Chapitre 4 : Automatisation des travaux

4.1 Éléments d'un script shell

4.1.1 Introduction

Jusqu'à présent nous avons utilisé le shell en mode interactif, en tapant dans le terminal les lignes de commande successives que le shell doit traiter. Mais on peut également l'utiliser en mode script. Dans ce mode, la suite de lignes de commande est écrite dans un fichier texte qu'on appelle un script. Le nom de ce fichier devient alors une nouvelle commande dont le traitement va consister à déclencher successivement chacune des lignes de commande écrites dans le fichier texte. Dans cette partie, nous allons voir ce qu'est un script shell à l'aide d'un petit exemple d'illustration. Ensuite nous apprendrons comment utiliser des variables et les arguments des scripts afin de créer des traitements plus sophistiqués.

4.1.2 Hello World !

Voyons tout de suite un petit exemple illustratif afin de bien comprendre ce dont il s'agit. En informatique, la tradition veut que le premier programme écrit en apprentissage d'un nouveau langage de programmation soit le programme : Hello World. C'est un programme très simple qui se limite à afficher un message sur l'écran. Nous allons donc voir ici comment s'écrit un tel programme dans un script shell. Comme nous l'avons dit, un script est un simple fichier texte qui contient des commandes shell. Nous allons donc commencer par créer un tel fichier. Vous pouvez utiliser n'importe quel éditeur de texte. Par exemple l'éditeur **vi**[1] conviendra très bien. On va choisir de créer un nouveau fichier sous le nom **hello.sh** :

```
$ vi hello.sh
```

L'extension **.sh** n'est pas obligatoire, mais nous prendrons l'habitude de l'utiliser afin de pouvoir reconnaître facilement nos scripts. La première ligne que nous allons écrire dans ce fichier sera toujours la même :

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

On appelle cette ligne le ***shebang***. Ce nom vient de la contraction de ***sharp*** et de ***bang***, en référence aux deux premiers caractères de cette ligne : le dièse '**#**' (***sharp*** en anglais), et le point d'exclamation '**!**' (***bang***). Le *shebang* doit toujours être écrit au tout début du fichier script. Il permet d'indiquer au système l'interpréteur à utiliser pour les lignes qui suivent dans le reste du fichier. Dans notre cas, nous indiquons que le script doit être traité par le Bash.
Nous pouvons ensuite sauter une ou plusieurs lignes. Cela n'a pas d'incidence, mais ça rend le contenu plus lisible. Pour ce premier exemple, nous allons faire un script minimaliste qui ne va contenir que deux commandes d'affichage.

```
#!/bin/bash
echo "Hello"
echo "world !!!"
```

Voilà, on enregistre le fichier et on quitte l'éditeur de texte. À ce stade, il nous faut faire encore une dernière manipulation. Dans la mesure où nous allons devoir lancer l'exécution de ce script,

[1] Souvenez-vous qu'avec **vi** il faut commencer par appuyer sur la touche **i** pour passer en mode d'édition et sortir de ce mode par la touche **Esc** et de l'éditeur par **:wq** .

il nous faut au préalable nous assurer que ce fichier dispose bien du droit « exécuter ». Nous devons donc taper la commande :

```
$ chmod +x hello.sh
```

À partir de maintenant, nous n'aurons plus besoin de re-spécifier ce droit pour ce fichier, même si nous modifions son contenu à l'avenir. Aussi nous n'aurons plus besoin d'utiliser cette commande. Tout est donc en place. Nous pouvons maintenant tester notre script en lançant son exécution. Pour ce faire, il nous suffit de taper le nom du fichier dans la console, comme on le ferait pour une commande classique.

```
$ ./hello.sh
Hello
world !!!
$
```

Notez que nous avons fait précéder le nom du script par le chemin d'accès au script avec le symbole **'./'**, car le fichier script se trouve dans le répertoire courant. Si le script se trouve dans un autre répertoire, il faudra indiquer le chemin menant au fichier script pour que cela fonctionne. Il est possible de se passer de spécifier le chemin d'accès au répertoire courant, si le symbole point '.' se trouve dans la variable d'environnement PATH. Aussi, si vous souhaitez bénéficier de cet allègement d'écriture, vérifiez la présence du point dans la variable PATH et ajoutez-le s'il n'est pas présent.

4.1.3 Les variables dans un script

Un vrai script va généralement réaliser un traitement un peu plus complexe qu'un simple affichage ***Hello World*** . Et nous aurons souvent besoin de manipuler des variables durant ce traitement. La syntaxe de manipulation des variables est la même que celle du mode interactif. Et pour rappel, les variables shell ont les particularités suivantes :

• elles ne sont pas typées,
• elles n'ont pas besoin d'être déclarées,
• leur contenu par défaut est vide, c'est-à-dire la chaîne vide,
• pour accéder au contenu d'une variable, il faut préfixer son nom par le caractère '$'.

La différence avec le mode interactif est qu'ici les variables ne vont exister que pendant l'exécution du script. Elles seront en quelque sorte locales au script. Voyons cela en pratique. Nous allons éditer un nouveau fichier de nom **faye.sh** :

```
$ vi faye.sh
```

Dans ce script on va utiliser deux variables age et nom, et leur affecter les valeurs 65 et faye.

```
#!/bin/bash
age=65
nom=faye
```

Ici les règles syntaxiques sont exactement les mêmes que celles du mode interactif. L'affectation d'une variable doit se faire sans espace autour du caractère '='. La valeur à affecter peut être une chaîne de caractères ou un nombre. Si la chaîne contient des espaces, il faut supprimer la fonction de séparateur de mots de l'espace en utilisant une inhibition. Cette valeur peut aussi être obtenue par une des différentes constructions syntaxiques que vous avez étudiées dans les activités précédentes. Citons en particulier :

• pour le contenu d'une variable, la substitution de variable : $variable
• pour le code retour de la dernière commande exécutée : $?
• pour le résultat d'un calcul, la substitution arithmétique : $((calcul))
• pour le résultat d'une commande, la substitution de commande : $(commande)
Maintenant, dans notre exemple, nous allons exploiter les deux variables dans une commande d'affichage echo :

```
#!/bin/bash
age=65
nom=faye
echo "Monsieur $nom a $age ans"
```

Comme vous le savez, le caractère '**$**' va conduire à remplacer le nom de la variable par son contenu. La substitution de variable reste opèrante dans une inhibition partielle, à savoir entre deux guillemets. Une fois l'édition terminée, on enregistre, on quitte et on fixe le droit « exécutable » sur le fichier script :

```
$ chmod +x faye.sh
```

Maintenant si on lance l'exécution du script, on voit le texte s'afficher.

```
$ ./faye.sh
Monsieur faye a 65 ans
$
```

On peut constater qu'après l'exécution, les variables age et nom n'existent pas dans le shell :

```
$ ./faye.sh
Monsieur faye a 65 ans
$ echo $age
        (<- pas d'affichage)
$ echo $nom
        (<- pas d'affichage)
$
```

Cela s'explique par le fait que le script a été exécuté par un processus fils au shell. Nous reviendronssur ce phénomène de localité un peu plus loin.
Pour être complet, on peut rappeler que la substitution de variable s'écrit dans sa forme complète avec des accolades : ${variable}. La forme complète s'utilise quand le caractère qui suit le nom de la variable est un caractère alphanumérique ou le caractère souligné. Ainsi, si dans notre script faye.sh nous devons faire un pseudonyme composé du nom et de l'âge, séparés par la lettre *A* , nous allons avoir besoin d'utiliser la forme complète. Voyons ce que cela donne :

```
#!/bin/bash
age=65
nom=faye
pseudo1="$nomA$age"
echo $pseudo1 # -> 65
pseudo2="${nom}A$age"
echo $pseudo2 # -> fayeA65
```

Pour l'affectation de pseudo1, le nom de la première variable pose problème car il va être interprété comme le nom nomA au lieu du nom de variable nom suivi de la lettre *A* . L'affichage va ainsi donner 65 car nomA a la valeur vide. En ayant recours aux accolades comme pour pseudo2, le nom de la variable est isolé. Et dans ce cas, le bon résultat s'affiche : *fayeA65*.

```
$ ./faye.sh
65
fayeA65
$
```

Cet exemple montre aussi comment ajouter des commentaires dans le script. Il suffit d'utiliser le caractère '#', tout ce qui va être écrit après jusqu'au retour à la ligne sera ignoré par le shell. Ce qui est parfait pour y placer des remarques et des notes explicatives.

4.1.3.1 La portée des variables

Dans un script, on dit que les variables sont locales car les affectations de variables qu'on réalise ne vont pas se répercuter dans le contexte du processus qui a lancé l'exécution du script (i.e. le shell de votre terminal). Par exemple, si on exécute la dernière version du script henri.sh, puis si nous essayons d'afficher le contenu de la variable pseudo2 depuis le shell du terminal, rien ne va s'afficher :

```
$ ./faye.sh
65
fayeA65
$ echo $pseudo2

$
```

En effet, les commandes shell sont traitées dans un processus auquel est rattaché un ensemble de variables : on appelle cet ensemble un *contexte d'exécution*. Les commandes d'un script sont traitées dans un contexte d'exécution différent de celui du shell du terminal sur lequel il a été lancé. Et à la fin de l'exécution du script, son contexte d'exécution est supprimé. Ainsi, au niveau du terminal, les opérations d'affectation réalisées par le script sont donc sans effet.

4.1.3.2 « Sourcer » un script

Il existe un moyen de contourner ce comportement de localité des variables manipulées au sein d'un script. Il faut pour cela lancer l'exécution du script dans le même contexte d'exécution que le shell du terminal. Ceci se fait à l'aide de la commande source qui se note de manière abrégée par le caractère point '.' devant le nom du script à exécuter. On dit alors qu'on « source » le script.

```
$ source ./faye.sh # ou . ./faye.sh
65
fayeA65
$ echo $pseudo1
65
$
```

Si vous consultez les PID, par exemple avec la commande ps, vous remarquerez que le PID n'a pas changé lorsque le script est sourcé. Ce qui n'est pas le cas quand le script est lancé comme

une commande. Quand il n'est pas sourcé, le lancement du script crée un nouveau processus qui va se charger de réaliser l'exécution du code du script. Ce nouveau processus est ce que l'on a appelé un sous-shell dans les séquences précédentes.

4.1.3.3 Utiliser des variables externes

On peut aussi vouloir faire l'inverse. C'est-à-dire utiliser dans le script des variables qui sont fixées avant de lancer son exécution. Et là aussi il nous faut être vigilant car le contexte d'exécution du shell du terminal ne se transmet pas systématiquement au nouveau processus qui va être créé pour exécuter le script. Mettons cela en évidence en commentant l'affectation de la variable AGE dans notre script faye.sh :

```
#!/bin/bash
#AGE=65
NOM=faye
[...]
```

Si on affecte 30 à cette variable dans le shell du terminal, avant de lancer le script, cela n'aura aucun effet. L'affichage produit par le script ne va pas faire apparaître l'âge :

```
$ AGE=30
$ ./faye.sh

fayeA
$
```

Cela est une nouvelle fois dû au fait que le shell du terminal et le script sont exécutés dans deux processus différents et donc dans deux contextes d'exécution différents. On peut bien sûr s'en sortir en « sourçant » le script, mais s'il s'agit de ne transmettre qu'une seule variable il est préférable d'utiliser la commande export . Cette commande effectuée dans le terminal du shell indique que la variable en argument devra être copiée dans tous les contextes d'exécution des processus qui seront créés par la suite. Essayons cela sur la variable AGE :

```
$ export AGE=30
$ ./faye.sh
30
fayeA30
$
```

On a exporté cette variable en lui affectant 30, juste avant de lancer le script henri.sh. Et on remarque cette fois que le script a bien pris en compte le contenu de la variable du processus parent. Attention, cela ne veut pas dire que les modifications faites sur une variable exportée dans le processus d'un script seront répercutées au niveau du shell du terminal.

4.1.3.4 Résumé sur la portée des variables

Pour résumer, concernant la portée des variables dans un script, nous avons les trois situations suivantes :

• Le cas le plus courant où on lance simplement le script. Ce dernier dispose alors d'un contexte d'exécution qui lui est propre. Toutes ses variables sont locales.

• Le cas où on source un script. Le script partage alors le même contexte d'exécution que son shell de lancement. Toutes les variables sont partagées.

• Enfin, le cas où un export de une ou plusieurs variables a été fait avant de lancer le script. Les variables exportées sont copiées dans le contexte d'exécution qui sera créé lors du lancement du script. Seules ces variables conserveront leur contenu dans le script.

4.1.4 Les arguments d'un script : les variables positionnelles

Comme pour n'importe quelle commande, des arguments peuvent être passés à un script par la ligne de commande. Dans le code du script, ces arguments sont placés dans des variables particulières qui sont automatiquement renseignées au début de l'exécution du script. On qualifie ces variables de positionnelles. De plus, les variables positionnelles peuvent être renseignées lors d'un appel d'une fonction ou lors de la commande interne set. Les variables positionnelles ont la particularité d'être nommées par un numéro.

La syntaxe de consultation est **$*n*** où ***n*** est un nombre qui indique la position de l'argument à l'appel. Pour les variables positionnelles ayant plus de un chiffre, il faut avoir recours aux accolades **${*nn* }.**

En plus des variables positionnelles, des variables spéciales sont chargées pour décrire la liste des arguments. Ces variables spéciales ne sont pas utilisables avec l'affectation. Par conséquents, elles sont présentées sous leur nom d'utilisation à savoir avec leur nom précédé par le caractère '$'. Ces variables spéciales sont les suivantes :

• **$0** : le nom du script ;

• **$#** : le nombre d'arguments ;

• **$*** et **$@** : ces deux variables permettent de récupérer la liste de tous les arguments positionnels, mais pas de la même façon selon l'utilisation.

La différence entre les variables $* et $@ apparaît lorsqu'on les encadre de doubles quotes ("):

• sans double quote $* et $@ sont équivalents, on récupère la liste des mots obtenue en évaluant les arguments passés au script avec une espace entre chaque mot ;

• "$*" donne une chaîne de caractères résultant de la concaténation des valeurs des arguments en conservant les éventuelles espaces.

• "$@" donne la liste de toutes les valeurs des arguments passés au script **en conservant toutes les éventuelles espaces présentes à l'intérieur de ces arguments** (par exemple lorsque la valeur d'un argument est une suite de mots séparés par des espaces).

Illustrons ces éléments à l'aide des deux scripts qui suivent.

Le premier, nbArg.sh, contient le code suivant :

```
#!/bin/bash
echo "Il y a $# argument(s)"
echo "Le 1er est : $1"
```

Ainsi, ce script va afficher le nombre d'arguments qu'il reçoit lors de son appel, puis le contenu du premier d'entre eux. Si on le lance sans argument, il va afficher 0 comme nombre d'arguments, et rien pour la valeur du premier argument :

```
$ chmod +x nbArg.sh
$ ./nbArg.sh
Il y a 0 argument(s)
Le 1er est :
$
```

Et naturellement, si on passe plusieurs arguments, l'affichage produit sera cohérent :

```
$ ./nbArg.sh yop yep yap
Il y a 3 argument(s)
Le 1er est : yop
$
```

Le deuxième script, lesArg.sh illustre la différence entre les variables positionnelles $* et $@ :

```
#!/bin/bash
echo "Il y a $# arguments"
echo "Sans double quote \$* donne : " $*
echo "Sans double quote \$@ donne : " $@
echo "Avec double quote \$@ donne : " "$@"
echo "Avec double quote \$* donne : " "$*"
```

Ce script affiche ce qu'on obtient en évaluant $* et $@ encadrées ou pas avec les doubles quotes.

```
$ chmod +x lesArg.sh
$ ./lesArg.sh un " deux et trois" quatre
Il y a 3 arguments
Sans double quote $* donne : un deux et trois quatre
Sans double quote $@ donne : un deux et trois quatre
Avec double quote $@ donne : un deux et trois quatre
Avec double quote $* donne : un deux et trois quatre
$
```

Dans cet exemple d'exécution on fournit au script trois arguments : le premier et le troisième sont des mots simples, le deuxième est une chaîne formées de plusieurs mots et d'espaces.
Le script affiche d'abord le nombre d'arguments qui est bien égal à trois. On voit ensuite que les évaluations de $* et $@ fournissent le même résultat : une suite de mots avec un seul espace entre chaque mot.
Les deux dernières lignes nous montrent bien que l'évaluation entre double quote est différente : les espaces contenus dans le deuxième argument sont conservés. En revanche l'affichage ne permet pas de se rendre compte que "$@" est la suite des trois chaînes *"un"* , *" deux et trois"* et *"quatre"* , chacune séparée par une espace, alors que "$*" est l'unique chaîne *"un deux et trois quatre"* .
La différence est subtile : "$*" crée *une chaîne* contenant les arguments séparés chacun par une espace (ou plus exactement par le caractère contenu dans la variable d'environnement IFS qui est par défaut l'espace) alors que "$@" crée *la liste* des arguments.
Nous verrons une utilisation pratique de "$@" qui aborde les structures itératives du Bash.

4.1.4.1 Décalage des variables positionnelles

La commande shift décale vers la gauche les variables positionnelles. Par exemple, si dans un script la variable $1 vaut *a* et $2 vaut *b* , la commande shift va décaler le contenu de la variable positionnelle $2 sur la variable positionnelle $1. Le contenu de $1 qui était préalablement à la commande shift est perdu. Le nombre de variables positionnelles est décrémenté de un.
Pour illustrer cette commande, prenons le script decal.sh avec le code suivant :

```
#!/bin/bash
echo $# "$@"
```

```
shift
echo $# "$@"
```

L'exécution de ce script montre que le contenu du premier argument est perdu et qu'il n'y a plus que deux variables positionnelles.

```
$ decal.sh a b c
3 a b c
2 b c
$
```

La commande shift accepte un argument *n* qui doit être un nombre positif ou nul. Par défaut, shift suppose *n* à 1. Si *n* est donné, c'est un décalage de *n* positions vers la gauche qui est opéré : le contenu des *n* premières variables est perdu, $1 contient la valeur qui était avant l'opération dans la *n* + 1e variable positionnelle et le nombre de variables devient $# - n. Si *n* vaut 0 ou si *n* est supérieur à $# aucun décalage n'a lieu. Par exemple, la commande shift 2 supprime le contenu des variables $1 et $2 et décale le contenu des variables en commençant à $3. La variable $3 devient $1, $4 devient $2 et ainsi de suite.

La commande shift trouve son utilité quand il faut accéder au contenu d'un argument particulier dont l'argument est désigné par un nombre. Par exemple, nous voulons afficher le contenu de l'argument dont le numéro est donné par une variable. Si la variable a=5, la commande shift $((a-1)) rendra $1 avec le contenu demandé.

4.1.4.2 Affectation des variables positionnelles

Avant d'aller plus loin, revenons sur la commande interne set. Nous savons que la commande set exécutée sans option et sans argument affiche le nom et la valeur de chaque variable du shell. La commande interne set arg1 ... affecte ses arguments aux paramètres positionnelles. Si on passe des arguments à la commande set, ceux-ci sont affectés aux variables positionnelles en commençant au numéro un. Illustrons, l'affectation par la commande set et l'accès aux variables :

```
$ set 1 2 3 4 5 6 7 8 9 a b
$ echo $1 $9 ${10}
1 9 a
```

Plus précisément, les variables positionnelles sont re-initialisées avec les nouveaux arguments. L'exemple ci-dessous montre que les variables positionnelles initialisées par la première commande set ont été supprimées par la seconde commande set.

```
$ set a b c
$ echo $#
3$
set a
$ echo $#
1
```

La commande set accepte aussi des options. Pour éviter que le premier argument, s'il commence par un tiret, ne soit considéré comme une option et entraîne une erreur dans l'interprétation des arguments, on écrira les arguments après le double tiret '--'. Ce double tiret signifie en Bash la fin des options. Ce qui suit le double tiret est donc obligatoirement un argument même si l'argument commence par un tiret. La commande set -- sans argument supprime toutes les variables positionnelles.

```
$ set -- -a b
$ echo $@ $#
-a b 2
$ set --
$ echo $#
0
```

Classiquement les arguments d'une commande sont des mots séparés par l'espace. Le séparateur de mots est défini dans la variable d'environnement IFS (*Internal Field Separator*). En modifiant la valeur de l'IFS, la commande set peut servir à découper une chaîne de caractères en plusieurs mots et à les associer aux variables positionnelles.

```
$ ligne=ibou:toktok
$ IFS=':'
$ set -- $ligne
$ echo $1 $2
ibou toktok
```

4.1.4.3 Prudence vis-à-vis du contenu des variables

Quand des variables sont placées comme arguments d'un script ou comme arguments d'une commande, il y a une précaution à prendre pour éviter des erreurs lors de l'exécution du script. Par exemple, dans l'écriture echo \$var, le contenu de la variable var est passé comme argument de la commande echo. Dans cette situation, il est conseillé d'effectuer la substitution de variable dans une inhibition partielle (des guillemets) de la manière suivante echo "\$var". Car si la variable utilisée contient une chaîne de caractères comprenant des espaces, en l'absence de l'inhibition partielle, les espaces vont être traitées par le shell comme séparateur de mots et donc à l'appel de la commande chaque mot sera vu comme un argument supplémentaire, au lieu d'en former un seul à eux tous. Pour vous en convaincre, faisons l'expérience suivante :

```
$ TEXTE="Bonjour tous"
$ echo $TEXTE
Bonjour tous
$
```

A priori, le contenu de la variable TEXTE ne pose ici aucun problème. Mais si on utilise notre script nbArg.sh comme commande à la place de echo, regardons ce que cela produit :

```
$ TEXTE="Bonjour tous"
$ ./nbArg.sh $TEXTE              #le script nbArg
Il y a 2 argument(s)
Le 1er est : Bonjour
$
```

Nous constatons que le script affiche seulement le mot Bonjour et pas l'intégralité du contenu de la variable. L'espace a en effet séparé la chaîne de caractères en deux arguments. Pour inhiber les espaces, il faut utiliser l'inhibition partielle. C'est-à-dire encadrer la substitution de variable par des guillemets. Et cette fois l'affichage sera bien cohérent :

```
$ TEXTE="Bonjour tous"
$ ./nbArg.sh "$TEXTE"            #le script nbArg
```

```
Il y a 1 argument(s)
Le 1er est : Bonjour tous
$
```

Aussi, comme on ne sait pas par avance si le contenu d'une variable contient ou non des espaces, nous prendrons l'habitude de toujours encadrer la substitution de variable par des guillemets pour éviter toute mauvaise surprise.

4.1.4.4 Manipulation avancée des arguments

Pour mieux exploiter les arguments d'un script il est courant d'utiliser les syntaxes avancées de la substitution de variable. Cette syntaxe est utilisée principalement pour des scripts interactifs dans lesquels il est prudent d'avoir un contrôle sur la saisie de l'utilisateur.

Par exemple, pour éviter qu'une variable positionnelle ne retourne pas une valeur vide si elle ne correspond à aucun argument ; on peut spécifier une substitution de variable avec valeur par défaut.

La valeur par défaut est indiquée après le symbole ':-' et se note de la manière suivante :

```
${nom_variable:-defaut}
```

La valeur par défaut n'est pas affectée à la variable, elle est juste retournée à l'accès au contenu de la variable.

Pour contrôler la présence d'un contenu associé à une variable, il y a la syntaxe :

```
${nom_variable:?message}
```

Cette substitution de variable indique que la valeur de la variable doit être retournée si elle existe, ou sinon que le message d'erreur indiqué après le caractère '?' est affiché et l'exécution du script est interrompue. Cette syntaxe trouve son utilité pour vérifier la présence d'un argument obligatoire.

Nous allons respectivement voir leur usage dans deux petits scripts. Nous allons commencer par une amélioration de notre script nbArg.sh. On souhaite cette fois afficher le mot « vide » si le premier argument n'est pas présent. On va donc utiliser la syntaxe qui permet d'affecter une valeur par défaut en cas d'absence du premier argument.

```
#!/bin/bash
echo "Il y a $# argument(s)"
FIRST=${1 :-vide}
echo "Le 1er est : $FIRST"
```

Si on lance l'exécution du script, l'affichage produit est maintenant plus parlant :

```
$ ./nbArg.sh
Il y a 0 argument(s)
Le 1er est : vide
$
```

Pour illustrer la deuxième syntaxe, nous allons créer un nouveau script arg1.sh, qui exige au moins un argument à son lancement. Ici il nous faut utiliser la syntaxe qui va contrôler l'existence du premier argument. Dans celle-ci on va renseigner la phrase d'erreur à afficher en cas d'absence de cet argument.

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

```
ARG1=${1:?"Vous devez fournir un argument"}
echo "Le 1er est : $ARG1"
```

Si on lance ce script sans argument, ce message d'erreur va s'afficher :

```
$ chmod +x arg1.sh
$ ./arg1.sh
./arg1.sh line 3: 1: Vous devez fournir un argument
```

Notez que l'exécution du script s'arrête après l'affichage du message d'erreur (la commande suivante (echo) et tout ce qui suit dans le script est ignoré).

4.1.5 Le code retour d'un script

Quand une commande termine son exécution, elle renvoie un code retour (voir l'activité 2.4) pour indiquer au processus parent (celui qui a créé le processus de la commande), si l'exécution s'est bien passée (avec la valeur 0), ou si une erreur s'est produite (avec une valeur entre 1 et 255). C'est la même chose pour un script. Le code retour retourné par un script est :

• Soit celui spécifié par la commande : exit *code* , qui va mettre explicitement fin à l'exécution du script en renvoyant le code retour *code* . L'argument *code* est optionnel, et vaut 0 par défaut.

• Soit celui renvoyé par la dernière commande exécutée dans le code du script.

Le code retour se récupère par l'appelant du script en utilisant $?.

Voici un petit exemple de consultation de ce code retour avec le script arg1.sh :

```
$ cat arg1.sh              # Affichage du contenu du script arg1
#!/bin/bash
ARG1=${1:?"Vous devez fournir un argument"}
echo "Le 1er est : $ARG1"
exit 0
$ ./arg1.sh coucou         # appel le script arg1
Le 1er est : coucou
$ echo $?
0$
```

Quand un argument est passé à ce script, son code retour prend la valeur 0. Cette valeur est obtenue par la commande exit située à la fin du script. La commande exit est ici facultative. En effet, le même code retour aura été obtenu car la dernière commande qui s'est déroulée sans erreur à la commande echo. Le code retour de cette commande est le code retour du script. A l'inverse, si le script est lancé sans argument, l'affectation de ARG1 ne peut se faire. L'exécution du script s'arrête et le code retour prend la valeur 1 signifiant la présence d'une erreur :

```
$ ./arg1.sh
./arg1.sh: line 3: 1: Vous devez fournir un argument
$ echo $?
1
```

La valeur de $? est celle retournée par la dernière commande traitée dans arg1.sh, qui est cette fois la commande d'affectation de la variable ARG1. Dans cette commande, la syntaxe utilisée vérifie la présence du premier argument :

```
ARG1=${1:?"Vous devez fournir un argument"}
```

4.2 Expressions et conditions

4.2.1 Introduction

Comme tout langage de programmation, le Bash fournit des structures de contrôle permettant de prendre des décisions lors de l'exécution d'un script. Ces décisions sont prises à l'aide de structures de contrôle et de la réalisation de tests.

Toutes les commandes d'un système Unix ont un code retour égal à 0 si la commande s'est terminée normalement et entre 1 et 255 sinon. En shell Bash on peut donc voir toute commande comme une expression conditionnelle dont le code retour est assimilé à vrai s'il vaut 0 et à faux dans les autres cas.

4.2.2 La commande test

Supposons qu'on veuille tester si la valeur d'une variable numérique a est plus grande que 10. Avec la représentation mathématique usuelle, cette condition s'écrit $a > 10$. Le shell propose deux écritures pour effectuer ce test :

• soit on écrit la commande test suivie de l'expression à tester :

```
test $a -gt 10
```

• soit on écrit l'expression à tester entre crochets :

```
[ $a -gt 10 ]
```

Attention :

L'espace suivant le crochet ouvrant et l'espace précédant le crochet fermant sont obligatoires ! Il en est de même des espaces autour des opérateurs utilisés dans l'expression à tester.

Ces deux écritures sont équivalentes, elles permettent de tester la valeur d'une expression conditionnelle et renvoient comme code retour 0 si le test est vrai et 1 si le test est faux.

4.2.2.1 Opérateurs de comparaison arithmétique

La commande test dispose des principaux opérateurs de comparaison arithmétique sur les entiers. Ils sont introduits sous la forme d'options de commande, c'est-à-dire par un tiret, et suivies de deux lettres qui sont des abréviations de leurs noms en anglais.

Voici la liste de ces opérateurs ; ils s'utilisent avec des nombres entiers ou des variables numériques entières :

test *n1* **-eq** *n2*
: vrai si le nombre *n1* est égal au nombre *n2*

test *n1* **-ne** *n2*
: vrai si le nombre *n1* est différent du nombre *n2*

test *n1* **-gt** *n2*
: vrai si le nombre *n1* est strictement supérieur (*greater than*) au nombre *n2*

test *n1* **-lt** *n2*
: vrai si le nombre *n1* est strictement inférieur (*less than*) au nombre *n2*

test *n1* **-ge** *n2*
: vrai si le nombre *n1* est supérieur ou égal (*greater or equal*) au nombre *n2*

test *n1* **-le** *n2*
: vrai si le nombre *n1* est inférieur ou égal (*less or equal*) au nombre *n2*

Dans les exemples qui suivent et qui sont réalisés en ligne de commande, on visualise le résultat de la commande test en affichant son code retour qui est contenu dans la variable $?

Répétons-le encore une fois : un code retour 1 signifie que le test est faux, 0 que le test est vrai.

```
$ a=3
$ test $a -gt 12; echo $?
1$
test $a -lt 0; echo $?
1$
test $a -ge 2; echo $?
0$
[ $a -ne 3 ]; echo $?
1$
[ 3 -eq $a ]; echo $?
0
```

4.2.2.2 Opérateurs sur les chaînes de caractères

Pour effectuer des comparaisons sur des chaînes de caractères, nous disposons des opérateurs suivants :

test **-n** *chaine*
: vrai si *chaine* est une chaîne de caractères non vide

test **-z** *chaine*
: vrai si *chaine* est une chaîne de caractères vide

test *chaine1* = *chaine2*
: vrai si *chaine1* est identique à *chaine2*

test *chaine1* == *chaine2*
: vrai si *chaine1* est identique à *chaine2*

test *chaine1* != *chaine2*
: vrai si *chaine1* est différente de *chaine2*

test *chaine1* < *chaine2*
: vrai si *chaine1* est inférieure à *chaine2*

test *chaine1* > *chaine2*
: vrai si *chaine1* est supérieure à *chaine2*

Attention '<' et '>' sont des caractères spéciaux du shell. Ces caractères redirigent l'entrée et la sortie standard d'une commande. Pour ne pas avoir de surprise, il faut donc bloquer leur comportement par défaut (en les précédant par exemple du caractère d'inhibition de caractère '\').

À propos de la comparaison de chaînes de caractères

La comparaison de deux chaînes de caractères s'effectue d'une manière analogue à la comparaison de deux mots dans un dictionnaire : on peut dire qu'un mot est inférieur à un autre s'il arrive avant dans l'ordre alphabétique du dictionnaire.

Comme les chaînes de caractères ne sont pas constituées uniquement de lettres mais de n'importe quels caractères, la comparaison s'effectue suivant l'ordre dans lequel les caractères apparaissent dans le codage ASCII (*American Standard Code for Information Interchange*).

Pour comprendre comment sont comparées les chaînes, retenez que dans le code ASCII :

• les lettres de l'alphabet sont rangées dans l'ordre habituel ;

• les lettres majuscules et minuscules sont différenciées et les lettre majuscules (codes 65 à 90) sont rangées avant les lettres minuscules (code 97 à 122), elles sont donc « inférieures » aux lettres minuscules ;

• les caractères représentant les chiffres (codes 48 à 57) sont aussi rangés dans l'ordre et avant les lettres, ils sont donc « inférieurs » aux lettres.

Voici quelques exemples pour illustrer la comparaison des chaînes de caractères :

```
$ mot1=ibou
$ echo $mot1
ibou
$ [ -n $mot1 ]; echo $?
0
```

Comme le code retour du test est vrai (0), la chaîne contenue dans la variable mot1 n'est effectivement pas vide.

```
$ [ $mot1 = IBOU ]; echo $?
1
```

Le code retour 1 indique que le test est faux. La chaîne contenue dans la variable mot1, c'est-à-dire la chaîne *ibou*, n'est pas égale à la chaîne *IBOU*. En effet, les lettres minuscules et majuscules sont distinctes.

```
$ [ $mot1 \> Ibou ]; echo $?
0
```

Le code retour 0 indique que le test est vrai, c'est à dire que la chaîne *ibou* est supérieure à la chaîne *Ibou*. En effet, dans le codage ASCII le caractère 'a' vient après le caractère 'A' ; autrement dit 'a' est supérieur à 'A'.

```
$ [ bay \> ibou ]; echo $?
0
```

Le code retour 0 indique que la chaîne *bay* est supérieure à la chaîne *ibou* . On voit dans cet exemple que ce n'est pas le nombre de caractères qui compte, c'est l'ordre ASCII des caractères, et comme le caractère 'b' vient après le caractère 'a', la chaîne *bay* est bien supérieure à la chaîne *ibou*.

Attention à ne pas confondre chaîne de caractères et nombre.

Les opérateurs '<' et '>' peuvent être source d'erreurs si vous les utilisez avec des nombres car ce ne sont pas les valeurs numériques qui seront comparées mais des chaînes de caractères constituées de caractères chiffres.

Par exemple, le test suivant dont le résultat est vrai est trompeur car il ne compare pas les valeurs arithmétiques 32 et 20 mais les chaînes de caractères *32* et *20* et comme le caractère '3' vient bien après le caractère '2' dans le codage ASCII, la réponse est vrai.

```
$ [ 32 \> 20 ]; echo $?
0
```

On se rend mieux compte du problème avec le test suivant qui peut paraître contre-intuitif.

```
$ [ 32 \> 200 ]; echo $?
0
```

Le code retour 0 indique que le test est vrai. En effet, ce sont les chaînes de caractères *32* et *200* qui sont comparées lorsqu'on utilise l'opérateur '>', non les nombres 32 et 200. Et comme le caractère

’3’ vient après le caractère ’2’ dans le codage ASCII, la chaîne *32* est effectivement supérieure à la chaîne *200*. Pour effectuer le test sur les nombres 32 et 200 il faut utiliser l’opérateur de comparaison arithmétique -gt vu plus haut.

4.2.2.3 Opérateurs sur les fichiers

La commande test fournit beaucoup d’opérateurs pour réaliser des tests sur les fichiers. En voici quelques-uns parmi les plus couramment utilisés ; pour en avoir la liste exhaustive, le lecteur est encouragé à utiliser la commande man.

test **-s** *fichier*
vrai si *fichier* n’est pas vide
test **-f** *fichier*
vrai si *fichier* existe et est un fichier ordinaire
test **-d** *fichier*
vrai si *fichier* existe et est un répertoire
test **-e** *fichier*
vrai si *fichier* existe, quel que soit son type
test **-r** *fichier*
vrai si *fichier* existe et est accessible en lecture
test **-w** *fichier*
vrai si *fichier* existe et est accessible en écriture
test **-x** *fichier*
vrai si *fichier* existe et possède le droit exécutable
test *fichier1* **-nt** *fichier2*
vrai si *fichier1* et *fichier2* existent et que *fichier1* est plus récent que *fichier2*
test *fichier1* **-ot** *fichier2*
vrai si *fichier1* et *fichier2* existent et que *fichier1* est plus ancien que *fichier2*
Illustrons ces opérateurs par quelques exemples :

```
$ > monfichier
$ [ -s monfichier ]; echo $?
1
```

Juste après avoir créé ou écrasé un fichier monfichier avec la commande > monfichier, le code retour 1, c’est à dire faux, du test effectué indique que le fichier est vide (-s est vrai si le fichier est non vide), ce qu’on peut vérifier en listant les propriétés de monfichier.

```
$ ls -l monfichier
-rw-r--r-- 1 ibou user 0 22 sep 11:25 monfichier
$
```

Ajoutons maintenant du contenu dans monfichier et refaisons le test.

```
$ echo Bonjour le monde >> monfichier
$ [ -s monfichier ]; echo $?
0
```

Une fois qu’on a écrit dans monfichier, il n’est évidement plus vide !
Créons maintenant un répertoire monrep afin de tester les types de fichiers.

```
$ mkdir monrep
$ [ -f monrep ]; echo $?
1$
[ -f monfichier ]; echo $?
0$
[ -d monrep ]; echo $?
0
```

On a vérifié que monrep n'est pas un fichier ordinaire, contrairement à monfichier et que c'est bien un répertoire.

```
$ [ -w monfichier ]; echo $?
0
```

Si on a pu écrire dans monfichier, c'est parce que celui-ci est accessible en écriture, ce que le test ci-dessus permet de vérifier.

```
$ [ monrep -nt monfichier ]; echo $?
0
```

Le répertoire monrep a été créé après le fichier monfichier, il est effectivement plus récent.

```
$ [ -e zorglub ]; echo $?
1
```

4.2.2.4 Opérateurs logiques

Naturellement il est possible de combiner les expressions des tests vus dans les paragraphes précédents à l'aide d'opérateurs logiques :
! est l'opérateur logique de négation, le NON
\(\) est l'opérateur de regroupement
-a est l'opérateur logique binaire ET
-o est l'opérateur logique binaire OU
Ces opérateurs sont présentés selon un ordre de priorité descendant :
! est l'opérateur le plus prioritaire,
-o est l'opérateur le moins prioritaire
Voyons à travers quelques exemples l'utilisation des opérateurs logiques pour réaliser des tests complexes.
Imaginons une situation où nous devons vérifier que si un fichier de nom confidentiel existe, alors celui-ci ne doit pas être accessible en lecture.
Avec l'opérateur de négation nous pouvons vérifier que le fichier n'est pas accessible en lecture :

```
$ [ ! -r confidentiel ]; echo $?
0
```

Le code retour 0 signifie que le test est vrai, c'est-à-dire que le fichier n'est pas accessible en lecture. Cependant ce test ne permet pas de savoir si le fichier existe. En effet, s'il n'existe pas, il n'est pas accessible en lecture.
Essayons de compléter le test avec l'opérateur logique -a pour vérifier que le fichier existe et qu'il n'est pas accessible en lecture :

```
$ [ -e confidentiel -a  !  -r confidentiel ]; echo $?
1
```

Le code retour 1 indique que le test est faux. Une formule logique *A ET B* est fausse lorsque soit *A* est faux, soit *B* est faux. Dans notre cas, le résultat faux du test peut donc vouloir dire que :

• le fichier confidentiel n'existe pas

• le fichier confidentiel existe mais il est accessible en lecture

Pour en avoir le coeur net, on peut formuler le test « le fichier confidentiel existe ET n'est pas accessible en lecture OU BIEN il n'existe pas », ce qui se traduit par la commande :

```
$ [ -e confidential  -a  !  -r confidential  -o  !  -e  confidentiel ]; echo $?
0
```

Si vous avez un doute sur l'ordre de priorité, vous pouvez utilisez les parenthèses de la façon suivante :

```
$ [ \( -e confidentiel -a ! -r confidentiel \) -o ! -e confidentiel ]; echo $?
0
```

4.2.3 Effectuer des tests numériques avec la commande let

La commande let qui s'abrège par l'écriture ((***expression***)) permet de réaliser des calculs avec des expressions arithmétiques et en particulier des tests arithmétiques. Cette commande n'affiche rien mais son code retour peut être utilisé pour réaliser des tests. Son intérêt réside dans l'utilisation de la syntaxe du langage C pour exprimer l'expression.

Dans le cas de tests arithmétiques, si l'expression entre les doubles parenthèses est vraie le code retour vaut 0 et il vaut 1 sinon. Ce qui est équivalent aux résultats obtenus avec la commande test.

```
$ a=2
$ b=4
$ [ $a -lt $b ]; echo $?
0
$(( a < b )); echo $?
0
$[ $a -eq $b ]; echo $?
1
$(( a == b )); echo $?
1
```

Attention cependant si l'expression arithmétique n'est pas une expression conditionnelle, l'interprétation du code retour est délicate. En effet, le code retour obtenu est 0 (vrai) si la valeur de l'expression entre les doubles parenthèses donne une valeur différente de 0 et 1 (faux) si la valeur de l'expression vaut 0.

Ainsi le résultat d'une expression arithmétique non conditionnelle dont on utilise le code retour est équivalent à faux si la valeur de l'expression est 0 et vrai autrement. **Il est donc important de ne pas confondre le code retour et la valeur de l'expression entre parenthèses**.

```
$ a=4
$ (( a-4 )); echo $?
1
$(( a-2 )); echo $?
0
$(( a/10 )); echo $?
1
$(( a/2 )); echo $?
0
```

La valeur de l'expression arithmétique $a - 4$ est 0 donc le code retour de ((a-4)) est 1, ce qui, utilisé comme un test, est équivalent à faux. L'interprétation serait donc « a est-il différent de 4 ? ». De la même manière l'interprétation du code retour de ((a-2)) serait « a est-il différent de 2 ? » ; et pour ((a/10)) : « la division entière de a par 10 est-elle différente de 0 ? » ou autrement dit « a est-il supérieur ou égal à 10 ? »

4.2.4 Type booléen

Comme nous venons de le voir, l'évaluation d'une expression donne un code retour qui peut avoir deux significations : vraie ou fausse. On qualifie de booléen une donnée qui ne peut avoir que deux valeurs. S'il existe des opérateurs booléens dans le shell, il n'y a pas de type de données booléennes. Cependant, les variables peuvent contenir true et false. true et false sont des commandes internes qui retourne le code retour associé à leur nom. Il est alors possible « d'émuler » le type booléen pour des variables avec ces deux commandes. Pour illustrer ceci, prenons l'exemple ci-dessous :

```
$ a=true
$ $a ; echo $?
0
```

Cet exemple montre que la phase d'interprétation de la seconde ligne de commande donne true et son exécution retourne le code retour 0 qui est affiché par la commande echo. On peut considérer que la variable a est comme de type booléen.

4.2.5 Récapitulatif

Les expressions sont évaluées par la commande test. Il existe une variante syntaxique de cette commande qui s'écrit avec une paire de crochets '[ ]' . Le tableau 1 récapitule les opérateurs courants. La liste complète est disponible dans le **man bash** ou plus directement par le **help test**.

Attention de bien **laisser une espace** après le crochet ouvrant et avant le crochet fermant !

Opérateur	Sémantique	Syntaxe
Opérateurs de logiques		
`-a`	et logique (and)	*expr1* `-a` *expr2*
`-o`	ou logique (or)	*expr1* `-o` *expr2*
`!`	négation logique (not)	`!` *expr*
Pour les valeurs numériques		
`-eq`	égale à (equal)	*expr1* `-eq` *expr2*
`-ne`	différente de (not equal)	*expr1* `-ne` *expr2*
`-lt`	inférieure à (less than)	*expr1* `-lt` *expr2*
`-gt`	supérieure à (greater than)	*expr1* `-gt` *expr2*
`-le`	inférieure ou égale à	*expr1* `-le` *expr2*
`-ge`	supérieure ou égale à	*expr1* `-ge` *expr2*
Pour les chaînes de caractères		
`=`	égale à	*chaîne1* `=` *chaîne2*
`\<`	inférieure à	*chaîne1* `\<` *chaîne2*
`\>`	supérieure à	*chaîne1* `\>` *chaîne2*
`!=`	différente de (not equal)	*chaîne1* `!=` *chaîne2*
`-z`	la chaîne a une taille égale à zéro	`-z` *chaîne*
`-n`	la chaîne est de longueur non zéro	`-n` *chaîne*
Pour les fichiers		
`-e`	le fichier existe	`-e` *fichier*
`-r`	le fichier est lisible	`-r` *fichier*
`-w`	le fichier est modifiable	`-w` *fichier*
`-x`	le fichier est exécutable	`-x` *fichier*
`-f`	le fichier est ordinaire	`-f` *fichier*
`-d`	le fichier est un répertoire	`-d` *fichier*
`-s`	le fichier est non vide	`-s` *fichier*

Tableau 1 : Operateurs

La commande de test étendu notée par une double paires de crochets '[[ ]]' reprend les opérateurs de la commande test (à l'exception de -a et -o). Les extensions de cette commande sont rappelées par le tableau 2.

Opérateur	Sémantique	Syntaxe
`&&`	et logique (and)	*expr1* `&&` *expr2*
`||`	ou logique (or)	*expr1* `||` *expr2*
`( )`	regroupement	`(` *expr* `)`
`==`	correspondance	*chaîne* `==` *pattern*
`=~`	correspondance	*chaîne* `=~` *regexp*

Tableau 2 : Operateurs pour testes étendus

4.3 Structures conditionnelles

4.3.1 Introduction

Prendre des décisions, conditionner l'exécution d'une commande au résultat d'une autre sont des éléments indispensables dès lors qu'on souhaite programmer des tâches un peu complexes. Nous avons vu comment effectuer des tests sur des chaînes de caractères, sur des nombres et sur des fichiers. Ces tests sont effectués à l'aide de commandes et c'est leur code retour qui indique si le test est vrai ou faux.

Nous allons voir comment utiliser le code retour d'une commande pour conditionner l'exécution d'autres commandes. Il s'agit de voir les structures de contrôle conditionnel du Bash : l'enchaînement conditionnel de commandes, la commande if et la commande case.

4.3.2 Enchaînement conditionnel

L'enchaînement conditionnel de commandes consiste à conditionner l'exécution d'une commande au résultat d'une autre. Selon le code retour d'une première expression, une autre sera ou ne sera pas exécutée.

Il existe trois caractères spéciaux permettant l'enchaînement de commande : le point-virgule, la double barre verticale et la double esperluette (aussi appelé « et commercial »).

4.3.2.1 Le point-virgule

commande1 ; *commande2*

Le point-virgule est un séparateur de commande et il permet d'exécuter plusieurs commandes sur une même ligne de commande. Ces commandes sont exécutées les unes après les autres sans se soucier de la réussite ou de l'échec de chacune d'elles. Cet enchaînement n'est à utiliser que lorsqu'on est sûr de la réussite de chacune des commandes ou bien lorsque leur éventuel échec n'a aucune incidence sur la suite.

On peut le voir comme un moyen de regrouper plusieurs lignes de commandes sur une même ligne de commande. Par exemple, si on veut créer un répertoire et l'utiliser comme répertoire courant pour y créer un fichier vide, on pourra écrire :

```
$ mkdir ~/repertoire1; cd ~/repertoire1; touch fichiervide
```

Cette ligne de commande peut également s'écrire en 3 lignes de commandes.

4.3.2.2 La double esperluette

commande1 && *commande2*

Avec cet enchaînement la *commande2* n'est exécutée que si le code retour de la *commande1* est 0. Autrement dit si la *commande1* réussit.

Par exemple on peut tester l'existence d'un fichier avant d'en afficher le contenu :

```
$ [  -f monCV  ]  &&  cat monCV
```

Si le fichier monCV existe la commande [-f monCV] a comme code retour 0, soit « vrai », et cat monCV sera exécutée mais si le fichier n'existe pas elle sera ignorée.

4.3.2.3 La double barre vertical

commande1 | | *commande2*

Cette fois la *commande2* ne sera exécutée que si la *commande1* échoue ; par exemple si c'est un test dont le résultat est « faux ».

On peut compléter l'exemple précédent pour qu'un message s'affiche sur la sortie d'erreur au cas où le fichier monCV n'existe pas :

```
$ [  -f  monCV  ]  &&  cat monCV || echo "le fichier monCV n'existe pas"  > &2
```

Attention les commandes enchaînées sont exécutées dans le sens d'écriture, de gauche à droite sans qu'il y ait de priorité. Le code retour d'un enchaînement est celui de la dernière commande exécutée.

Ainsi dans l'exemple précédent :

• si monCV existe, le résultat de [-f monCV] est 0 ce qui provoque l'exécution de la commande

cat monCV. Cette dernière réussit, son code retour est 0 et par conséquent la commande echo est ignorée

• si monCV n'existe pas, le résultat de [-f monCV] est 1, la commande cat monCV ne s'exécute pas, le résultat de l'enchaînement [-f monCV] && cat monCV est le code retour de la seule commande qui a été exécutée : [-f monCV], c'est-à-dire 1. Par conséquent la commande echo qui suit | | sera exécutée et le message s'affichera.

4.3.3 La structure de contrôle conditionnel if

L'enchaînement de commandes est pratique mais se limite à l'exécution conditionnelle de commandes simples et peut vite devenir illisible si plusieurs commandes s'enchaînent. S'il y a plusieurs commandes à exécuter pour une même condition, il est déconseillé alors d'utiliser l'enchaînement de commandes.

La commande if répond à ce besoin d'une écriture plus lisible et de proposer des blocs de commandes dont l'exécution est liée au résultat d'une condition. Avec cette commande, des contrôles plus élaborés peuvent être construits.

4.3.3.1 Condition simple

Dans sa version la plus simple la structure conditionnelle permet d'effectuer une opération si une condition est réalisée, la syntaxe est la suivante :

```
if    condition
then
     lignes-commandes-si-vrai
fi
```

Attention : pour éviter toute erreur dans l'écriture de la structure il est nécessaire de respecter les passages à la ligne. On écrit le mot clé if suivi de la condition sur une seule ligne puis on passe à la ligne, on écrit le mot clé then seul sur une ligne, on passe de nouveau à la ligne pour écrire les lignes de commandes à exécuter lorsque la condition est vraie, puis on termine la structure en écrivant le mot clé fi seul sur une ligne.

L'interprétation de cette structure de contrôle est la suivante : si, et seulement si, la *condition* est vérifiée, alors toutes les commandes entre le **if** et le **fi** seront exécutées.

Pour que la condition soit vérifiée il faut que le code retour de celle-ci soit 0 (interprété comme vrai). Le plus souvent la condition est exprimée à l'aide de la commande test mais cela peut aussi être n'importe quelle commande sachant que c'est son code retour qui sera utilisé pour décider de l'exécution des *lignes-commandes-si-vrai*.

Exemple

À titre d'exemple on peut tester la valeur d'une variable avant d'effectuer un calcul :

```
if  test  "$mavariable" -ne  0
then
         ;resultat=$((10 / mavariable))
fi
```

Qu'on peut aussi écrire en utilisant la commande (( )) à la place de la commande test.

```
if  (( $mavariable != 0 ))
then
    resultat=$((10 / mavariable))
fi
```

4.3.3.2 Condition avec alternative

La structure conditionnelle peut inclure une partie **else** qui permet d'exécuter une suite de commandes alternatives au cas où la condition n'est pas vérifiée. Dans ce cas la syntaxe est :

```
if    condition
then
    lignes-commandes-si-vrai
else
    lignes-commandes-si-faux
fi
```

L'interprétation est cette fois : si, et seulement si, la *condition* est vérifiée, alors les ***lignes-commandes-si-vrai*** seront exécutées sinon les ***lignes-commandes-si-faux*** seront exécutées.

Exemple

On peut ainsi récrire l'exemple de la section précédente avec le fichier monCV en utilisant une structure conditionnelle :

```
if   [   -f monCV   ]
then
     cat monCV
else
    echo "le fichier monCV n'existe pas" >&2
fi
```

4.3.3.3 Conditions imbriquées

Parfois une simple alternative ne suffit pas à traduire ce qu'on veut programmer, en particulier si plusieurs conditions entrent en jeu. Heureusement il est possible d'imbriquer des structures conditionnelles à l'intérieur d'autres structures conditionnelles pour obtenir des constructions telles que :

```
if    condition-1
then
    lignes-commandes-si-vrai-1
else
if   condition-2
then
    lignes-commandes-si-vrai-2
else
if condition-3
then
...
fi
fi
fi
```

Pour simplifier l'écriture de telles structures imbriquées le Bash permet de contracter un **else** suivi d'un **if** avec le mot-clé **elif** et de terminer les imbrications par un seul **fi**.
On peut donc écrire plus simplement :

```
if  condition-1
then
   lignes-commandes-si-vrai-1
elif  condition-2
then
      lignes-commandes-si-vrai-2
elif  condition-3
then
...
fi
```

Exemple
Pour illustrer les imbrications de structures conditionnelles, écrivons un script dont le traitement consiste à écrire la phrase « I love Bash » à la fin d'un fichier qui s'appelle **message**.
Pour écrire ce script, qu'on nommera lovebash.sh, nous devons faire plusieurs vérifications : est-ce que le fichier message existe, est-ce un fichier ordinaire et est-il accessible en écriture. Si toutes les conditions sont réunies on ajoute la phrase « I love Bash » à la fin de ce fichier en utilisant la redirection >> et s'il n'existe pas on le crée en redirigeant le résultat de la commande echo dans le fichier avec >. Écrivons déjà cela :

```
#!/bin/bash
If  [  -f message  -a  -w  message  ]
then
echo  "I love Bash"  >>  message
else
echo  "I love Bash"  >  message
fi
```

Tel qu'il est écrit, ce script va tenter de créer ou d'écraser le fichier message si le test est faux. Mais ce test peut être faux pour plusieurs raisons : soit le fichier n'est pas un fichier ordinaire, soit le fichier est un fichier ordinaire mais il n'est pas accessible en écriture, soit il n'existe tout simplement pas. Or si ça n'est pas un fichier ordinaire ou bien s'il n'est pas accessible en écriture la commande dans la partie else va provoquer une erreur.
Nous pouvons compléter ce script en imbriquant des elif pour détailler tous les cas : si le message n'existe pas on le crée ; si c'est un fichier ordinaire mais sans droit d'écriture, on modifie les droits et on écrit dedans ; sinon on affiche un message sur la sortie standard d'erreur.

```
#!/bin/bash
if [ -f message -a -w message ]
then
echo "I love Bash" >> message
elif [ ! -e message ]
then
echo "I love Bash" > message
elif [ -f message -a ! -w message ]
then
chmod u+w message
echo "I love Bash !" >> message
else
echo "message existe mais n'est pas un fichier ordinaire" >&2
fi
```

4.3.4 La structure de branchement conditionnel case

La commande case est une bonne alternative à l'utilisation de **if** imbriqués, en particulier quand il s'agit de tester une valeur et d'exécuter différentes commandes en fonction de cette valeur. La syntaxe de la commande case est la suivante :

```
case expression in
motif1)
        lignes-commandes1;;
motif2)
        lignes-commandes2;;
...
motifN)
        lignes-commandesN;;
esac
```

Notez les particularités de cette syntaxe :

• l'expression à tester est encadrée des mots-clés case et in sur une seule ligne

• suivent ensuite chacun des cas introduits par un motif suivi d'une parenthèse fermante) puis de la liste des commandes à effectuer terminée par un double point-virgule ;;

• enfin la structure est fermée par le mot clé esac seul sur une ligne.

On peut traduire cette structure de contrôle par :

• au cas où la valeur de *expression* est de la forme *motif1*, exécuter les *lignes-commandes1* ;

• au cas où la valeur de *expression* est de la forme *motif2*, exécuter les *lignes-commandes2* ;

• . . .

• au cas où la valeur de *expression* est de la forme *motifN*, exécuter les *lignes-commandesN*.

L'exécution d'un **case** se déroule de la manière suivante : d'abord l'*expression* est évaluée afin d'obtenir une chaîne de caractères. Celle-ci est comparée avec chaque motif, dans l'ordre. Au premier motif qui correspond les *lignes-de-commandes* sont exécutées jusqu'au **;;** puis le case s'arrête en retournant comme code retour celui de la dernière commande exécutée.

Ainsi la structure case exécute les commandes en fonction du filtrage de la valeur de l'*expression* selon des motifs. Les motifs sont décrits avec la même syntaxe que celle des

abréviations pour le nom des fichiers. Les caractères spéciaux pouvant être utilisés sont définis dans le tableau 2.
La plupart du temps l'expression est une substitution de variable (on veut tester le contenu de la variable) ou le résultat de la substitution d'une commande ou d'une expression numérique (dans ce dernier cas c'est la chaîne de chiffres qui est testée).
Si aucun des motifs ne correspond à la valeur de l'expression alors aucune commande n'est exécutée et le code retour de la commande case est 0. Toutefois s'il y a besoin d'exécuter des commandes dans le cas où aucun des motifs ne correspond on peut utiliser le motif * comme cas par défaut puisque celui-ci filtre n'importe quelle chaîne de caractères.
Normalement si la structure est bien écrite les différents cas doivent être disjoints, c'est-à-dire que pour une valeur donnée de l'expression un seul motif doit correspondre. Si ça n'est pas le cas seules les commandes du premier motif correspondant à la valeur de l'expression seront exécutées.
Si plusieurs cas doivent mener à l'exécution d'une même liste de commandes on peut écrire plusieurs motifs séparés par une barre verticale :

```
case   expression in
motif1 | motif2 | motif3)
    lignes-commandes123;;
...
esac
```

Exemple
Le script suivant écrit le type d'un fichier donné en argument en se fondant sur l'extension du nom de fichier. On utilise le caractère spécial '*' dans les filtres pour ignorer le nom du fichier jusqu'à l'extension. Le troisième cas est constitué de trois filtres séparés par une barre verticale. Le script commence par vérifier qu'il y a bien exactement un argument. Si ce n'est pas le cas, un message d'erreur et l'usage du script sont affichés dans la sortie d'erreur (numéro de canal 2).

```
#!/bin/bash
if [ $# -ne 1 ]
then
        echo "Erreur : nombre d'argument incorrect" >&2
        echo "Usage : $0 fich" >&2
        exit 1
fi
case $1 in
        *.c)
            echo "Code source en langage c";;
        *.sh)
            echo "Script bash";;
        *.jpeg | *.jpg | *.png)
            echo "Image";;
        *)
            echo "Type de fichier non reconnu";;
esac
```

Ce deuxième exemple demande à l'utilisateur de répondre par oui ou non à une question donnée en argument (à condition qu'il y en ait une). Il prévoit toutes les formes de réponses possibles pour oui ou non. La réponse est lue sur l'entrée standard par la commande **read** et placée dans la variable qui suit la commande.

```
#!/bin/bash
If [ $# -eq 1 ]
then
echo "$1 ?"
echo "Etes vous d'accord (oui ou non) ?"
read rep
case $rep in
[oO] | [oO][uU][iI])
echo "Ok, merci"
;;
[nN] | [nN][oO][nN])
echo "Tres bien, je respecte votre choix"
;;
*)
echo "Desole, je ne comprends pas votre reponse"
;;
esac
fi
```

4.4 Structures itératives

Dans cette partie nous allons nous concentrer sur les structures itératives. Ce sont des commandes qu'on appelle plus simplement des « boucles ». Dans une boucle, un même bloc de commandes s'exécute plusieurs fois. On écrit une seule fois le bloc de commandes à répéter au sein de la boucle, et ce bloc sera ensuite exécuté un certain nombre de fois. On dit que la boucle fait plusieurs tours. Le nombre de tours qui sera réalisé est déterminé par le type de boucle utilisé et les paramètres spécifiés pour celle-ci.

Il existe plusieurs types de boucles. Nous allons étudier ici les deux plus courantes. Il s'agit :

• de l'itération conditionnelle ; elle continue de boucler sur le bloc de commandes tant que la condition de continuation est vérifiée.

• de l'itération bornée ; elle réalise l'exécution du bloc de commandes pour chaque élément d'une liste.

4.4.1 L'itération conditionnelle

L'itération conditionnelle consiste à répéter l'exécution d'une séquence de commandes tant qu'une condition est vérifiée. Cet usage est connu sous le nom de boucle **while** qui est commun à beaucoup de langages de programmation. Avec le Bash, la commande **while** va servir à répéter l'exécution d'une séquence de commandes tant que la condition après le **while** retourne un code retour à vrai. La condition peut être une commande quelconque ou l'évaluation d'une expression. Dans ce dernier cas, l'expression est évaluée à l'aide de la commande test.

Il faut rappeler qu'en shell, le test d'une expression retourne un code retour 0 pour vrai et une valeur différente de 0 pour faux. Il faut être très vigilant à ce niveau, car généralement dans les autres langages la signification du vrai et du faux est inversée.

```
while   condition
do
        commande1
        commande2
...
done
```

La condition à considérer s'écrit après le mot clé **while**, sur la même ligne. Vient ensuite la séquence des commandes pour constituer le corps de la boucle sur lequel les boucles vont être effectuées. Le corps de la boucle est délimitée par les mots clés **do** et **done**. Ces délimiteurs doivent chacun être isolés sur une ligne. Pour une écriture plus compacte, il est courant d'utiliser l'opérateur enchaînement séquentiel de commande '**;**' pour mettre **while** et **do** sur une même ligne.

```
While   condition ; do
        commande1
        commande2
...
done
```

4.4.1.1 Exemples d'utilisation

Écrivons un petit script pour illustrer l'usage de cette boucle. Son traitement va consister à répéter le mot qui lui est fourni un certain nombre de fois en fonction d'un nombre indiqué. Nous allons donc appeler ce script repeat.sh. Il aura besoin du mot à répéter et du nombre de répétitions à faire. Ce nombre sera par défaut *10*. Ces deux paramètres seront indiqués comme argument lors du lancement du script.

Écrivons déjà la structure de la boucle while, en laissant en suspens le test pour le moment, et écrivons juste la commande à répéter entre les délimiteurs do et done :

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
while ??? ; do
echo $mot
done
```

Maintenant que la structure est en place, il nous faut faire en sorte d'effectuer *10* tours dans cette boucle. Nous allons donc définir une variable i qui va nous servir à compter le nombre de tours. Au départ cette variable aura la valeur 0, et à la fin de chaque tour nous allons incrémenter sa valeur à l'aide de la syntaxe la substitution arithmétique $((...)) :

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
i=0
while ??? ; do
   echo $mot
   i=$(($i+1))
done
```

Grâce à cette information du nombre de tours, nous pouvons facilement déterminer le test qu'il nous faut écrire pour calibrer correctement notre boucle. Il va s'agir de comparer la valeur du compteur **i** avec le nombre de tours à réaliser que nous avons conservé dans la variable nb.
Tant que la valeur de i est inférieure à celle de nb on continue. Cela s'écrit avec l'opérateur -lt (*less than*), qui veut dire « inférieure à » en français :

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
i=0
while [ $i -lt $nb ] ; do

echo $mot

i=$(($i + 1))
done
```

Voilà il ne nous reste plus qu'à enregistrer, fixer le droit d'exécution et tester le script :

```
$ chmod +x repete.sh
$ ./repete.sh Bonjour 3
Bonjour
Bonjour
Bonjour
$
```

4.4.1.2 Lecture de fichier avec while et read

Une utilisation classique de la boucle while est la lecture de fichier ligne à ligne. Cette lecture se fait en utilisant la commande read. Mais prenons tout de suite un exemple avec le script suivant appelé print.sh.

```
#!/bin/bash
a=0
while read ligne; do
   ((a++))
   echo $a $ligne
done
```

Ce script lit une ligne depuis l'entrée standard, l'affiche précédée de son numéro de ligne et recommence tant que la fin de fichier (indiquée par ctrl + d) n'a pas été rencontrée. Il faut savoir que la commande read retourne le code vrai (0) quand la fin de fichier n'a pas été lue. Ainsi la condition d'arrêt de cette boucle est définie par la commande read quand elle lit la fin du fichier. A noter que la variable ligne prend à chaque itération de la boucle le contenu d'une nouvelle ligne lue de l'entrée standard. Une ligne se définit comme une succession de caractères qui se termine par le caractère de fin de ligne.
Dans la version présentée, ce script doit recevoir sur son entrée standard le contenu d'un fichier comme par exemple :

```
$ cat print.sh | ./print.sh
```

Si à la place d'un contenu, c'est un nom de fichier qui est donné en argument comme :

```
$ ./print.sh print.sh
```

alors la redirection d'entrée doit être utilisée ; le code du script print.sh devient :

```
#!/bin/bash
a=0
while read ligne; do
((a++))
echo $a $ligne
done <"$1"
```

La redirection prend place après le marqueur de fin de commande de la boucle while à savoir après le done. Le symbole $1 représente le premier argument de la ligne de commande et donc le nom du fichier à lire en entrée de la boucle. Lançons le script print.sh avec comme argument le fichier print.sh. Nous obtenons :

```
$ ./print.sh print.sh
1 #!/bin/bash
2 a=0
3 while read ligne; do
4 ((a++))
5 echo $a $ligne
6 done <"$1"
```

Nous avons présenté ici une des utilisations de la commande read avec la boucle while pour lire une ligne de l'entrée standard. Utilisée avec plusieurs arguments la commande read permet aussi de découper la ligne lue : le premier argument capte le premier mot de la ligne, le deuxième argument le deuxième mot, etc. S'il y a moins d'arguments que de mots sur la ligne, le dernier argument capte tout le reste de la ligne. Nous ne rentrerons pas plus dans les détails de cette commande.

4.4.1.3 Contrôle de la boucle while

Nous avons vu dans les sections précédentes que la boucle while est contrôlée par une condition. Il est aussi possible d'introduire des ruptures de séquence dans le corps de la boucle avec les commandes break et continue. Les commandes break et continue sont le plus souvent employées avec la boucle while, mais elles peuvent aussi être utilisées avec une boucle for.

4.4.1.4 La commande continue

La commande continue renvoie en début de boucle. Les commandes du corps de la boucle après la commande continue ne sont pas exécutées. Prenons tout de suite un exemple que nous allons commenter ensuite. Soit le script suivant que l'on appellera continue.sh

```
#!/bin/bash
a=0
while [ "$a" -le 10 ]; do
a=$(($a+1))
if [ "$a" -eq 2 ] || [ "$a" -eq 8 ]
then
  continue      # renvoie au debut de la boucle while
fi
echo -n "$a " # Cette partie ne sera pas exécutée pour a=2 et a=8
done
```

Si on lance le script continue.sh on obtiendra :

```
$ ./continue.sh
1 3 4 5 6 7 9 10 11
```

Nous voyons dans le résultat que ni 2 ni 8 ne sont affichés. On peut voir la commande continue comme un retour direct à la ligne while ["$a" -le 10]; do. Attention ici de ne pas créer une boucle infinie, ce qui peut arriver si la ligne a=$(($a+1)) est placée après la commande continue puisque la variable a ne sera alors plus incrémentée. La commande continue peut prendre un paramètre numérique qui est utilisé en cas de boucles imbriquées. Ce paramètre indique alors le nombre de boucles imbriquées qu'il faut « remonter ».

4.4.1.5 La commande break

La commande break termine la boucle en sortant directement de celle-ci. Prenons le script précédent et changeons la commande continue par la commande break pour constituer le script break.sh.

```
#!/bin/bash
a=0
while [  "$a"  -le  10  ]; do
a=$(($a+1))
if  [  "$a"  -eq  2  ]  ||  [  "$a"  -eq  8  ]
then
   break          # quitte directement la boucle while
fi
   echo -n "$a "      # Cette partie ne sera pas exécutée pour a supérieur ou égal à 2
done
```

Si on lance le script break.sh on obtiendra :

```
$ ./break.sh
1
```

Nous voyons dans le résultat que la boucle s'arrête dès que la valeur de la variable a est égale à 2. La commande break renvoie directement à la ligne **done**. La commande break peut aussi prendre en argument un niveau d'imbrication de boucle. Par exemple, **break 3** va « terminer » trois boucles imbriquées.

Avec la commande **break**, il est possible de faire une boucle avec plusieurs conditions de sortie ou d'avoir la condition de sortie dans le corps de la boucle. Le script ci-contre illustre l'arrêt de la boucle sur une fin de fichier ou sur la saisie de la lettre **'q'** :

```
#!/bin/bash
while true ; do
echo "faire qqchose 1"
if ! read -p "? " lettre ; then
break # quitte sur fin de fichier (CTRL+D)
fi
echo "faire qqchose 2"
if [ "$lettre" == "q" ] ; then
break # quitte sur lettre q
fi
echo "faire qqchose 3"
done
```

4.4.2 L'itération bornée

Le deuxième type de boucle est celui consistant à répéter l'exécution de la séquence de commandes pour chacun des éléments d'une liste autrement dit la séquence de commandes est à répéter un nombre connu de fois. En shell, on utilise pour cela la commande for. On trouve l'équivalent de cette commande dans les autres langages de programmation, où elle implique généralement l'usage d'un compteur. En shell les choses sont un peu différentes, la commande for va en fait parcourir les éléments d'une liste.

La syntaxe de cette structure fait donc apparaître :

• La liste à parcourir : il s'agit simplement d'une suite de valeurs séparées par des espaces ou des sauts de lignes.

• Le nom de la variable : qui à chaque tour de boucles va recevoir successivement chacune des valeurs de la liste.

• Et la séquence de commandes à exécuter à chaque passage.

```
for variable in liste
do
    commande1
    commande2
...
done
```

La liste à parcourir peut-être indiquée littéralement mais elle peut provenir suite à une substitution de variable, de commande ou de nom de fichier. Si la liste n'est pas spécifiée, c'est la liste des arguments de la ligne de commande qui est retenue par défaut.

Comme pour la commande while, les commandes continue et break sont utilisables avec la boucle for.

4.4.2.1 Exemple d'utilisation

Pour mettre en pratique cette boucle for, nous allons faire un premier test simple avec un script de nom testFor.sh :

```
$ vi testFor.sh
```

Le code de ce script va consister à parcourir une liste de 3 noms avec une variable appelée courant. À chaque tour de boucle on va juste afficher la valeur courante de cette variable :

```
#!/bin/bash
for courant in Aicha Ibou Fatou
do
   echo $courant
done
```

Notez qu'une liste d'éléments s'écrit simplement sous la forme d'une suite de valeurs (ici des prénoms) séparées par des espaces.
On enregistre le fichier, on fixe le droit d'exécution, et on lance le script :

```
$ chmod +x testFor.sh
$ ./testFor.sh
Aicha
Ibou
Fatou
$
```

On constate que chacun des noms s'affiche bien l'un après l'autre.

4.4.2.2 Liste issue d'une variable

La liste à parcourir peut aussi être exprimée par la valeur d'une variable. Par exemple, avec $@ qui contient la liste de tous les arguments de lancement du script. Nous allons justement faire un petit script avec une boucle for qui va parcourir les valeurs présentes dans cette variable $@. Cela va avoir pour effet d'afficher toutes les valeurs qui sont fournies en argument de lancement du script. Nous allons appeler ce script afficheArg.sh :

```
$ vi afficheArg.sh
```

On va utiliser une variable num pour numéroter chaque argument. Dans la structure du for, nous allons utiliser une variable var pour parcourir les éléments de $@. À chaque tour de boucle, on affichera le numéro puis la valeur de var et on incrémentera la valeur de la variable num pour la prochaine itération.

```
#!/bin/bash
num=1
for var in "$@" ; do
  echo "argument $num = $var"
  ((num++))
done
```

On enregistre le fichier, on fixe le droit d'exécution, et on lance le script avec quelques arguments :

```
$ chmod +x afficheArg.sh
$ ./afficheArg.sh yap "Thieb Thieb"
argument 1 = yap
argument 2 = Thieb Thieb
```

4.4.2.3 Liste issue d'une commande

En choisissant bien la liste à parcourir on peut revenir aisément au fonctionnement plus classique d'une boucle *for* à base de compteur. En effet, il suffit simplement d'utiliser

l'affichage produit par la commande seq pour générer la liste. Cette commande attend 2 nombres en argument, et elle va produire l'affichage des valeurs successives entre ces 2 nombres :

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

On va mettre cette possibilité en pratique en reprenant notre script repeat.sh. On va modifier son code pour utiliser cette fois-ci un for au lieu d'un **while**. La variable **i** va maintenant être celle qui va parcourir la liste du for, et cette liste va être produite par la commande seq sur l'intervalle 1 à *nb*. Nous allons capturer cet affichage sous forme de valeur grâce à la substitution de commande, à l'aide de la syntaxe $(...).

```
#!/bin/bash
mot=${1:?"Vous devez indiquer un mot"}
nb=${2:-10}
for i in $(seq 1 $nb) ; do
    echo $mot
done
```

Avec ce type d'écriture du for on n'a plus besoin d'incrémenter le compteur i à chaque tour, car il va passer tout seul d'une valeur à l'autre à l'aide de la séquence générée par **seq**. On enregistre, et on peut tester que le script fonctionne aussi bien qu'avant :

```
$ ./repete.sh Bonjour 3
Bonjour
Bonjour
Bonjour
$
```

Le Bash offre aussi un moyen de générer une liste au moyen du développement d'accolades (*Braces expansion*). Cette construction syntaxique génère des chaînes de caractères. Par exemple img.{pdf ,png,jpg} va produire img.pdf img.png img.jpg. Le développement d'accolades trouve aussi son intérêt pour produire des suites. La suite des nombres de 0 à 9 s'écrit {0..9}

```
$ echo {0..9}
0 1 2 3 4 5 6 7 8 9
```

Imaginons que vous voulez produire une suite avec des numéros non continus comme par exemple les numéros des différentes activités de ce cours.

```
$ echo A{1..4}{0..6}
A10 A11 A12 A13 A14 A15 A16 A20 A21 A22 A23 A24 A25 A26 A30 A31 A32 A33 A34 A35
A36 A40 A41 A42 A43 A44 A45 A46
```

Si nous voulons produire l'affichage d'un tableau. Le script ci-dessous le construit rapidement pour vous avec le développement d'accolades et une boucle for.

```
#!/bin/bash
    echo "|  Act.  | Commentaire |"
for i in A{1..4}{0..6} ; do
    echo "|  $i  |                    |"
done
```

L'exécution produit :

Act.	Commentaire
A10	
A11	
...	
A45	
A46	

4.4.2.4 Boucle avec un compteur

La boucle for peut aussi prendre une syntaxe héritée du langage de programmation C. Cette variante est caractérisée par trois paramètres qui contrôlent la boucle for. Ces trois paramètres sont EXP1 : qui est l'initialisation de la boucle, EXP2 qui présente le test qui conditionne l'exécution du prochain tour de boucle, et EXP3 qui spécifie l'expression de comptage. EXP1, EXP2, et EXP3 sont des expressions arithmétiques.

```
for (( EXP1; EXP2; EXP3 ))
do
  command1
  command2
done
```

Cette structure de boucle est équivalente à

```
(( EXP1 ))
while (( EXP2 )); do
   command1
   command1
   (( EXP3 ))
done
```

Voyons un exemple ci dessous dans le script for3.sh.

```
#!/bin/bash
for ((i=1; i<=4; i++)); do
   echo "Nombre $i"
done
```

Dans le script for3.sh, le premier paramètre de la boucle for est la déclaration et l'initialisation d'une variable i à 1. Le deuxième paramètre est la condition d'entrée dans le prochain tour de boucle, dans notre cas i doit être inférieur ou égal à 4. Le troisième paramètre est l'expression de comptage qui à chaque passage de la boucle incrémente la variable i de 1. Dans le corps de la boucle nous affichons simplement la valeur de la variable i à chaque itération de la boucle. Le résultat de ce script est :

```
$ ./for3.sh
Nombre 1
Nombre 2
Nombre 3
Nombre 4
```

4.5 Structures de routines

Que ce soit en mode script ou en mode interactif, certaines actions sont à faire de manière répétitive et espacée dans le temps. La notion de fonction répond à ce besoin. Les fonctions sont également appelées routines, méthodes ou procédures. Nous allons voir dans ce chapitre comment regrouper des commandes pour former comme des petits scripts à l'intérieur d'un script mais aussi à l'intérieur du shell. La notion de fonction réduit dans le premier cas la répétition de code dans un script. C'est une bonne pratique car elle encourage la réutilisation de code et évite d'écrire des scripts trop longs, difficiles à lire et au final durs à maintenir. Dans le second cas, elle offre le moyen de composer ses propres commandes pour interagir en ligne de commande. En mode interactif, le Bash offre aussi la notion d'alias pour raccourcir la saisie des commandes les plus courantes.

Nous allons commencer par présenter les éléments d'une fonction pour son utilisation dans un script.

Nous indiquerons également comment organiser le code d'un script avec des fonctions. Nous terminerons ce chapitre en présentant les moyens de créer ses propres commandes pour que l'utilisateur puisse adapter ses commandes courantes à ses habitudes et à son activité.

4.5.1 Notion de function

Comme dans la plupart des langages de programmation, la fonction est un moyen pour regrouper un ensemble de commandes qui peuvent s'exécuter à plusieurs reprises. Elle est un élément important pour ré-utiliser des regroupements de commandes. La fonction est une chose assez classique en programmation. Une fonction en shell est un objet qui est appelé comme une simple commande et exécute un regroupement de commandes avec un nouveau jeu de variables positionnelles.

L'usage d'une fonction nécessite plusieurs étapes. Il faut d'abord déclarer la fonction c'est à dire donner son nom et définir son traitement. Ensuite, il faut effectuer l'appel de la fonction avec éventuellement des arguments. Nous allons voir la réalisation de ces étapes en indiquant comment :

• déclarer et utiliser une fonction dans un fichier script,
• passer des arguments à une fonction,
• définir des variables locales au code d'une fonction et gérer correctement la valeur que doit retourner une fonction.

4.5.1.1 Définir une function

La déclaration d'une fonction en shell peut se faire en 2 formats différents

```
nom_fonction ( ) {
    commande1
    commande2
    ...
}
```

Ou

```
function nom_fonction {
    commande1
    commande2
    ...
}
```

Le mot function est optionnel. S'il est utilisé, les parenthèses sont optionnelles. Le corps de la fonction est constitué par le regroupement de commandes défini entre les accolades.
Aussi concrètement, pour définir une fonction, il faut :
• sur une nouvelle ligne du fichier source, écrire le nom de la fonction.
• ce nom doit être précédé par le mot réservé function ou ce nom doit être suivi d'un couple de parenthèses ouvrant/fermant.
• ensuite encadrée par des accolades, le corps de la fonction est défini par une suite des commandes qui expriment le traitement à réaliser par la fonction.
Par exemple une simple fonction de nom say_hello qui va simplement afficher le texte : *Hello World !!!* se définit de la manière suivante :

```
function say_hello {
     echo "Hello World !!!" ; }
```

Un point de vigilance, si l'accolade fermante est sur la même ligne que la dernière commande, cette dernière doit être suivie par le caractère de séparation de commandes '**;**'.

4.5.1.2 Invoquer une function

Quand on définit une fonction avec la syntaxe ci-dessus, la fonction n'existe que pour le fichier script dans lequel elle a été définie. De plus, la fonction doit être déclarée avant d'en faire l'appel. On prendra donc l'habitude de déclarer systématiquement toutes les fonctions au début des fichiers scripts.
Pour utiliser la fonction, il suffit simplement de taper son nom, comme si c'était une commande shell classique. Ainsi, quand on définit une fonction dans un script c'est un peu comme si on créait une nouvelle commande shell propre à ce script. Reprenons l'exemple précédent, on va commencer par créer un fichier script de nom hello.sh :

```
$ vim hello.sh
```

On va définir la fonction say_hello au début de ce fichier. On va ensuite effectuer un appel de cette fonction. Il suffit pour cela de taper son nom. Et on va même faire un deuxième appel de cette fonction juste en dessous. Et pour que les choses soient plus visibles, des commentaires sont ajoutés pour bien montrer où se trouve la définition de la fonction, et où se trouvent les appels de celle-ci.

```
#!/bin/bash
# définition de la fonction
function say_hello {
    echo "Hello World !!!"
}#
Appel de la fonction
say_hello
say_hello
```

On enregistre. On fixe le droit d'exécution, et on lance le script :

```
$ chmod +x hello.sh
$ ./hello.sh
Hello World !!!
Hello World !!!
$
```

On a bien l'affichage attendu.

Il est important de noter que l'exécution de la fonction s'effectue dans le même contexte que l'appelant, à la différence d'un script qui s'exécute par un sous-shell et qui possède son propre contexte. On peut voir la fonction comme équivalent au regroupement de commandes noté entre accolades auquel un nom a été associé pour l'identifier et l'appeler. Pour illustrer le point commun entre une fonction et le regroupement de commandes, prenons l'exemple du test du contenu d'une variable qui est incrémentée et affichée si elle vaut la valeur 10. Avec le regroupement de commandes, nous écrivons un enchaînement conditionnel de la manière suivante :

```
$ a=10
$ [ "$a" -eq 10 ] && { ((a++)) ; echo $a ; }
11
```

La variable a est définie dans le contexte du shell, si elle vaut 10, elle passera à 11. En utilisant une fonction, nous devons la définir préalablement à son appel. Dans ce cas, l'exemple devient comme indiqué ci-dessous et nous aurons le même comportement.

```
$ fun() {
> ((a++)) ; echo $a ; }
$ a=10 ; [ "$a" -eq 10 ] && fun
11
```

La fonction est bien ici équivalente au regroupement de commandes. Maintenant si nous avons recours au script fun.sh ci-dessous :

```
#!/bin/bash
((a++))
echo $a
```

Le script s'exécute dans un autre contexte, dans lequel la variable appelée a prend la valeur 0 (c'est en fait une autre variable mais avec le même nom). C'est ce que nous pouvons en déduire avec l'exécution ci-dessous :

```
$ a=10
$ [ "$a" -eq 10 ] && fun.sh
1
```

4.5.1.3 Les arguments d'une function

Comme on vient de le voir, la syntaxe d'appel d'une fonction est identique à celle d'une commande classique. Cela se vérifie aussi pour le cas où des arguments doivent être passés à la fonction. Sur la commande d'appel, ces arguments sont placés les uns après les autres juste après le nom de la fonction :

```
nom_fonction   arg1 arg2 ... argN
```

On utilise simplement une ou plusieurs espaces pour les séparer. Les arguments de la fonction sont placés dans des variables positionnelles. Il s'agit en effet des mêmes variables que l'on a déjà utilisées pour exploiter les arguments des scripts. Le corps de la fonction définit et utilise des paramètres notés sous la forme de variables positionnelles. On retrouve donc les variables :

• $1, $2, $3, . . . qui vont contenir les arguments passés à l'appel de la fonction.
• $# qui aura pour valeur le nombre d'arguments transmis au moment de l'appel.
• $@ qui contiendra l'ensemble des arguments.:

C'est le fait de se trouver dans le corps d'une fonction qui va donner cette signification à ces variables. En dehors du code de la fonction, elles retrouvent leur rôle précédent qui est celui lié aux arguments du script.

À titre d'illustration, nous allons modifier notre script hello.sh de sorte que la fonction say_hello utilise le premier argument qui lui est passé. Cette fonction affiche dans sa phrase le mot passé en argument. Ainsi une phrase différente pourra s'afficher pour chaque argument différent placé à la fin du script.

```
#!/bin/bash
#definition de la fonction
say_hello() {
   echo "Hello $1 !!!"
}#
utilisation par appel de la fonction
say_hello Me
say_hello You
```

On enregistre, et on lance l'exécution.

```
$ ./hello.sh
Hello Me !!!
Hello You !!!
$
```

Voilà qui devrait rendre la fonction say_hello un peu plus utile !

4.5.1.4 Les variables locales à une fonction

Quand on manipule une variable dans un script, les changements subis par cette variable valent pour l'ensemble du script. Et c'est aussi le cas pour les variables utilisées dans le code d'une fonction. Aussi, on dit que les variables sont *globales* au script, ou encore que la *portée* d'une variable soit globale.

Il est cependant possible de limiter la portée d'une variable dont l'usage ne concerne que l'espace du corps d'une fonction. Cela permet d'éviter d'entrer en conflit avec les autres traitements du script.
Car quand le code source d'un script est long, le risque d'utiliser le même nom de variable à deux endroits différents, sans s'en rendre compte, est grand. Pour obtenir cette limitation, il suffit d'utiliser la commande **local**. Cette commande doit se placer au début du code de la fonction, et elle prend en argument le nom de la variable concernée avec éventuellement la valeur de départ de celle-ci :

```
local nom_variable
```

ou

```
local nom_variable=valeur
```

Une fois marquée comme locale, les manipulations réalisées sur la variable dans la fonction ne vont pas avoir d'incidence sur le reste du code script, et ce même si on utilise une variable de même nom ailleurs. En tant que variable locale à la fonction, elle sera toujours considérée comme une nouvelle variable propre à l'exécution de la fonction.