# Module administration système et réseau GNU/Linux

## Introduction au scripting Bash


L.D.N.R. La Distance Nous Rapproche
Buroparc2
150 rue de la Découverte – 31670 LABÈGE

Tel : 05.61.00.14.85 – FAX : 09.72.15.22.73
RCS Toulouse : 521 323 964 00036
SARL au capital de 2000 Euros.

Enregistré auprès des services de la DIRECCTE sous le n°73310567731.
Cet enregistrement ne vaut pas agrément de l'état.

# Table des matières

# A. Introduction

Un script est une suite d'instructions, de commandes qui constituent un scénario d'actions. C'est un **fichier texte** que l'on peut **exécuter**, c'est à dire, lancer comme une commande.

Il ne suffit pas d'écrire une suite d'instructions pour que le système puisse l'exécuter. Il faut également préciser à la première ligne du fichier, l'interprète pour lequel ce script est écrit. C'est l'objet de la première ligne d'un script : la ligne "**shebang**". Pour un script en shell bash, la ligne se présente ainsi : ***#!/bin/bash*** *(le symbole dièse en fait parti)*

```
$ chmod +x script.sh
$ ls -l
-rwxr-xr-x   1 stag  stags  26 Sep  5 15:31 script.sh
```

Quatre solutions sont possibles pour exécuter un script.

En utilisant ./ si l'on se trouve dans le répertoire du script :

```
$ ./script.sh
```

ou en spécifiant le chemin absolu :

```
$ /home/stag/script.sh
```

si script se trouve dans le répertoire /home/stag.

Une autre solution est de modifier la variable d'environnement PATH et d'y faire figurer le répertoire qui contient le script à exécuter. Dans ce cas, il est possible d'invoquer le script depuis n'importe quel endroit du système de fichiers.

```
$ PATH=$PATH:/home/stag/
$ script.sh
```

Enfin, une dernière solution est d'appeler directement l'interprète et de lui transmettre le script à exécuter.

```
$ bash script.sh
```

**La variable d'environnement** PATH :

Que contient-elle ?

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

À quoi sert-elle ?
C'est tout simplement ce qui vous évite de devoir saisir le chemin absolu des commandes. Par exemple quand vous souhaitez tester la connectivité réseau d'une machine vous faites « ping IP » et non « /bin/ping IP ».

# B. Les variables

## 1. Déclaration et affectation d'une variable

Déclaration :

```
$ mavariable=Bonjour
```

L'affichage se fait par :

```
$ echo $mavariable
Bonjour
```

Autre exemple :

```
$ var=debut
$ echo $var
debut
$ echo $varant
(aucun resultat)
$ echo ${var}ant
debutant
```

### 1.1 Simple quote (ou apostrophe) / Doubles quotes (ou guillemets)

Une variable peut contenir des caractères spéciaux, le plus souvent des espaces. On peut soit vérouiller un par un les caractères spéciaux soit utiliser les simples quotes (l'apostrophe) ou double quotes (les guillemets) :

- var=Bien\ le\ bonjour # Solution lourde
- var="Bien le bonjour" # Solution correcte
- var='Bien le bonjour' # Solution correcte

Les doubles quotes délimitent une chaîne de caractères, mais les noms de variable sont interprétés par le shell. Par exemple :

```
$ variable="secret"
$ echo "Mon mot de passe est $variable."
Mon mot de passe est secret.
```

### 1.2 Back-quote (apostrophe inversée ou accent grave)

Bash considère que les back-quotes « ` » (*AltGr+7*) délimitent une **commande à exécuter**. Les noms de variable et les commandes sont donc interprétés. Par exemple :

```
$ uname
Linux
$ var=`uname`
$ echo $var
Linux
```

## 1.3 Et si les variables ne sont pas définies ?

Exemple :

```
$ echo Marie est $choix jolie
Marie est jolie
$ echo Marie est ${choix:-très}, jolie Alice est $choix souriante
Marie est très jolie, Alice est souriante
$ echo Marie est ${choix=très} jolie, Alice est $choix souriante
Marie est très jolie, Alice est très souriante
```

***"choix:-initial"*** utilise la valeur initial lors de cette seule utilisation de la variable « choix » alors que ***"choix=initial"*** initialise la variable choix et lui laisse cette valeur pour les utilisations postérieures.

Pour saisir plusieurs variables d'un coup, on utilise « **read** ». Exemple :

```
$ read verbe objet sujet
est noir chat
$ echo $verbe
est
$ echo Le $sujet $verbe $objet
Le chat est noir
```

Les mots qui restent sont affectés à la dernière variable :

```
$ read verbe objet sujet
est noir petit chat
$ echo $sujet
petit chat
```

Pour saisir des mots composés :

```
$ read mot1 mot2
mot\ compte\ double  mot\ compte\ triple
$ echo $mot2
mot compte triple
```

Utilisation de « read » dans un script

**Script readtest.sh :**

```
#!/bin/bash
echo -n "Saisissez votre prénom : "
read prenom
echo "Vous vous appelez $prenom"
```

## 1.4 Variables spéciales

La variable **$?** permet de **connaître le code de retour de la dernière commande effectuée** qui **vaut généralement 0 si cette commande s'est bien déroulée** et autre chose sinon.

La variable **$$** retourne le PID du shell.

La variable **$!** retourne le PID du dernier processus lancé en arrière-plan.

Pour obtenir la longueur d'une chaîne on utilise la caractère **#** :

```
$ a=Coucou
$ echo ${#a}
6
```

## 1.5 Variables d'environnement

Une variable d'environnement est une **variable accessible par tous les processus fils du shell courant**. Pour créer une variable d'environnement, on exporte la valeur d'une variable avec la commande export.

```
$ export variable
```

Pour illustrer la différence entre une variable locale et une variable d'environnement, il suffit de créer une variable d'environnement, de lancer un shell fils et d'afficher la variable.

```
$ ma_variable=toto
$ export ma_variable
$ bash
$ echo $ma_variable
toto
$ exit
```

Dans ce cas, la valeur de la variable est accessible depuis le shell fils. Si on essaye de faire la même chose sans exporter la variable, la valeur ne sera pas accessible.

```
$ ma_variable=toto
$ bash
$ echo $ma_variable

$ exit
```

La commande **env** permet de récupérer la liste des variables d'environnement du système d'exploitation. D'une manière générale, les variables d'environnement sont notées en majuscules.

## 1.6 Manipuler les chaînes de caractères

### Extraction d'une sous-chaîne

*${chaine:position} :*

```
$ chaine="ABCD"
$ echo ${chaine:0}
ABCD
$ echo ${chaine:1}
BCD
```

*${chaine:position:longueur} :*

```
$ chaine="ABCD12345"
$ echo ${chaine:0:4}
ABCD
$ echo ${chaine:4:5}
12345
```

### Conversion d'une chaîne de caractères en majuscule/minuscule

```
Minuscule
$ chaine="Homer Simpson"
$ echo ${chaine,,}
homer simpson
Majuscule
$ echo ${chaine^^}
HOMER SIMPSON
```

## 1.7 Les tableaux

L'interpréteur bash permet d'utiliser des **tableaux indicés** (dit aussi **indexés**) ou **associatifs** (dit aussi **tables de hachage**).

**Un tableau indicé** est un tableau dont les index permettant d'accéder aux éléments du tableau sont des entiers positifs ou nuls.

Dans un **tableau associatif**, les index permettant d'accéder aux éléments du tableau ne sont plus des entiers positifs ou nuls mais des chaînes de caractères appelées clés.

### 1.7.1 Tableau indexé

Pour créer un ou plusieurs tableau classiques vides, on utilise généralement l'option -a de la commande interne **declare.**

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

```
$ tab=( "Hello" "Coucou" "Bonjour" )
```

Par défaut, Hello aura pour index la valeur 0, Coucou la valeur 1 et Bonjour la valeur 2.

On peut illustrer ce tableau de cette manière :

| tab | |
|---|---|
| ***Indice*** | ***Valeur*** |
| 0 | Hello |
| 1 | Coucou |
| 2 | Bonjour |

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

```
$ tab=( "Hello" [42]="Coucou" "Bonjour" )
```

Hello aura pour index la valeur 0, Coucou la valeur 42 et Bonjour la valeur 43.

| tab | |
|---|---|
| ***Indice*** | ***Valeur*** |
| 0 | Hello |
| 42 | Coucou |
| 43 | Bonjour |

**Pour afficher un des éléments :**

```
$ echo ${tab[0]}
Hello
```

**Pour lister tous les éléments :**

```
$ echo ${tab[*]} ou bien $ echo ${tab[@]}
Hello Coucou Bonjour
```

Pour **connaitre le nombre d'éléments** on utilise le caractère #:

```
$ echo ${#tab[*]}
3
```

Si l'index est passé à travers une variable, on ne met pas le symbole **$** devant la variable :

```
$ varindex=2
$ echo ${tab[varindex]}
Bonjour
```

**Ajouter une valeur au tableau :**

Exemple 1 :

```
$ tab[1]=Salut
$ echo ${tab[*]}
Hello Salut Coucou Bonjour
```

Exemple 2 :

```
$ tab[44]=Hi
$ echo ${tab[*]}
Hello Salut Coucou Bonjour Hi
```

**Remplacer une valeur du tableau par une autre :**

```
$ tab[44]=Hola
$ echo ${tab[*]}
Hello Salut Coucou Bonjour Hola
```

### 1.7.2 Tableau associatif

Les tableaux associatifs peuvent être créés de la même manière, mais une clé doit bien sûr être précisée pour chaque élément et ils doivent être créés de façon explicite à l'aide du mot-clef declare, suivi de -A pour un tableau associatif :

```
$ declare -A tab_asso
$ tab_asso=( ['nom']="simpson" ['prenom']="homer" ['tel']="0123456789" )
```

ou bien

```
$ declare -A tab_asso=( ['nom']="simpson" ['prenom']="homer"
['tel']="0123456789" )
```

| tab_asso | |
|---|---|
| ***Clé*** | ***Valeur*** |
| nom | simpson |
| prenom | homer |
| tel | 0123456789 |

**Lecture d'un élément :**

```
$ echo ${tab_asso["prenom"]}
homer
```

**Lister tous les éléments :**

```
$ echo ${tab_asso[@]}
0123456789 homer simpson
```

**Ajout d'un élément :**

```
$ tab_asso["nourriture"]=Donuts
$ echo ${tab_asso["nourriture"]}
Donuts
```

**Remplacement d'un élément :**

```
$ tab_asso[prenom]=Marge
$ echo ${tab_asso["prenom"]}
Marge
```

**Lister le nombre d'éléments :**

```
$ echo ${#tab_asso[@]}
4
```

Lister les clés :

```
$ echo ${!tab_asso[@]}
tel nourriture prenom nom
```

### 1.7.3 Destruction d'un élément ou d'un tableau

***Destruction d'un élément de tableau***

```
$ declare -a tableau_indi=( "un" "deux" "trois" "quatre" )
$ echo ${tableau_indi[@]}
un deux trois quatre
$ unset tableau_indi[1]
$ echo ${!tableau_indi[@]}
0 2 3
$ echo ${tableau_indi[@]}
un trois quatre
```

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

***Destruction d'un tableau entier***

Valable pour les tableaux associatifs et indexés (les trois lignes suivantes sont équivalentes) :

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

# C. Tests de condition

## 1. Tests sur une chaîne

**test -z "variable"** : retour OK si la variable est vide (ex : test -z "$a").

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

1. On déclare la variable « a » avec une valeur vide (pas de valeur).
2. On teste la variable pour vérifier qu'elle est vide avec « test -z ».
3. On chaine (en utilisant « **;** ») avec l'affichage du code de retour (**$?**) de la commande « test ». Si le code de retour est 0, alors la condition est vrai et donc la chaine est bien vide. **Si la chaine n'est pas vide** alors le code de retour sera une valeur différente de « 0 » (1 en général) :

```
$ a=coucou
$ test -z "$a" ; echo $?
1
```

```
# équivalent et plus simple
$ a=
$ [ -z "$a" ]; echo $?
0
```

**test -n "variable" :** retour OK si la variable n'est pas vide (texte quelconque).

```
$ a=coucou
$ test -n "$a" ; echo $?
0
```

**test "variable" = chaîne** : OK si les deux chaînes sont identiques.

```
$ a=coucou
$ test "$a" = coucou ; echo $?
0
ou bien
$ [ "$a" = coucou ]; echo $
0
```

**test "variable" ! = chaîne :** OK si les deux chaînes sont différentes

```
$ a=coucou
$ test "$a" != bonjour ; echo $?
0
```

# 2. Enchaînements conditionnels

L'enchainement avec **le symbole « ; »** permet de chainer plusieurs commandes sur une seule ligne. Mais elle ne prend pas de conditions dans le sens où si la première commande échoue ou pas, la commande suivante sera tout de même exécutée.

**L'intérêt d'un test** est justement de pouvoir déclencher une action ou non suivant le code de retour de la commande précédente.

```
# Avec action si condition vérifiée grâce à &&                    $ a=
$ [ -z "$a" ] && echo 'La variable est vide'
$a est vide
```

On peut traduire la commande ci-dessus par : **Si** la variable « a » est vide **alors** on affiche le message « La variable est vide ».

Si nous passons une valeur à la variable « a » alors le message ne sera pas affiché :

```
$ a=10
$ [ -z "$a" ] && echo 'La variable est vide'
$
```

On utilise souvent le double symbole « **&&** » avec Debian lors de la mise à jour du système :

```
$ sudo apt-get update && sudo apt-get dist-upgrade
```

On met à jour les dépôts et si cette commande renvoie un code retour 0 alors la commande suivante est lancée.

**Le double &&** peut se résumer ainsi : si la commande précédente est OK, alors la commande suivante est exécutée.

Il existe une alternative à && dont l'action est le contraire de celle-ci : || (double pipe)

**Le double pipe « || »** peut se résumer ainsi : si la commande précédente est KO, alors la commande suivante est exécutée.

Si la variable « a » est vide alors le message sera affiché sinon un autre message sera affiché :

```
$ a=
$ [ -z "$a" ] && echo "La variable est vide" || echo "La variable n'est pas vide"
La variable est vide
                                                                    $
a=10
$ [ -z "$a" ] && echo "La variable est vide" || echo "La variable n'est pas vide"
La variable n'est pas vide
```

# 3. Test sur valeurs numériques

Les opérateurs de test sur des chaînes de caractères ne sont pas les mêmes sur les valeurs numériques.

| 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

La valeur de a est-elle différente de la valeur de b ?
$ test  "$a" -ne "$b" ; echo $?
0

La valeur de a est-elle supérieure ou égale à la valeur de b ?
$ test "$a" -ge "$b" ; echo $?
1

La valeur de a est-elle inférieure à la valeur de b ?
$ test "$a" -lt "$b" && echo "$a est inferieur a $b"
10 est inferieur a 20
```

# 4. Tests sur les fichiers

Nous avons vu jusqu'à présent des tests sur des variables contenant des chaines ou des valeurs numériques. Nous pouvons aussi opérer avec Bash des test sur des fichiers.

| 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 olivier   users       1392 Aug 14 15:55 dump.log
lrwxrwxrwx   1 olivier   users          4 Aug 14 15:21 lien_fic1 -> fic1
lrwxrwxrwx   1 olivier   users          4 Aug 14 15:21 lien_fic2 -> fic2
-rw-r--r--   1 olivier   users        234 Aug 16 12:20 liste1
-rw-r--r--   1 olivier   users        234 Aug 13 10:06 liste2
-rwxr--r--   1 olivier   users        288 Aug 19 09:05 param.sh
-rwxr--r--   1 olivier   users        430 Aug 19 09:09 param2.sh
-rwxr--r--   1 olivier   users        292 Aug 19 10:57 param3.sh
drwxr-xr-x   2 olivier   users       8192 Aug 19 12:09 rep01
-rw-r--r--   1 olivier   users       1496 Aug 14 16:12 resultat.txt
-rw-r--r--   1 olivier   users       1715 Aug 16 14:55 toto.txt
-rwxr--r--   1 olivier   users         12 Aug 16 12:07 voir_a.sh

      « lien_fic01 » est-il un fichier normal ?
$ test -f lien_fic1 ; echo $?
1 (signifie non)

« dump.log » est-il un fichier exécutable ?
$ test -x dump.log ; echo $?
1 (signifie non)

« rep01 » est-il un répertoire ?
$ test -d rep01 ; echo $?
0 (signifie oui)
```

# 5. Tests combinés

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

```
$ test -d "rep01" -a -w "rep01" && echo "rep01: répertoire, droit en écriture"
rep01: répertoire, droit en écriture
```

# D. Structure de contrôle conditionnel

## 1. Paramètres de position

Certaines variables ont une signification spéciale réservée. Ces variables sont très utilisée lors la création de scripts :

- pour récupérer les paramètres transmis sur la ligne de commande,
- pour savoir si une commande a échoué ou réussi,
- pour automatiser le traitement de tous paramètres.

| 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” .. |

**(voir exemple page 17)**

## 2. Structure conditionnelle « if »

Syntaxe :

```
if [ <commandes_condition> ]
then
     <commandes exécutées si conditions remplies>
else
     <commandes exécutées si conditions non remplies>
fi
```

**Exemple :**

Soit le script **param.sh** :

```
#!/bin/bash
if [ $# -ne 0 ]
then
  echo "$# paramètres en ligne de commande;"
else
  echo "Aucun paramètre;"
  echo "Passage de paramètres par défaut alfred oscar romeo zoulou;"
  set alfred oscar romeo zoulou
fi

echo "Nombre de paramètres : $#;"
echo "Paramètres : 1=$1 2=$2 3=$3 4=$4;"
echo "Liste : $*."
```

Rendre exécutable ce script puis le lancer en lui passant des paramètres :

```
$ ./param.sh toto titi tutu
3 paramètres en ligne de commande;
Nombre de paramètres : 3;
Paramètres : 1=toto 2=titi 3=tutu 4=;
Liste : toto titi tutu.
```

On relance le script sans passer de paramètres :

```
$ ./param.sh
Aucun paramètre;
Passage de paramètres par défaut alfred oscar romeo zoulou;
Nombre de paramètres : 4;
Paramètres : 1=alfred 2=oscar 3=romeo 4=zoulou;
Liste : alfred oscar romeo zoulou.
```

Il est bien sur possible d'imbriquer des **`if`** dans d'autres **`if`** et notamment des constructions telles que celle ci sont assez courantes :

```
if condition1               (si)
then                        (alors)
    instruction(s)
else                        (sinon)
    if condition2           (si)
    then                    (alors)
        instruction(s)
    else                    (sinon)
        if condition3       (si)
            ...
        fi
    fi
fi
```

Exemple :

**Script ifif.sh**

```
#!/bin/bash
if [ "$1" = Bonjour ]
then
   echo "Salut"
else
   if [ "$1" = Bye ]
   then
      echo "Au revoir"
   else
      echo "Vous n'êtes pas très poli"
   fi
fi
```

Lancez le script :

```
$ ./ifif.sh
Vous n'êtes pas très poli

$ ./ifif.sh Bye
Au revoir

$ ./ifif.sh Bonjour
Salut
```

Pour permettre d'alléger ce type de code, on peut utiliser : **elif** que l'on pourrait traduire par « sinon si ».

Le code précédent pourrait être réécrit ainsi :

```
if condition1    (si)
then             (alors)
    instruction(s)
elif condition2 (sinon si)
then             (alors)
    instruction(s)
elif condition3 (sinon si)
    ...
fi
```

Exemple :

**Script ifelif.sh**

```
#!/bin/bash
if [ "$1" = Bonjour ]
then
   echo "Salut"
elif [ "$1" = Bye ]
then
   echo "Au revoir"
else
   echo "Vous n'êtes pas très poli"
fi
```

# 3. Choix multiples « case »

Syntaxe :

```
case Valeur in
    valeur1) instructions ;;
    valeur2) instructions ;;
    valeur3) instructions ;;
    ...
esac
```

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

Cette commande permet d'améliorer la lisibilité contrairement à « **if** » quand il y beaucoup de conditions à vérifier.
La syntaxe ci-dessous avec « **if** » est quand même moins lisible que son équivalente avec « **case** » :

```
if [ valeur_testee = valeur1 ]
    then instruction(s)
elif [ valeur_testee = valeur2 ]
    then instruction(s)
elif [ valeur_testee = valeur3 ]
    then instruction(s)
...
fi
```

**Exemple**

Soit le script **case.sh** :

```
#!/bin/bash
if [ $# -eq 1 ]
then
  echo -n "$# paramètre passé en ligne de commande et "
else
  echo "Vous devez passer un seul paramètre: Bonjour ou Bye. (Ex.:
./case.sh param)"
  exit 10
fi

case $1 in
  Bonjour)echo "Salut" ;;
  Bye)echo "Au revoir" ;;
  *)echo "Seulement Bonjour ou Bye sont acceptés" ;;
esac
```

Le script vérifie dans un premier temps si un paramètre a été passé sinon le script s'arrête en affichant l'usage de la commande et en retournant le code de retour modifié de « 10 » au lieu de 1 (On peut modifier la valeur d'un code de retour et y mettre une valeur arbitraire).
Si la variable $1 (où $1 est le premier paramètre passé au script) contient « Bonjour » puis alors le script affichera « Salut ». Même logique pour « Bye »

Lancez le script :

```
$ ./case.sh
Usage: Vous devez passer un seul paramètre. (Ex.: ./case.sh param)

$ ./case.sh param param2
Usage: Vous devez passer un seul paramètre. (Ex.: ./case.sh param)


$ ./case.sh Bonjour
1 paramètre passé en ligne de commande et Salut

$ ./case.sh Bye
1 paramètre passé en ligne de commande et Au revoir

$ ./case.sh test
1 paramètre passé en ligne de commande et Seulement Bonjour ou Bye sont
acceptés
```

# E. Boucles conditionnelles

## 1. Boucle for

La boucle **for** permet de parcourir une liste de valeurs, elle effectue donc un nombre de tours de boucle qui est connu à l'avance.

Syntaxe :

```
for variable in liste_de_valeurs
do
  commandes à exécuter
done
```

### 1.1 Avec une variable

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

*Rappel : $@ affiche la liste des arguments passés*

On lance le script :

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

### 1.2 Liste implicite

Étant donné que les paramètres que nous passions au script précédent « for1.sh » était une liste nous aurions pu écrire le script de cette manière :

```
Script for1bis.sh
#!/bin/bash
for params
do
  echo "$params"
done
```

On lance le script :

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

On peut utiliser la boucle for pour programmer un clone de la commande « ls » :

**Script clonels.sh**
```
#!/bin/bash
for i in *
do
 echo $i
done
```

Dans cet exemple, l'étoile est remplacée par tous les fichiers du répertoire courant, la boucle for va donc donner successivement comme valeur à la variable « **i** » tous ces noms de fichier. Le corps de la boucle affichant la valeur de la variable « **i** », le nom de tous les fichiers du répertoire courant sont affichés successivement.

## 1.3 Avec une liste d'éléments explicite

```
$ script for2.sh
#!/usr/bin/sh
for params in fic01 fic02
do
        ls -l $params
done
```

On lance le script (les fichiers fic01 et fic02 doivent être créés auparavant)

```
$ ./for2.sh
-rw-r--r--   1 olivier   olivier       234 Aug 19 14:09 fic01
-rw-r--r--   1 olivier   olivier       234 Aug 13 10:06 fic02
```

## 1.4 Avec des critères de recherche sur nom de fichiers

***Script for3.sh***
```
#!/bin/bash
for params in *
do
 echo -n "$params "
 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
```

## 1.5 Avec un intervalle de valeurs

```
# La commande seq permet d'afficher une séquence de nombres
$ seq 5
1
2
3
4
5
$

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

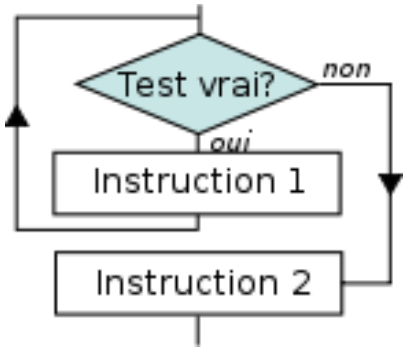
```
# Méthode consiste à utiliser une syntaxe proche du langage C :
$ for ((a=1 ; a<=5 ; a++)); do echo $a; done
1
2
3
4
5
$
```

## 1.6 Avec une substitution de commande

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

```
$ ./for4.sh
Liste des utilisateurs dans /etc/passwd
Utilisateur : root
Utilisateur : bin
Utilisateur : daemon
Utilisateur : adm
Utilisateur : lp
Utilisateur : sync
Utilisateur : shutdown
Utilisateur : halt
Utilisateur : mail
Utilisateur : operator
Utilisateur : games
Utilisateur : ftp
Utilisateur : nobody
...
```

# 2. Boucle while




Syntaxe :

```
while condition
    do instruction(s)
done
```

La boucle `while` exécute un bloc d'instructions tant qu'une certaine condition est satisfaite, lorsque cette condition devient fausse la boucle se termine. Cette boucle permet donc de faire un nombre indéterminé de tours de boucle, voire infini si la condition ne devient jamais fausse.

```
script while1.sh
#!/bin/bash
n=0
while [ $n -le 5 ]
do
   echo "${n}"

   n=$(($n+1))
   # Ou bien
   # let n=n+1
done
```

On peut traduire ce script par tant que la variable « n » est inférieure ou égale à 5, on incrémente la valeur de départ (zéro dans ce cas) de 1 à chaque itération et on sort de la boucle quand la valeur est supérieure à 5.

```
$ ./while1.sh
0
1
2
3
4
5
```

```
Script while2.sh
#!/bin/bash
while
  echo -n "Saisir une chaine : "
  read nom
  [ -z "$nom" ]
do
  echo "ERREUR : pas de chaine saisie"
done
echo "Vous avez saisi : $nom"
```

On lance le script :

```
$ ./while2.sh
Saisir une chaine :
ERREUR : pas de chaine saisie
Saisir une chaine :
ERREUR : pas de chaine saisie
Saisir une chaine : Bonjour
Vous avez saisi : Bonjour
```

Tant que la variable *$nom* est vide, on ne sort pas de la boucle.

Lecture d'un fichier ligne à ligne

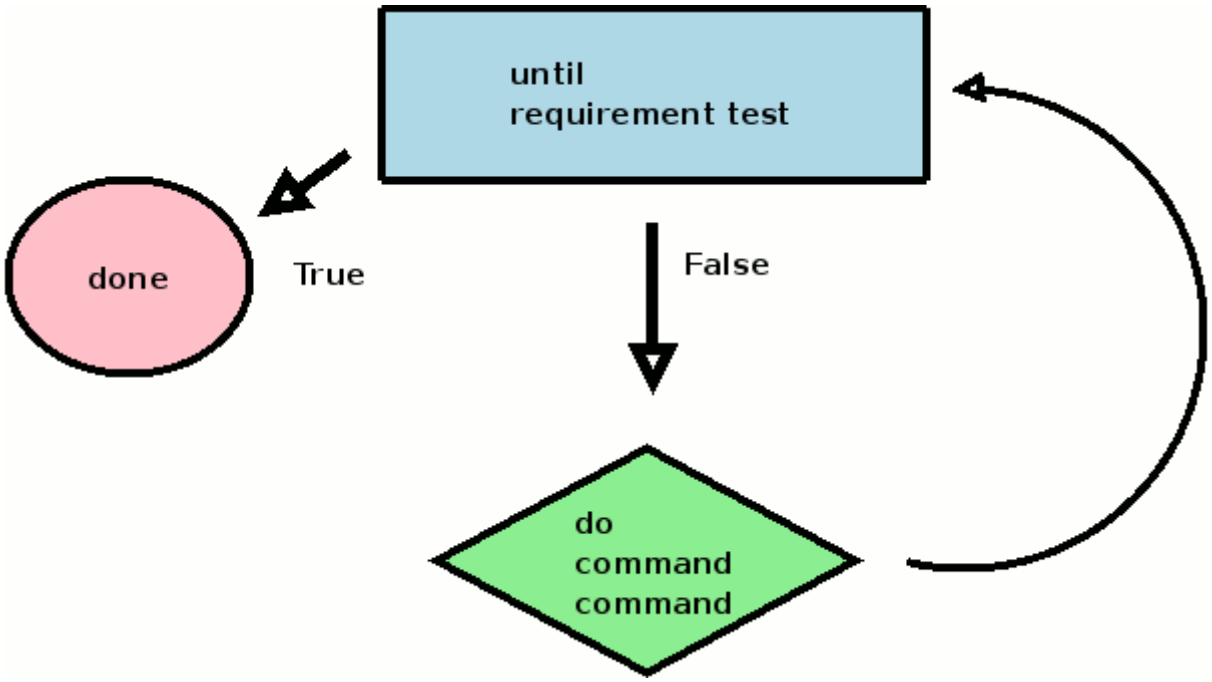
```
Script while3.sh
#!/bin/bash
cat /etc/passwd | while read line
do
  echo "$line"
done
```

ou bien :

```
Script while3.sh
#!/bin/bash
while read line
do
  echo "$line"
done < /etc/passwd
```

```
$./while3.sh
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
...
```

# 3. Boucle until




Syntaxe :

```
until condition
    do instruction(s)
done
```

On remarque que la syntaxe de la boucle `until` est exactement la même que celle de la boucle `while`, mais sa signification est inversée : la boucle est exécutée tant que la condition est fausse.

**Script until1.sh**

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
     echo COUNTER $COUNTER
     let COUNTER-=1
done
```

```
$ ./until1.sh
COUNTER 20
COUNTER 19
COUNTER 18
COUNTER 17
COUNTER 16
COUNTER 15
COUNTER 14
COUNTER 13
COUNTER 12
COUNTER 11
COUNTER 10
```

# 4. Boucle select

La commande **select** affiche un menu qui vous propose le choix parmi la liste définie ce qui permet de rendre les scripts plus intéractifs. Chaque ligne est numérotée en partant de 1.

Select affiche une phrase qui vous demande d'entrer votre choix. Cette phrase est contenu dans la variable PS3 qu'il faudra initialiser avant (par défaut, elle affiche "#?" si elle n'est pas initialisée).

La variable interne $REPLY (en majuscule) en s'appliquant au menu select, renvoie seulement le numéro de l'élément de la variable choisie et non pas la valeur de la variable elle-même.

### Exemple n°1 :

**script 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 "Numéro de réponse : $REPLY"
        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
Numéro de réponse : 1
Vous avez choisi Jules
Votre choix :2
Numéro de réponse : 2
Vous avez choisi Romain
Votre choix :3
Numéro de réponse : 3
Vous avez choisi Francois
Votre choix :4
Au revoir.
```

## Exemple n°2 :

```
$ Script menu.sh
#!/bin/bash
PS3="> selectionnez un plat : " # definie l'invite du menu
echo " -- menu du jour -- " # affiche un titre
select choix in cassoulet pizza "salade du chef" "quitter (q|Q)"
do
 case $REPLY in #on récupère le numéro de la réponse
  1) echo "Voici votre $choix."
     echo "Desirez-vous autre chose ?";;
  2) echo "Une pizza ? Excellent choix !"
     echo "Desirez-vous autre chose ?";;
  3) echo "Et une $choix, une !"
     echo "Desirez-vous autre chose ?";;
  4|Q*|q*) echo "Au revoir" # on quitte en appuyant sur 4, ou en tapant
un mot commençant par Q ou q
     break;;
  *) echo "Je n'ai pas compris votre commande. Veuillez répéter svp.";;
 esac
done
```

```
$ ./menu.sh
-- menu du jour --
1) cassoulet
2) pizza
3) salade du chef
4) quitter (q|Q)
> selectionnez un plat : azerty
Je n\'ai pas compris votre commande. Veuillez repeter svp.
1) cassoulet
2) pizza
3) salade du chef
4) quitter (q|Q)
> selectionnez un plat : 2
Une pizza ? Excellent choix !
Desirez-vous autre chose ?
1) cassoulet
2) pizza
3) salade du chef
4) quitter (q|Q)
> selectionnez un plat : q
Au revoir
```

# F. Fonction

## 1. Déclaration

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

La déclaration d'une fonction doit toujours se situer avant son appel.

## 2. Appel

```
nom_fonction param # ou sans param
```

### 2.1 Paramètres passés à la fonction

Ces paramètres sont optionnels. À l'intérieur de la fonction, ils sont représentés, respectivement, par les variables $1, $2,… , $n. $0 représente toujours le nom du 'script' (et non de la fonction) qui s'exécute.

Le nombre de paramètres passés à une fonction est représenté par la variable

**Exemple 1** :

```
Script fonc1.sh
#!/bin/bash

maFonction()
{ local varlocal="je suis la fonction"
  echo "$varlocal"
  echo "Nombres de paramètres : $#"
  echo $1
  echo $2
}
# Appel de la fonction
maFonction "Hello" "World!"
# La ligne dessous n'affichera qu'une ligne vide
echo "$varlocal"
```

Lancez le script :

```
$ ./fonc1.sh
je suis la fonction
Nombres de paramètres : 2
Hello
World!
```

**Exemple 2 :**

**Script search.sh**

```
#!/bin/bash
mycmd()
{
  find . -name "$1"
}

mycmd $1
exit 0
```

Lancez le script :

```
$ ./search.sh for01.sh
./for1.sh
```

# 3. Fin

Une fonction termine son exécution lorsqu'elle n'a plus d'instructions à exécuter ou lorsqu'elle rencontre l'instruction return ou exit. Ces instructions peuvent être suivies d'un entier positif, qui correspond à la valeur de retour de la fonction. Si aucune valeur n'est spécifiée, c'est la valeur 0 qui est renvoyée.

La valeur de retour de la dernière fonction appelée est stockée dans la variable $?

**Script fonc2.sh**

```
#!/bin/bash
print_something() {
echo "Hello $1"
return 5
}
print_something Mars
print_something Jupiter
echo "La fonction précédente a retourné un code retour : $?"
```

Lancez le script :

```
$ ./fonc2.sh
Hello Mars
Hello Jupiter
La fonction précédente a retourné un code retour : 5
```

# 4. Variables

Outre les paramètres, une fonction peut utiliser plusieurs variables:

- **Réutiliser toutes les variables globales du script**. Par défaut dans un script shell, les variables sont déclarées comme étant **globales**.
- **En déclarer de nouvelles**.
- **Déclarer des variables locales**. Pour déclarer une variable localement, il faut la faire précéder du mot clé « **local** ». Une telle variable ne sera utilisable que dans la fonction qui la déclare durant l'exécution de celle-ci mais sera considérée comme globale (donc connue) par les fonctions appelées à partir de cette fonction (après la déclaration locale de la variable).

Exemple :

**Script varscope.sh**

```
#!/bin/bash
# Variable locale et globale / Portée des variables

separateur () {
   echo "--------------------------------------------------------------"
}

var_change () {
  local var1='local 1'
  echo "Dans la fonction: var1 est $var1 : var2 est $var2"
  var1='Nouveau changement'
  var2='Valeur 2 changée par la fonction'
}

var1='global 1'
var2='global 2'

separateur
echo "Avant l'appel de la fonction: var1 est $var1 : var2 est $var2"
separateur
var_change
separateur
echo "Après l'appel de la fonction: var1 est toujours $var1 : var2 a
changé pour $var2"
separateur
```

Lancez le script :

```
$ ./varscope.sh
--------------------------------------------------------------
Avant l'appel de la fonction: var1 est global 1 : var2 est global 2
--------------------------------------------------------------
Dans la fonction: var1 est local 1 : var2 est global 2
--------------------------------------------------------------
Après l'appel de la fonction: var1 est toujours global 1 : var2 a changé
pour Valeur 2 changée par la fonction
--------------------------------------------------------------
```

# G. De la couleur dans le shell

## 1. Syntaxe

```
\033[XXmTexte à afficher\033[0m
```

La balise **\033[XXm** déclenche la mise en couleur du texte qui suit. **XX** est le code associé à une couleur.
La balise **\033[0m** ramène la coloration aux valeurs par défaut.

## 2. Les codes couleur et de mise en forme

Les coloris disponibles sont :

| Couleur | Code couleur police | Code couleur de fond |
|---|---|---|
| noir | 30 | 40 |
| rouge | 31 | 41 |
| vert | 32 | 42 |
| jaune | 33 | 43 |
| bleu | 34 | 44 |
| rose | 35 | 45 |
| cyan | 36 | 46 |
| gris | 37 | 47 |

```
$ echo -e '\033[31mUn texte en rouge !\033[0m Retour a la normale'
Un texte en rouge ! Retour a la normale
```

```
$ echo -e '\033[41mUn texte en rouge !\033[0m Retour a la normale'
Un texte en fond rouge ! Retour a la normale
```

On peut aussi jouer sur la mise en forme du texte :

| Code | Mise en forme |
|---|---|
| 0 | Retour aux valeurs par défaut |
| 1 | Texte en gras |
| 4 | Texte souligné |
| 5 | Texte clignotant |
| 7 | Texte surligné |

```
$ echo -e '\033[4mUn texte souligné !\033[0m Retour a la normale'
Un texte souligné ! Retour a la normale
```

On peut combiner les codes de couleur et de mise en forme :

```
$ echo -e '\033[32;1;5mUn texte en vert, en gras et clignotant  !\033[0m
Retour a la normale'
Un texte en vert, en gras et clignotant  ! Retour a la normale
```

Mais au fait pourquoi « echo **-e** » ? « **-e** » interprète les séquences de caractères précédées d'un backslash '\' . Plus d'infos : **man echo**