450 Musique
  137984 neoload_projects
 2753810 NetBeansProjects
51773448 Public
51905148 R
```

11. Calculs et expressions

a. expr

La commande **expr** permet d'effectuer des calculs sur des valeurs numériques, des comparaisons, et de la recherche dans des chaînes de texte.

Opérateur	Rôle
+	Addition.
-	Soustraction. L'étoile étant reconnue par le shell comme un wildcard, il faut la verrouiller avec un antislash : *.
*	Multiplication.
/	Division.
%	Modulo.
!=	Différent. Affiche 1 si différent, 0 sinon.
=	Égal. Affiche 1 si égal, 0 sinon.
<	Inférieur. Affiche 1 si inférieur, 0 sinon.
>	Supérieur. Affiche 1 si supérieur, 0 sinon.
<=	Inférieur ou égal. Affiche 1 si inférieur ou égal, 0 sinon.
>=	Supérieur ou égal. Affiche 1 si supérieur ou égal, 0 sinon.
:	Recherche dans une chaîne. Ex : expr Jules : J* retourne 1 car Jules commence par J. Syntaxe particulière : expr "Jules" : ".*" retourne la longueur de la chaîne.

```
$ expr 7 + 3
10
$ expr 7 \* 3
21
$ a=$(expr 13 - 10)
$ echo $a
3
$ cat expr1.sh
#!/bin/bash
cumul=0
compteur=0
nb_boucles=10
while [ "$compteur" -le "$nb_boucles" ]
do
	cumul=$(expr $cumul + $compteur)
	echo "$cumul=$cumul, boucle=$compteur"
	compteur=$(expr $compteur + 1)
done
$ ./expr1.sh
cumul=0, boucle=0
cumul=1, boucle=1
cumul=3, boucle=2
cumul=6, boucle=3
cumul=10, boucle=4
cumul=15, boucle=5
cumul=21, boucle=6
cumul=28, boucle=7
cumul=36, boucle=8
cumul=45, boucle=9
cumul=55, boucle=10
$ expr "Jules Cesar" : ".*"
11
```

b. Calculs avec bash

Le bash propose une forme simple de calculs sur les entiers, en plaçant l'opération entre **$((...))**:

```
$ a=1
$ a=$((a+1))
$ echo $a
2
$ b=2
$ a=$((a*b))
$ echo $a
4
```

Vous n'avez pas besoin de spécifier les $ des noms des variables entre les doubles parenthèses.

c. Calculs de nombres réels

Ni expr, ni les possibilités internes de bash ne permettent d'effectuer des calculs avec des nombres réels. Pour cela, il faut utiliser la commande **bc**, un langage de calcul. S'agissant d'un langage de programmation à part entière, bc sort du cadre de cette formation big data, mais voici cependant comment effectuer quelques calculs simples et comparaisons. Le principe est d'envoyer les expressions de calculs en entrée standard de la commande. Notez la présence du -l qui précise l'utilisation d'une bibliothèque mathématique permettant le travail sur les décimaux:

```
$ echo "2+2" | bc -l
4
$ echo "1/3" | bc -l
.33333333333333333333
```

Il est possible de limiter la longueur des chiffres après la virgule avec scale.

```
$ echo "scale=2 ; 1/3" | bc -l
.33
```

Voici par exemple comment calculer un pourcentage avec deux chiffres après la virgule. Soit le script suivant:

```
BLOCK_USAGE=1245786
BLOCK_LIMIT=1964856
WARNING=60
USAGE_PERCENT=$(echo "scale=2; ${BLOCK_USAGE}*100/${BLOCK_LIMIT}" | bc -l)
echo $USAGE_PERCENT
if (( $(echo "${USAGE_PERCENT}>=${WARNING}" |bc -l) ))
then
  echo "WARNING"
fi
```

Son exécution va afficher:

```
63.40
```

WARNING

12. Une variable dans une autre variable

Voici un exemple :

```
$ a=Jules
$ b=a
$ echo $b
a
```

Comment afficher le contenu de a et pas simplement a ? En utilisant la commande **eval**. Cette commande située en début de ligne essaie d'interpréter, si possible, la valeur d'une variable précédée par deux «$», comme étant une autre variable.

```
$ eval echo \$$b
Jules

$ cat eval.sh
cpt=1
for i in a b c
do
   eval $i=$cpt
   cpt=$((cpt+1))
done
echo $a $b $c

$ ./eval.sh
1 2 3
```

13. Commande «:»

La commande **«:»** est généralement totalement inconnue des utilisateurs Unix. Elle retourne toujours la valeur 0 (réussite). Elle peut donc être utilisée pour remplacer la commande **true** dans une boucle par exemple :

```
while :
do
   ...
done
```

Cette commande placée en début de ligne, suivie d'une autre commande, traite la commande et ses arguments mais ne fait rien, et retourne toujours 0.

```
$ : ls
```

TRAVAUX PRATIQUES

But : dans un fichier texte nous avons les lignes suivantes :

1 3

5 7

12 19

...

Écrivez un script qui accepte ce fichier comme paramètre, qui le lit et pour chacune de ses lignes calcule la somme des deux nombres et l'affiche sous la forme suivante :

1 + 3 = 4

5 + 7 = 12

12 + 19 = 31

...

1. Vérifiez en début de script que le nombre de paramètres passé au script est égal à 1 et que ce paramètre est bien un fichier.

   ```
   [ $# -ne 1 -o ! -f $1 ] && exit
   ```

2. Initialisez une variable à 0 qui contiendra le total de chacune des lignes.

   ```
   result=0
   ```

3. Le fichier doit être lu ligne à ligne ; écrivez une boucle qui lit une ligne tant que la fin du fichier n'est pas atteinte :

   ```
   while read ligne
   do
   ...
   done < $1
   ```

4. Dans la boucle, récupérez les deux valeurs des lignes, le séparateur est l'espace. Placez les deux valeurs dans les variables c1 et c2 :

   ```
   while read ligne
   do
       c1=$(echo $ligne | cut -d" " -f1)
       c2=$(echo $ligne | cut -d" " -f2)
       ...
   done < $1
   ```

   voir page 15 équivalent à « cat toto,txt| while read ligne »

5. Additionnez ces deux valeurs et placez le résultat dans result.

Affichez result.

```
[ $# -ne 1 -o ! -f $1 ] && exit
result=0
while read ligne
do
    c1=$(echo $ligne | cut -d" " -f1)
    c2=$(echo $ligne | cut -d" " -f2)
    result=$((c1+c2))
    echo -e "$c1 + $c2 = $result"
done < $1
```

6. Un nombre est premier seulement si ses seuls diviseurs sont 1 et lui-même.

Autrement dit, si on peut diviser un nombre par autre chose que par un et lui-même et que le résultat de cette division est un entier (ou que le reste de cette division est 0, ce qui revient au même) alors il n'est pas premier.

Le chiffre 1 n'est pas premier car il n'a pas deux diviseurs différents.

Le chiffre 2 est premier (2 x 1 donc deux diviseurs différents).

Aucun nombre pair (sauf le chiffre 2) n'est premier (car ils sont tous divisibles par 2).

Liste des dix premiers nombres premiers : 2 3 5 7 11 13 17 19 23 29.

Écrivez une fonction appelée **est_premier**, qui prend comme paramètre un nombre et qui renvoie le code retour 0 si le nombre est premier, 1 sinon.

```
est_premier()
{
	[ $1 -lt 2 ] && return 1
	[ $1 -eq 2 ] && return 0
	[ $(expr $1 % 2) -eq 0 ] && return 1
	i=3
	while [ $((i*i)) -le $1 ]
	do
		[ $(expr $1 % $i) -eq 0 ] && return 1
		i=$((i+2))
	done
	return 0
}
```

EXERCICES

1-Pourquoi est-il important de commencer votre script en indiquant la commande du shell ?

2- Un script se voit transmettre 10 paramètres. Comment récupérer le dixième le plus rapidement possible ?

- A - par $10.

- B - en écrivant neuf « shift » et en récupérant $1.
- C - un shift puis $9.
- D - par : A=10 puis ${$A}.

3- Comment depuis votre shell pouvez-vous récupérer la valeur retournée par « exit » d'un script quelconque ?

4- Comment récupérer, connaissant un login, son UID dans une variable de même nom ?

5- Quelle ligne écrire dans un script pour sortir avec un message d'erreur et un code de retour 1 si le nombre de paramètres passés est égal à 0 ?

- A - test $# != 0 || echo Erreur ; exit 1
- B - [$# -eq 0] && { echo Erreur ; exit 1 ; }
- C - test $0 -ne 0 && (echo Erreur; exit 1)
- D - [[$# -eq 0]] && echo Erreur && exit 1

6- Si le fichier fic existe et est lisible, listez son contenu. Sinon, créez-le.

7- À l'aide d'un if, s'il n'y a aucun paramètre transmis au script, mettez-en deux prépositionnés : -l et -r.

8- Un programme attend trois valeurs au clavier, les unes après les autres, séparées par des espaces, sur la même ligne. Quelle doit être la commande à utiliser ?

9- Comment faire une boucle pour lister tous les paramètres ? Choisissez deux réponses.

- A - for param ; do echo $param ; done
- B - while [$# -ne 0]; do echo $1; shift ; done
- C - until [$# -ne 0]; do shift; echo $1; done
- D - a=1; while [$a -ne $#]; do echo ${$a}; a=$((a+1)); done

10- Comment sortir d'une boucle sans fin ?

11- Comment programmer une fonction abs qui retourne la valeur absolue d'un nombre ?