esiea

SYSTÈME

Linux : les Scripts shell

By Rexy

SOMMAIRE

1. INTRODUCTION

Le Shell est un "**INTERPRÉTEUR DE COMMANDE**". Il ne fait pas partie du système d'exploitation ; c'est la raison pour laquelle il porte ce nom (coquille), par opposition au "noyau".
Son rôle est d'analyser la commande tapée afin de faire réagir le système en fonction des besoins de l'utilisateur. C'est le premier langage de commandes développé sur UNIX par Steve BOURNE.
D'autre Shell ont ensuite été développés à partir de cette base. Nous trouvons entre autres (liste non exhaustive) :

- le Bourne Shell ("/bin/sh")
- le Korn Shell ("/bin/ksh")
- le C-Shell ("/bin/csh") pour les utilisateurs préférant un langage apparenté au "C"
- le job Shell ("/bin/jsh")
- le Shell réseau ("/bin/rsh")
- le Bourne again Shell ("/bin/bash") qui a repris le Bourne Shell, mais qui l'a agrémenté de nouvelles fonctionnalités (rappel de commandes, terminaison automatique de mots, etc.)

C'est un langage de commandes, mais aussi un langage de programmation. Il permet donc :

- l'utilisation de variables
- la mise en séquence de commandes
- l'exécution conditionnelle de commandes
- la répétition de commandes

1.1. Présentation

Il existe deux moyens de « programmer » en Shell.

Le premier est dit en « direct ». L'utilisateur écrit « directement » la ou les commandes qu'il veut lancer. Si cette commande a une syntaxe qui l'oblige à être découpée en plusieurs lignes, le Shell indiquera par l'affichage d'un « prompt secondaire » que la commande attend une suite et n'exécutera réellement la commande qu'à la fin de la dernière ligne.

Exemple :

```
[user1 ]$  date
Tue Jan 16 17:26:50 NFT 2001
[user1 ]$  pwd
/tmp
[user1 ]$  if test "5" = "5"
> then
> echo "oui"
> else
> echo "non"
> fi
oui
```

Le second est dit en « script » ; appelé aussi « batch » ou « source Shell ». L'utilisateur crée un fichier texte au moyen de l'éditeur de son choix (ex. : « vi »). Il met dans ce script toutes les commandes qu'il voudra faire exécuter ; en respectant la règle de base de ne mettre qu'une seule commande par ligne. Une fois ce script fini et sauvegardé, il le rend exécutable par l'adjonction du droit "x" (*cf. Gestion des fichiers*). Il peut ensuite lancer l'exécution de ce fichier comme n'importe quelle autre commande (attention cependant à la variable « PATH »). Toutes les commandes inscrites dans le fichier texte seront exécutées séquentiellement.

Exemple :

```
Lancement de l'éditeur
[user1 ]$  vi prog

Mise à jour du droit d'exécution
[user1 ]$  chmod u+x prog

Exécution du script situé dans le
répertoire courant
[user1 ]$  ./prog
Tue Jan 16 17:26:50 NFT 2001
/tmp
oui
```

Contenu du script "prog" :

```
date
pwd
if test "5" = "5"
then
                    echo "oui"
else
                    echo "non"
fi
```

Remarque :
L'adjonction du droit "x" n'est pas obligatoire, mais l'utilisateur devra alors spécifier au système quel Shell a la charge d'interpréter le fichier "source" écrit.

Exemple : L'utilisateur demande d'interpréter le script par l'intermédiaire du "*Bourne Shell*"

```
[user1 ]$  sh prog
Tue Jan 16 17:26:50 NFT 2001
/tmp
oui
```

Exemple : L'utilisateur demande d'interpréter le script par l'intermédiaire du "K*orn Shell*"

```
[user1 ]$  ksh prog
Tue Jan 16 17:26:50 NFT 2001
```

```
/tmp
oui
```

1.2. Remarques

Comme le Shell est un interpréteur, chaque ligne est analysée, vérifiée et exécutée une à une. Afin de ne pas perdre en rapidité d'exécution, il y a très peu de règles d'analyse. Cela implique une **grande rigidité d'écriture** de la part du programmeur. Une majuscule n'est pas une minuscule ; et surtout, **deux éléments distincts sont toujours séparés par un espace**.
Enfin, le premier mot de chaque ligne, si la ligne n'est pas mise en commentaire, doit être une instruction Shell correcte que l'on appelle « instruction ».
Exemple :

```
[user1 ]$  echosalut
sh: echosalut: not found
[user1 ]$  echo salut
salut
```

On distingue ainsi :

- un programme "Unix/Linux" (ex : date)
- un script "Shell" (ex : ./prog)
- une commande interne du langage (ex : if…)

1.3. Conventions du support de cours

Il est difficile d'écrire un support de cours en essayant de faire ressortir des points importants par l'utilisation de caractères spécifiques (guillemets, parenthèses, etc.) sur un langage utilisant certains caractères comme instructions spécifiques (guillemets, parenthèses, etc.). De plus, un lecteur ne sait pas forcément distinguer les éléments provenant de l'ordinateur des éléments qu'il doit taper au clavier. C'est pourquoi il est nécessaire de définir des conventions

- La chaîne « [user1]$ » utilisée dans les exemples de ce cours est une « invite de commande »; appelée aussi « prompt ». C'est une chaîne affichée par le système afin d'indiquer à l'utilisateur (« user1 » dans notre cas) qu'il attend une instruction. Cette chaîne n'est pas forcément la même pour des utilisateurs différents. Dans ce support, sa présence signifie que l'exemple proposé peut être tapé directement en ligne, sans obligatoirement passer par un fichier script.
- Des éléments mis entre crochets « ***[elem]*** » signifient qu'ils sont facultatifs. L'utilisateur peut les utiliser ou pas, mais ne doit en aucun cas mettre de crochets dans sa frappe ou son programme.
- Le caractère « ***n*** » employé dans la syntaxe signifie « nombre entier quelconque positif ou nul ».
- Les points de suspension "**…**" signifient que l'élément placé devant peut être répété autant de fois que l'on désire.

2. ÉLÉMENTS DE BASE DU LANGAGE

2.1. Les commentaires

Un commentaire sert à améliorer la lisibilité du script. Il est constitué d'une ou plusieurs lignes précédées du caractère ***dièse*** (« ***#*** »). Tout ce qui suit ce dièse est ignoré jusqu'à la fin de la ligne ;

cela permet à des instructions d'être suivies de commentaires. Il ne faut pas oublier alors l'espace séparant le dernier mot « utile » de la ligne du caractère ***dièse***.

Exemple :

```
# Ce programme affiche la date
date                    # Cette ligne est la ligne qui affiche la date
```

2.2. Qui analyse ?

Rappelons qu'il existe plusieurs Shells. Chacun possède des caractéristiques particulières. Chaque utilisateur Unix/Linux peut travailler avec son Shell préféré. Ainsi, un script écrit par un utilisateur travaillant en « Bourne Shell » peut ne pas fonctionner s'il est exécuté par un utilisateur travaillant en « C-Shell ». Il est possible d'éviter ce problème en indiquant au début de chaque script l'interpréteur à utiliser (« ***#!interpreteur »***). Quand cette indication n'est pas présente, le système utilisera l'interpréteur de travail de l'utilisateur qui exécute le script.

Exemple :

```
#!/bin/sh
# Ce programme affiche la date en Bourne Shell
date                    # Cette ligne est la ligne qui affiche la date en Bourne Shell
```

Le caractère ***dièse*** combiné au caractère ***point d'exclamation*** n'est plus pris comme un commentaire, mais comme une instruction indiquant quel programme a la charge d'analyser le script. Ce cours proposera globalement une syntaxe « Bourne Shell » qui est comprise par les autres Shells. Les quelques éléments Korn Shell ou Bourne Again Shell seront indiqués au moment où cela sera nécessaire.

2.3. Le débogueur

Un script long et compliqué peut ne pas aboutir du premier coup. Afin d'aider le développeur à détecter l'erreur, trois options de débogage peuvent être exploitées. Il s'agit des instructions :

- set –x (affichage de chaque instruction après analyse de l'instruction
- set –v (affichage de chaque instruction avant analyse de l'instruction)
- set –n (sortie immédiate sur erreur)

On active l'option en écrivant l'instruction "*set* -…" dans le programme. On la désactiver par l'instruction "*set* +…". On peut ainsi activer le « débogueur » sur une portion précise du programme.

Il est aussi possible de spécifier ces options de débogage au lancement du script (Exemple : sh -x mon_script .sh).

Remarque : Compte tenu du flot important d'informations produit par ces options, il peut être parfois avantageux de préférer un affichage des variables pouvant causer l'erreur (commande "*echo*").

3. LES VARIABLES

Il n'est pas de langage sans variable. Une variable sert à mémoriser une information afin de la réutiliser ultérieurement. Elles sont créées par le programmeur au moment où il en a besoin. En script Shell, il n'est pas nécessaire de les déclarer préalablement à leur utilisation. Il est ainsi possible de les créer quand on veut.

Leur nom est représenté par une suite de caractères commençant impérativement par une lettre ou par le caractère « **_** » (« souligné » ou « underscore ») et comportant ensuite des lettres, des chiffres ou le caractère souligné.
Leur contenu est interprété exclusivement comme du texte. Ainsi, les opérations arithmétiques sur des entiers nécessitent un traitement particulier. Il n'est pas non plus possible de dimensionner des variables (tableaux).
Pour toute variable, le Shell reconnaît deux états :

- non définie (non-existence) : elle n'existe pas dans la mémoire
- définie (existence) : elle existe dans la mémoire ; mais elle peut être vide ou pas

3.1. L'affectation – L'accès

Syntaxe :

```
variable=chaîne
```

L'affectation d'une variable se fait par la syntaxe « ***variable=chaîne*** ».
C'est la seule syntaxe du Shell qui ne veuille pas d'espace dans sa structure sous peine d'avoir une erreur lors de l'exécution.
Dans le cas où on voudrait entrer une chaîne avec des espaces dans la variable, il faut alors encadrer la chaîne par des guillemets simples ou doubles (la différence entre les deux sera vue plus tard). À partir du moment où elle a été affectée, une variable se met à exister dans la mémoire, même si elle a n'a pas été initialisée.
L'accès au contenu de la variable s'obtient en faisant précéder le nom de la variable avec le caractère "***$***".

Exemple :

```
[user1 ]$  nom="Mr Pignon"                 # Affectation de "Mr Pignon" dans la variable "nom"
[user1 ]$  objet="voiture"                 # Affectation de "voiture" dans la variable "objet"
[user1 ]$  coul=bleue                      # Affectation de"bleu" dans la variable "coul"
[user1 ]$  echo "Il se nomme $nom"         #Affichage d'un texte contenant une variable
[user1 ]$  phrase="$nom a une $objet $coul"  # Affectation d'un texte contenant des variables
[user1 ]$  echo phrase                     # Affichage du mot "phrase" (oubli du "$")
[user1 ]$ echo $phrase                     # Affichage de la variable "phrase" (oubli rectifié)
```

Remarque : Il est difficile et déconseillé de vouloir manipuler des « variables de variable » appelées aussi « pointeur ». Autrement dit, la syntaxe « $$var » qui pourrait vouloir dire « contenu du contenu de la variable *var* » signifie en réalité « contenu de la variable *$$* suivi de la chaîne *var* » (sera vue ultérieurement).

3.2. La substitution

On peut utiliser des séquenceurs spéciaux pour modifier la manière dont le Shell va renvoyer le contenu de la variable demandée :

- ${var} : renvoie le contenu de "*$var*". Il sert à isoler le nom de la variable par rapport au contexte de son utilisation. Ceci évite les confusions entre ce que l'utilisateur désire "*${prix}F*" (variable "*prix*" suivie du caractère "*F*") et ce que l'utilisateur écrit "*$prixF*" (variable "*prixF*").
- ${var-texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie ; sinon renvoie le

texte "*texte*".

- ${var:-texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie et non vide ; sinon renvoie le texte "*texte*".
- ${var+texte} : renvoie le texte "*texte*" si la variable "*var*" est définie ; sinon ne renvoie rien
- ${var:+texte} : renvoie le texte "*texte*" si la variable "*var*" est définie et non-vide ; sinon ne renvoie rien
- ${var?texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie ; sinon renvoie le texte "*texte*" (comme "*${var-texte}*"), mais sort du programme.
- ${var:?texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie et non-vide ; sinon renvoie le texte "*texte*" (comme "*${var:-texte}*"), mais sort du programme.
- ${var=texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie, sinon affecte le texte "*texte*" à la variable "*var*".
- ${var:=texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie et non-vide, sinon affecte le texte "*texte*" à la variable "*var*".

Les séquenceurs suivants ne sont valables qu'en Korn Shell et Bourne again Shell

- ${var#texte} : renvoie le contenu de la variable "*var*" en lui enlevant son début de chaîne si celle-ci commence par "*texte*".
- ${var%texte} : renvoie le contenu de la variable "*var*" en lui enlevant sa fin de chaîne si celle-ci se termine par "*texte*".
- ${var##texte} : renvoie le contenu de la variable "*var*" en lui enlevant son début de chaîne de façon récursive si celle-ci commence par "*texte*".
- ${var%%texte} : renvoie le contenu de la variable "*var*" en lui enlevant sa fin de chaîne de façon récursive si celle-ci se termine par "*texte*".
- ${#var} : renvoie le nombre de caractères contenus dans la variable "*var*".

Remarque : L'imbrication de séquenceurs est possible. Ainsi, la syntaxe "*${var1:-${var2:-texte}}*" renvoie le contenu de la variable "*var1*" si celle-ci est définie et non nulle ; sinon, renvoie le contenu de la variable "*var2*" si celle-ci est définie et non nulle ; sinon renvoie le texte "*texte*".

3.3. La saisie en interactif

Syntaxe :

```
read var1 [var2 …]
```

Cette action est nécessaire lorsque le programmeur désire demander une information à celui qui lance le programme. À l'exécution de la commande, le programme attendra du fichier standard d'entrée (*cf. Gestion des processus*) une chaîne terminée par la touche "*Entrée*" ou "*fin de ligne*". Une fois la saisie validée, chaque mot (séparé par un "*espace*") sera stocké dans chaque variable. En cas d'excédent, celui-ci sera stocké dans la dernière variable. En cas de manque, les variables non remplies seront automatiquement définies, mais initialisées "vide".

3.4. La protection

Syntaxe :

```
readonly var1 [var2 …]

readonly
```

Cette commande, lorsqu'elle est employée sur une variable, la verrouille contre toute modification et/ou suppression, volontaire ou accidentelle. Une fois verrouillée, la variable ne disparaîtra qu'à la mort du processus qui l'utilise (*cf. Gestion des processus*).
Employée sans argument, la commande "***readonly***" donne la liste de toutes les variables qui ont été protégées.

3.5. La suppression

Syntaxe :

```
unset var1 [var2 …]
```

Cette commande supprime la variable sur laquelle elle est appliquée à condition que cette dernière n'ait pas été protégée par la commande « *readonly* ».
Le mot "suppression" rend la variable ciblée à l'état de "non-défini" ou "non-existant". Il y a libération de l'espace mémoire affecté à la variable supprimée.

3.6. La visibilité

Syntaxe :

```
export var1 [var2 …]

export
```

Lorsqu'un Script Shell est lancé depuis l'environnement d'un utilisateur, ce script commence son exécution (*cf. Gestion des processus*) avec une zone mémoire vierge qui lui est propre. Il ne connaît donc, par défaut, aucune des variables de l'environnement qui lui a donné naissance.
Pour qu'un processus « père » puisse faire connaître une variable à un processus « fils », il doit l'exporter avec la commande "***export var***". Ceci fait, la variable exportée depuis un environnement particulier sera connue de tous les processus « fils » ; et de tous les processus « fils » des « fils », etc.
Cependant, modifier le contenu d'une variable dans un processus quelconque ne reporte pas cette modification dans les environnements supérieurs. Dans la même optique, il n'y a aucun moyen simple pour renvoyer une variable quelconque d'un processus vers un processus supérieur.
Employée seule, la commande "***export***" donne la liste de toutes les variables qui ont été exportées.

Exemple sans exportation :

```
[user1 ]$  # Affectation de "var"
[user1 ]$  var=Bonjour


[user1 ]$  # Lancement de "prog"
[user1 ]$  ./prog
```

```
Contenu de var :
[user1 ]$  # "prog" ne connaît pas
"var"


[user1 ]$  # Affichage de "var"
[user1 ]$  echo $var
```

```
Bonjour
[user1 ]$  # "var" n'a pas changé
```

Exemple avec exportation :

```
[user1 ]$  # Affectation de "var"
[user1 ]$  var=Bonjour


[user1 ]$  # Exportation de "var"
[user1 ]$  export var


[user1 ]$  # Lancement de "prog"
[user1 ]$  ./prog
Contenu de var : Bonjour
[user1 ]$  # "prog" connaît "var"


[user1 ]$  # Affichage de "var"
[user1 ]$  echo $var
Bonjour
[user1 ]$  # "var" n'a toujours pas
changé
```

Contenu du script "prog" :

```
#!/bin/sh
echo "Contenu de var : $var"    # Affichage de la variable "var"
var=Salut                       # Modification de la variable dans le script pour voir si elle remonte
```

3.7. Les variables prédéfinies

Un utilisateur possède lors de sa connexion plusieurs variables automatiquement définies par le système. Certaines sont modifiables, certaines ne le sont pas, mais toutes sont utiles.
Quelques variables prises parmi les plus courantes…

Variable	Signification
HOME	Répertoire personnel de l'utilisateur
PWD	Répertoire courant (uniquement en "Korn Shell" ou "Bourne Again Shell")
OLDPWD	Répertoire précédent (uniquement en "Korn Shell" ou "Bourne Again Shell")
LOGNAME	Nom de login
PATH	Chemin de recherche des commandes
CDPATH	Chemin de recherche du répertoire où aller avec la commande "***cd***"
PS1	Prompt de base
PS2	Prompt de suite
PS3	Prompt utilisé par la commande "***select***" du "Korn Shell" et du "Bourne again Shell"
PS4	Prompt affiché lors de l'utilisation du mode débogueur "***set –x***"
TERM	Type de terminal utilisé

IFS	Séparateur de champs internes
SHELL	Nom du Shell qui sera lancé chaque fois qu'on demandera un Shell dans une application ("***vi***", "***ftp***", etc.)
RANDOM	Nombre aléatoire (uniquement en "Korn Shell" ou "Bourne Again Shell")
$$	Numéro de pid (permets de générer des noms de fichiers uniques)
$!	Numéro de pid du dernier processus lancé en arrière-plan
$?	Statut (résultat) de la dernière commande lancée

4. LA "SOUS-EXÉCUTION"

La "sous-exécution" de commande est un des mécanismes les plus importants en Shell. Il permet de remplacer, lors de l'exécution, une partie du script par **le résultat d'une commande**.
Ce mécanisme s'obtient en encadrant la commande par des « **quotes inversées** » (caractère « ` » (<ALT GR> + « 7 » sur un clavier « azerty »).
Ce mécanisme peut être utilisé en coordination avec beaucoup d'autres qui seront vus ultérieurement. Mais l'utilisation la plus courante est l'affectation de variables.
Exemple :

```
[user1 ]$  var1=`echo Salut`      # Ce que la commande "echo Salut" affiche ira dans "var1"
[user1 ]$  var2=`date`            # Ce que la commande "date" affiche ira dans "var2"
[user1 ]$  var3=`ls –l`           # Ce que la commande "ls -l" affiche ira dans "var3"
[user1 ]$  var4=`cd /`            # Ce que la commande "cd /" affiche (rien) ira dans "var4"
[user1 ]$  var5=pwd               # La chaîne "pwd" ira dans "var5" (oubli des accents grave)
```

Exemple :

```
[user1 ]$  date
# Exécution normale de la commande "date"

[user1 ]$  `date`
# Erreur de syntaxe. La commande "date" est sous-exécutée et son résultat est alors interprété par le
shell dans le contexte où la sous-exécution s'est produite. Il est peu probable que le premier mot de la
chaîne affichée soit une commande Unix valide.

[user1 ]$  `echo date`
# Correct, mais inutile. La commande "echo date" est sous-exécutée et son résultat (en l'occurrence le
mot "date") est alors interprété par le shell dans le contexte ou la sous-exécution s'est produite. Ici, la
chaîne "date" produite par la sous-exécution correspond à une commande Unix valide et comme son
contexte la place en début de ligne, le shell interprétera cette chaîne comme une instruction et
exécutera celle-ci. On aura donc comme résultat final l'exécution de la commande "date". Cette
remarque sera valable chaque fois que le programmeur sera tenté de faire "`echo commande
quelconque`".
```

Remarques :

- Il n'est pas possible d'imbriquer des "niveaux" de sous-exécution comme on serait tenté de faire par la syntaxe "*var=`echo `pwd``*". En effet, le premier accent grave correspond au début de la sous-exécution, et le second correspond à la fin de celle-ci.
- La sous-exécution peut-être aussi obtenue en "Korn Shell" et en "Bourne again Shell" par la syntaxe "*$(commande*)". Cette syntaxe permet les sous-niveaux "*$($(commande))*», mais n'est pas compatible avec le "Bourne Shell".
- L'excellente compréhension de ce mécanisme est nécessaire à la maîtrise du langage Shell.

5. LES PARAMÈTRES

Un paramètre, appelé aussi "argument", est un élément (chaîne de caractères) situé entre le nom du programme et la touche "Entrée" qui active le programme. Il s'agit en général d'éléments que le programme ne connaît pas à l'avance et dont il a évidemment besoin pour travailler. Ces éléments peuvent être nécessaires au programme pour son bon fonctionnement, ou facultatifs pour lui demander un travail plus précis ou plus rapide, c'est-à-dire un travail "optionnel".
Ils constituent généralement un substitut plus avantageux qu'une saisie en "interactif», car le programme n'a alors pas besoin, pour son traitement, de la présence d'un utilisateur qui répondra à ses questions. Celles-ci sont déjà prévues par les "arguments" que le programme connaît, et il les utilisera au moment opportun.

Exemple :

```
[user1 ]$  cp fic1 fic2              # Programme "cp", argument1 "fic1", argument2 "fic2"
[user1 ]$  ls –l                 # Programme "ls", argument1 "-l"
[user1 ]$  cd                    # Programme "cd" sans argument
```

5.1. Récupération des paramètres

Dans un script, les paramètres ou arguments, positionnés par l'utilisateur exécutant le script, sont **automatiquement et toujours stockés** dans des zones appelées "variables automatiques". Ces variables sont :

- $0 : nom du script. Le contenu de cette variable est invariable. Il peut-être considéré comme étant "à part" du reste des arguments.
- $1 : argument placé en première position derrière le nom du script lors du lancement de celui-ci.
- $2… $9 : argument placé en seconde… neuvième position derrière le nom du script etc.
- $# : nombre d'arguments passés au programme
- $* : liste de tous les arguments

Exemple :

```
[user1 ]$  vi prog
[user1 ]$  chmod a+x prog
[user1 ]$  ./prog
./prog
0
[user1 ]$  ./prog a b c d e f g h i j k l
m
./prog
a
```

```
b
e
13
a b c d e f g h i j k l m
```

Contenu du script "prog" :

```
#!/bin/sh
echo $0          # Affichage nom du script
echo $1          # Affichage argument n° 1
echo $2          # Affichage argument n° 2
echo $5          # Affichage argument n° 5
echo $#          # Affichage nb
d'arguments
echo $*          # Affichage tous
arguments
```

5.2. Décalage des paramètres

Syntaxe :

```
shift [n]
```

Comme on peut le remarquer, le programmeur n'a accès de façon individuelle qu'aux variables "*$1*" à "*$9*". Si le nombre de paramètres dépasse neuf, ils sont pris en compte par le système, mais le programmeur n'y a pas accès de manière individuelle. Bien entendu, il peut y accéder en passant par la variable "*$**», mais il devrait alors se livrer à des manipulations difficiles d'extraction de chaînes. Ainsi, la commande "***echo $10***" produira l'affichage de la variable "*$1*" suivi du caractère "*0*".
Remarque : La syntaxe "***echo ${10}***" fonctionne en "Korn Shell" et "Bourne again Shell», mais pas en "Bourne Shell".
Il existe néanmoins en "Bourne Shell" un moyen d'accéder aux arguments supérieurs à neuf. Seulement, cela implique la perte de l'argument n° 1. Il s'agit de l'instruction "***shift [n]***", "n" étant facultativement positionné à "*1*" s'il n'est pas renseigné. Cette instruction produit un décalage des paramètres vers la gauche de "*n*" positions. Dans le cas de "*shift*" ou *"shift 1*", le contenu de "*$1*" disparaît pour être remplacé par celui de "*$2"*; celui de "*$2*" fait de même pour recevoir le contenu de "*$3*", etc. jusqu'à "*$9*" qui reçoit le contenu du dixième argument.
De plus, les variables "*$#*" et "*$**" sont aussi modifiées pour correspondre à la nouvelle réalité. Seule la variable "*$0*" reste invariante. Ainsi, une simple boucle testant la valeur décroissante de "*$#*" en utilisant l'instruction "*shift*" permet d'accéder individuellement à tous les arguments.
Remarque : Si la variable "*$1*" doit être malgré tout conservée, rien n'empêche de la récupérer dans une variable "personnelle" avant de perdre son contenu suite au "***shift***".

5.3. Ré affectation volontaire des paramètres

Syntaxe :

```
set valeur1 [valeur2 …]
```

L'instruction "***set valeur1 [valeur2 …]***" permet d'affecter les valeurs demandées aux variables "*$1*", "*$2*", …, "*$9*" au mépris d'un contenu éventuellement déjà présent. L'ensemble des variables est affecté par l'instruction, c'est-à-dire que les variables non remplies par une valeur sont alors vidées de leur contenu précédent.
De plus, les variables "*$#*" et "*$**" sont aussi modifiées pour correspondre à la nouvelle réalité. Comme précédemment, la variable "*$0*" n'est pas modifiée.

Remarque : Rien n'oblige les valeurs placées après "*set*" d'être des chaînes figées. Il est donc possible d'y inclure des variables ; ou des sous-exécutions de commandes. Par contre, l'instruction "***set***" est la seule permettant de modifier les variables "*$1*", "*$2*", etc. Autrement dit, on ne peut pas modifier ces variables par l'instruction d'affectation "*variable=valeur*".

Exemple :

```
#!/bin/sh
set toto titi tata          # Affectation de différentes chaînes dans $1, $2, $3
set `date`                  # Affectation du résultat de la commande "date"
set $2                      # Affectation du contenu de la variable "$2" dans $1
set `pwd` `date`            # Affectation du résultat affiché par les deux commandes
```

5.4. Le séparateur de champs internes

Syntaxe :

```
IFS=chaîne
```

Lorsque l'instruction "*set valeur1 [valeur2 …]*" est exécutée, le Shell arrive à isoler et déconcaténer les différentes valeurs dans les différentes variables "*$1*", "*$2*", etc., grâce à la variable d'environnement "***IFS***" (*Internal Field Separator* en majuscules) qui contient le ou les caractères devant être utilisés pour séparer les différentes valeurs ("espace" par défaut).
Une modification du contenu de cette variable permet d'utiliser un (ou plusieurs) autre caractère pour séparer des valeurs avec la commande "*set*".

Remarque : La variable "***IFS***" étant très importante pour l'analyseur syntaxique du Shell, il est conseillé de la sauvegarder avant de la modifier pour pouvoir la restaurer dans son état original après l'avoir utilisée.

Exemple :

```
#!/bin/sh
chaîne="toto:titi/tata"     # Préparation de la chaîne à affecter
old=$IFS                    # Sauvegarde de la variable IFS
IFS=:/                      # L'IFS prend le caractère ":" ou le caractère "/"
set $chaîne                 # Déconcaténation de la chaîne suivant l'IFS
IFS=$old                    # Récupération de l'ancien IFS
echo $3                     # Affichage du troisième argument ("tata")
```

6. NEUTRALISATION DES MÉTACARACTÈRES

Certains caractères alphanumériques du Shell ont une signification particulière.

Exemple : Essayer de faire afficher à l'écran

```
* (une étoile)
```

```
Il a crié "assez"; mais cela ne suffisait pas
L'habit à $100 ne fait pas le moine
\____/  >- (un bâtiment de combat et sa torpille)
```

Les problèmes rencontrés pour afficher ces quelques lignes proviennent du caractère ***étoile***, ***parenthèse ouvrante***, ***parenthèse fermante***, ***guillemet***, ***point-virgule***, ***dollar***, ***apostrophe***, ***backslash*** et ***supérieur*** qui ont une signification spéciale en Shell. On les appelle "*méta caractères*"**.** Afin de les utiliser tels quels par le langage, il faut y appliquer un "mécanisme de neutralisation de leur interprétation". C'est-à-dire qu'il faut demander au langage de ne pas les transformer en autre chose que ce qu'ils sont.

6.1. Rappel sur les méta caractères

Les méta caractères suivants sont utilisés lorsque l'on désire référencer un fichier sans connaître exactement son nom :

Méta caractère	**Signification**
*****	Toute chaîne de caractère, même une chaîne vide
?	Un caractère quelconque, mais présent
[xyz]	Tout caractère correspondant à l'un de ceux contenus dans les crochets
[x-y]	Tout caractère compris entre "*x*" et "*y*"
[!x-y]	Tout caractère qui n'est pas compris entre "*x*" et "*y*"

Les méta caractères suivants sont utilisés dans le Shell :

Méta caractère	**Signification**
$	Contenu d'une variable
"…"	Neutralisation de certains méta caractères
'…'	Neutralisation de tous les méta caractères
\	Neutralisation de tout méta caractère qui le suit
(…)	Groupement de commandes
;	Séparateur de commande (permets d'en placer plusieurs sur une ligne)
<	Redirection en entrée à partir d'un fichier
<<	Redirection en entrée à partir des lignes suivantes
>	Redirection en sortie vers un fichier
>>	Redirection en sortie vers un fichier en mode "ajout"
|	Redirection vers une commande (pipe mémoire)

6.2. Le "backslash"

Le méta caractère "***backslash***" a pour effet de neutraliser tout méta caractère qui le suit, quel qu'il soit. Mais il n'en neutralise qu'un seul.

Exemple : Solution avec le méta caractère "*backslash*"

```
[user1 ]$  echo \* \(une étoile\)
[user1 ]$  echo Il a crié \"assez\"\; mais cela ne suffisait pas
[user1 ]$  echo L\'habit à \$100 ne fait pas le moine
[user1 ]$  echo \\____/   \>- \(un bâtiment de combat et sa torpille\)
```

6.3. L'apostrophe ou "*quote simple*"

Le méta caractère "***apostrophe***" appelé aussi "***quote simple***" a pour effet de neutraliser tous les méta caractères situés après, sauf lui-même. Cette exception permet ainsi d'avoir un début et une fin de zone de neutralisation.
Mais on ne peut pas, avec cette option, neutraliser le méta caractère "***apostrophe***" puisqu'il marque la fin de la zone. Et le méta caractère "***backslash***" qui pourrait neutraliser cet inconvénient ne fonctionnerait pas puisqu'il serait lui-même neutralisé.

Exemple : Solution avec le méta caractère "*quote simple*"

```
[user1 ]$  echo '* (une étoile)'
[user1 ]$  echo 'Il a crié "assez"; mais cela ne suffisait pas'
[user1 ]$  ✯ (à cause de l'apostrophe)
[user1 ]$  echo '\____/   >- (un bâtiment de combat et sa torpille)'
```

6.4. Le double guillemet

Le méta caractère "***double guillemet***" a pour effet de neutraliser tous les méta caractères situés après, sauf le méta caractère "***dollar***" (qui permet d'accéder au contenu d'une variable), le méta caractère "***accent grave***" (qui permet la sous-exécution), le méta caractère "***backslash***" s'il est suivi d'un méta caractère de cette liste (cela permet donc d'afficher ledit méta caractère) et le méta caractère "***double guillemet***". Cette dernière exception permet ainsi d'avoir un début et une fin de zone de neutralisation. Et on peut, avec cette option, neutraliser le méta caractère " ***double guillemet***" puisqu'il fait partie de la liste ; donc qu'il est neutralisable par le méta caractère "***backslash***".

Exemple : Solution avec le méta caractère "*double guillemet***"**

```
[user1 ]$  echo "* (une étoile)"
[user1 ]$  echo "Il a crié \"assez\"; mais cela ne suffisait pas"
[user1 ]$  echo "L'habit à \$100 ne fait pas le moine"
[user1 ]$  echo "\____/   >- (un bâtiment de combat et sa torpille)"
```

7. LES CONTRÔLES BOOLÉENS

7.1. Introduction

Le Shell étant un langage, il permet l'utilisation de contrôles "*vrai/faux*" appelés aussi "*booléens*". Chaque commande, chaque programme exécuté par Unix, donc par le Shell, lui retransmet en fin d'exécution un ***code de retour*** appelé aussi ***code de statut***.
La convention qui a été établie veut que si la commande s'est bien exécutée, le code de statut ait pour valeur ***zéro***. Par contre, si la commande a eu un problème dans son exécution (pas de droit d'accès, pas de fichier à éditer, etc.), son code de statut aura une valeur ***différente de zéro***. En effet, il existe toujours plusieurs raisons qui peuvent faire qu'un programme ne s'exécute pas ou mal ; par contre il n'y a qu'une seule raison qui fait qu'il s'exécute correctement.

Cette convention ayant été établie, le Shell considérera alors un programme de statut "***0***" comme étant "***vrai***" ; et un programme de statut "différent ***de 0***" comme étant "***faux***". Grâce à cette convention, l'utilisateur peut programmer de manière booléenne.
Remarque : Une convention n'étant pas une obligation, rien n'empêche un programmeur de faire renvoyer un statut toujours à zéro ou toujours différend de zéro. S'il fait cela, son programme sera toujours interprété comme "vrai" ou comme "faux" et le programme ainsi créé n'aura pas la faculté d'être vérifié par le Shell qui le lance.

Le statut de la dernière commande exécutée se trouve dans la variable "***$?***" du processus ayant lancé la commande. On peut donc visualiser facilement une commande "vraie" ou "fausse" avec la commande "***echo $?***". Attention, visualiser un statut de cette manière perd ce dernier puisque la variable "***$?***" prend la valeur de la dernière commande exécutée (ici "*echo*").

Attention : Il ne faut pas confondre "affichage" (ce que la commande affiche à l'écran) et "statut" (état de son exécution). Une commande peut ne rien afficher, mais renvoie toujours une valeur zéro (statut "vrai") ou différente de zéro (statut "faux").
Exemple :

```
[user1 ]$  cd /tmp                          # La commande n'affiche rien, mais elle réussit
[user1 ]$  echo $?
0

[user1 ]$  rm /tmp                          # J'efface un répertoire sans avoir les droits
rm: 0653-603 Cannot remove directory /tmp
[user1 ]$  echo $?
2

[user1 ]$  rm /tmp 2>/dev/null              # Même si je redirige les erreurs…
[user1 ]$  echo $?
2
[user1 ]$  echo $?                          # J'affiche le statut de la dernière commande
0
```

7.2. La commande "*test*"

La commande "***test***" est une… commande. Celui qui sait utiliser "***echo***" sait utiliser "***test»,*** car il s'agit d'un même item appelé "*commande Unix*". À ce titre, "***test***" renvoie donc un statut vrai ou faux, mais il n'affiche rien à l'écran. Il faut donc, pour vérifier le résultat d'un test, vérifier le contenu de la variable "***$?***".
Cette commande a pour but de vérifier (tester) la validité de l'expression demandée, selon les options considérées. À ce titre, elle permet de vérifier l'état des fichiers, comparer des variables, etc.
La syntaxe générale d'une commande test est "***test expression***" ; mais elle peut aussi être "***[expression]***" (à condition de ne pas oublier les espaces séparant l'expression des crochets). Ici, les crochets ne signifient pas "élément optionnel», mais bien "crochets".

a) Test simple sur les fichiers

Syntaxe :

```
test option "fichier"
```

Option	Signification
-s	fichier "non-vide"
-f	fichier "ordinaire"
-d	fichier "répertoire"
-b	fichier "spécial" mode "bloc"
-c	fichier "spécial" mode "caractère"
-p	fichier "tube"
-L	fichier "lien symbolique"
-S	fichier "socket" ("Korn Shell" et "Bourne again Shell")
-r	fichier a le droit en lecture
-w	fichier a le droit en écriture
-x	fichier a le droit en exécution
-u	fichier a le "setuid"
-g	fichier a le "setgid"
-k	fichier a le "sticky bit"
-t [n]	fichier "n" est associé à un terminal (par défaut, "n" vaut "1")
-e	fichier existe quel que soit son type ("Bourne again Shell")
-O	fichier m'appartient ("Korn Shell" et "Bourne again Shell")
-G	fichier appartient à mon groupe ("Korn Shell" et "Bourne again Shell")

b) Test complexe sur les fichiers (uniquement en "Korn Shell" et en "Bourne again Shell")

Syntaxe :

```
test "fichier1" option "fichier2"
```

Option	Signification
-nt	fichier1 plus récent que fichier2 (date de modification)
-ot	fichier1 plus vieux que fichier 2 (date de modification)
-ef	fichier1 lié à fichier2 (même numéro d'inode sur même système de fichiers)

c) Test sur les longueurs de chaînes de caractères

Syntaxe :

```
test option "chaîne"
```

Option	Signification
-z	chaîne de longueur nulle
-n	chaîne de longueur non nulle

d) Test sur les chaînes de caractères

Syntaxe :

```
test "chaîne1" option "chaîne2"
```

Option	Signification
=	chaîne1 identique à chaîne2
!=	chaîne1 différente de chaîne2

e) Test sur les numériques

Syntaxe :

```
test nb1 option nb2
```

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

f) Connecteurs d'expression

Les connecteurs permettent de composer des expressions plus complexes

Option	Signification
-a	"ET" logique
-o	"OU" logique
!	"NOT" logique
(…)	Groupement d'expressions

Attention : L'emploi des doubles guillemets dans les syntaxes faisant intervenir des chaînes est important surtout lorsque ces chaînes sont prises à partir de variables. En effet, il est possible d'écrire l'expression sans doubles guillemets, mais si la variable est vide ou inexistante, l'expression reçue par le shell sera bancale et ne correspondra pas au schéma attendu dans la commande "test".

```
test $a = bonjour              # Si a est vide, le shell voit test = bonjour (incorrect)
test "$a" = "bonjour"          # Si a est vide, le shell voit test "" = "bonjour" (correct)
```

8. LES STRUCTURES DE CONTRÔLES

8.1. Introduction

Comme tout langage évolué, le *Shell* permet des structures de contrôles. Ces structures sont :

- l'alternative simple ("*&&…*", "*||…*")
- l'alternative complexe ("*if…*")
- le branchement sur cas multiples ("*case…*")
- la boucle ("*while…*", "*until…*", "*for…*")

8.2. L'alternative simple

Syntaxe :

```
cde1 && cde2

cde1 || cde2
```

La première syntaxe correspond à un "cde1 **ET** cde2". Le Shell n'exécutera la commande n° 2 que si la commande n° 1 a renvoyé "**vrai**".
La seconde syntaxe correspond à un "cde1 **OU** cde2". Le Shell n'exécutera la commande n° 2 que si la commande n° 1 a renvoyé "**faux**".
Il est possible d'enchaîner les opérateurs par la syntaxe "***cde1 && cde2 || cde3***"

Exemple :

```
[user1 ]$  test –d /tmp && echo "/tmp est un répertoire"

[user1 ]$  test –w /etc/passwd || echo "Pas le droit d'écrire dans /etc/passwd"

[user1 ]$  test "$LOGNAME" = "root" && echo "Je suis root" || echo "Je ne suis pas root"
```

8.3. L'alternative complexe

Syntaxe :

```
if liste de commandes
then
                    commande1
                    [ commande2 …]
[ else
                    commande3
                    [ commande4 …] ]
fi
```

La structure "***if…then…[else]…fi***" évalue toutes les commandes placées après le "***if»,*** mais ne vérifie que le code de retour de la dernière commande de la liste. Dans le cas où le programmeur voudrait placer plusieurs commandes dans la "liste de commandes", il doit les séparer par le caractère "**point-virgule**" qui est un séparateur de commandes *Shell*. Dans la pratique, cette possibilité est très rarement utilisée, un script étant plus lisible si les commandes non vérifiées par le "if" sont placées avant celui-ci.
Si le résultat de la dernière commande est "**vrai**", le Shell ira exécuter l'ensemble des instructions placées après dans le bloc "*then*" ; sinon, il ira exécuter l'ensemble des instructions placées dans le bloc "*else*" si celui-ci existe.
Dans tous les cas, le *Shell* ira exécuter les instructions éventuellement placées derrière le mot-clef "***fi»,*** car celui-ci termine le "***if***".

Il est possible d'imbriquer plusieurs blocs "*if…fi*" à condition de placer un mot-clef "***fi***" pour chaque Mot-clef "***if***".

On peut se permettre de raccourcir une sous-condition "*else if…*" par le mot-clef "***elif***". Dans ce cas, il n'y a plus qu'un seul "***fi***" correspondant au "***if***" initial.

Exemple avec des "if imbriqués" :

```
#!/bin/sh
echo "Entrez un nombre"
read nb
if test $nb –eq 0      # If n° 1
then
                    echo "C'était zéro"
else
                    if test $nb –eq 1       # If n° 2
                    then
                        echo "C'était un"
                    else
                        echo "Autre chose"
                    fi                      # Fin n° 2
fi                                  # Fin n° 1
```

Exemple avec des "elif" :

```
#!/bin/sh
echo "Entrez un nombre"
read nb
if test $nb –eq 0      # If n° 1
then
                    echo "C'était zéro"
elif test $nb –eq 1    # Sinon si
then
                    echo "C'était un"
else
                    echo "Autre chose"
fi                                  # Fin n° 1
```

8.4. Le branchement à choix multiple

Syntaxe :

```
case chaîne in
                    val1)
                        commande1
                        [ commande2 …]
                        ;;
                    [val2)
                        commande3
                        [ commande4 …]
                        ;;]
esac
```

La structure "***case***" évalue la valeur principale en fonction des différents choix proposés. À la première valeur trouvée, les instructions correspondantes sont exécutées.
Le double "***point-virgule***" indique que le bloc correspondant à la valeur testée se termine. Il est donc obligatoire… et facultatif si ce bloc est le dernier à être évalué.
La valeur évaluée et/ou les valeurs de choix peuvent être construites à partir de variables ou de sous-exécutions de commandes. De plus, les valeurs de choix peuvent utiliser les constructions suivantes :

Construction	**Signification**
[x-y]	La valeur correspond à tout caractère compris entre "*x*" et "*y*"
[xy]	La valeur testée correspond à "*x*" ou "*y*"
xx|yy	La valeur correspond à deux caractères "*xx*" ou "*yy*"
?	La valeur testée correspond à un caractère quelconque
*****	La valeur testée correspond à toute chaîne de caractères (cas "autre cas")

Exemple :

```
#!/bin/sh
# Saisie
echo "Entrez un nombre"
read nb

# Evaluation
case $nb in
                    0) echo "$nb vaut zéro";;
                    1|3|5|7|9) echo "$nb est impair";;
                    2|4|6|8) echo "$nb est pair";;
                    [1-9][0-9]) echo "$nb est supérieur ou égal à 10 et inférieur à 100";;
                    *) echo "$nb est un nombre trop grand pour être évalué"
esac
```

8.5. La boucle sur condition

Syntaxe :

```
while liste de commandes
do
                    commande1
                    [ commande2 …]
done
```

```
until liste de commandes
do
                commande1
                [ commande2 …]
done
```

La boucle "***while do…done***" exécute une séquence de commandes tant que la dernière commande de la "*liste de* ***commandes*****" est "*****vrai*****" (statut** égal à zéro).
La boucle "***until do…done***" exécute une séquence de commandes tant que la dernière commande de la "*liste de commandes*" est "*faux*" (statut différent de zéro).

8.6. Les commandes "*true*" et "*false*"

Syntaxe :

```
true

false
```

Ces commandes n'ont d'autre but que de renvoyer un état toujours "***vrai***" pour "*true*" ou toujours "***faux***" pour "*false*". Leur utilité peut sembler obscure, mais prend de l'importance quand la nécessité se fait sentir de programmer une "**boucle infinie**".

8.7. La boucle sur liste de valeurs

Syntaxe :

```
for var in valeur1 [valeur2 …]
do
                commande1
                [ commande2 …]
done
```

La boucle "***for… do…done***" va boucler autant de fois qu'il existe de valeurs dans la liste. À chaque tour, la variable "***var***" prendra séquentiellement comme contenu la valeur de la liste correspondant au numéro de tour.
Les valeurs de la liste peuvent être obtenues de différentes façons (variables, sous-exécutions, etc.).
La syntaxe "***in valeur1 …***" est optionnelle. Dans le cas où elle serait omise, les valeurs seraient prises dans la variable "***$****" contenant les arguments passés au programme.

Dans le cas où une valeur contiendrait un méta caractère de génération de nom de fichier ("***étoile***", "***point d'interrogation***", etc.), le *Shell* examinerait alors les fichiers présents dans le répertoire demandé et remplacerait le méta caractère par le ou les fichiers correspondants à la recherche.

Exemple :

```
#!/bin/sh
# Boucle sur tous les fichiers du répertoire courant et du répertoire "/tmp"
for fic in * /tmp/*
do
```

```
                echo "Le fichier se nomme '$fic'"
done

# La même boucle, mais avec sous-exécution
for fic in `ls` `ls /tmp`
do
                echo "Le fichier se nomme '$fic'"
done
```

8.8. Interruption d'une ou plusieurs boucles

Syntaxe :

```
break [n]

continue [n]
```

L'instruction "***break [n]***" va faire sortir le programme de la boucle numéro "***n***" ("***1***" par défaut). L'instruction passera directement après le "***done***" correspondant à cette boucle.
L'instruction "***continue [n]***" va faire repasser le programme à l'itération suivante de la boucle numéro "***n***" ("***1***" par défaut). Dans le cas d'une boucle "***while***" ou "***until***", le programme repassera à l'évaluation de la condition. Dans le cas d'une boucle "***for***", le programme passera à la valeur suivante.
La numérotation des boucles s'effectue à partir de la fin de boucle la plus proche de l'instruction "***break***" ou "***continue***", numérotée "***1***". Chaque boucle englobant la précédente se voit affecter un numéro séquentiel. Le programmeur peut choisir de sauter directement la boucle numérotée "***n***" en mettant la valeur désirée derrière l'instruction "***break***" ou "***continue***".

Remarques :

- L'utilisation de ces instructions est contraire à la philosophie de la "*programmation structurée*". Il incombe donc à chaque programmeur de toujours réfléchir au bien-fondé de leurs mises en application.
- Contrairement aux abus de langages couramment employés, la structure "***if… fi***" n'est pas une boucle.

Exemple :

```
#!/bin/sh
while true
do
                echo "Boucle infinie"
                echo "Entrez une valeur (0 pour quitter)"
                read val
                if test $val –eq 0
                then
                        break                   # On sort de la boucle "while" et pas seulement du
"if"
                fi
done
```

8.9. Interruption d'un programme

Syntaxe :

```
exit [n]
```

L'instruction "***exit [n]***" met immédiatement fin au Shell dans lequel cette instruction est exécutée. Si cette dernière est appelée à partir du Shell courant, elle déconnecte l'utilisateur qui l'utilise.
Le paramètre "***n***" facultatif vaut "***0***" par défaut, mais ne peut pas dépasser "***255***". Il correspond au "***statut***" du Shell dans lequel se trouve l'instruction "***exit***"… et est, de ce fait, retransmis à la variable "***$?***" du processus ayant appelé ce Shell (processus père). Cette instruction peut donc rendre un script "vrai" ou "faux" selon les conventions du *Shell*.

8.10. Le générateur de menus en boucle ("Korn Shell" et "Bourne again Shell")

Syntaxe :

```
select var in chaîne1 [chaîne2 …]
do
                    commande1
                    [ commande2 …]
done
```

La structure "***select…do…done***" proposera à l'utilisateur un menu prénuméroté commençant à "*1*". À chaque numéro sera associé une chaîne prise séquentiellement dans les chaînes de la liste. Il lui sera aussi proposé de saisir un des numéros du menu (le prompt de saisie provenant de la variable "*PS3*").
Après la saisie, la valeur correspondante au numéro demandé sera stockée dans la variable "*var*". Il appartient alors au programmeur de l'évaluer correctement ("*if…fi*" ou "*case…esac*") pour la suite de son programme. Dans le cas où l'utilisateur demanderait un numéro qui n'est pas dans la liste, la variable "*var*" reçoit une chaîne vide. Cependant, la variable de statut "*$?*" n'est pas modifiée par ce choix erroné.

Comme pour la boucle "*for*", les valeurs de la liste peuvent être obtenues de différentes façons (variables, sous-exécutions, etc.). La syntaxe "***in valeur1 …***" est optionnelle. Dans le cas où elle serait omise, les valeurs seraient prises dans la variable "***$****" contenant les arguments passés au programme. Dans le cas où une valeur contiendrait un méta caractère de génération de nom de fichier ("***étoile***", "***point d'interrogation***", etc.), le *Shell* examinerait alors les fichiers présents dans le répertoire demandé et remplacerait le méta caractère par le ou les fichiers correspondants à la recherche.

Remarque : La phase "menu + choix" se déroule de façon infinie. Il est donc nécessaire de programmer l'interruption de la boucle sur une valeur particulière de la variable "*var*" en utilisant une des instructions "*break*" ou "*exit*".

9. LES FONCTIONS

9.1. Introduction

Syntaxe :

```
nom_de_fonction()
{
                commande1
                [ commande2 …]
}
…
nom_de_fonction
```

Une fonction permet de regrouper des instructions fréquemment employées dans un ensemble portant un nom.
Ce nom utilisé ensuite comme commande Unix exécutera l'ensemble des instructions contenues dans la fonction. Il est intéressant de noter qu'une fonction ressemble en cela à un script Shell qui regroupe dans un seul fichier un ensemble de commandes fréquemment utilisées.
Chaque fonction doit être déclarée (créée) avant d'être utilisée. Une fois écrite, elle peut être utilisée comme n'importe quelle commande Unix. Les commandes utilisées dans la fonction peuvent être elles-mêmes d'autres fonctions (qui auront par voie de conséquence été écrites avant leur utilisation).

Remarque : Il n'y a pas recopie de variable pour une fonction. Autrement dit, modifier une variable dans une fonction revient à la modifier réellement dans le programme ; ce qui n'est pas le cas entre une variable extérieure et une variable intérieure à un script ; comme il l'a été expliqué au paragraphe sur la visibilité des variables.

Exemple :

```
#!/bin/sh
# Fonction qui modifie une variable
modif()
{
                i=5                             # Modification de "i"
}

# Programme principal
i=0                                             # Initialisation de "i"
echo "i=$i"                                     # Affichage de "i"
modif                                           # Appel de la fonction
echo "i=$i"                                     # Affichage de "i" modifié
```

9.2. Passage de valeurs

Syntaxe :

```
nom_de_fonction()
{
                    echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 $* $#
}
…
nom_de_fonction paramètre1 [paramètre2 …]
```

Comme pour un script Shell, une fonction peut avoir besoin de valeurs non connues à l'avance. De la même manière, ces valeurs lui seront passées comme "***argument***" ou "***paramètre***" lors de l'appel de la fonction, qui les récupèrera dans les variables connues "***$0***" (nom de la fonction), "***$1***" (premier paramètre), etc.

Remarque : Dans le cas où la fonction aurait à récupérer le contenu d'une variable, il est possible de passer celle-ci à la fonction comme paramètre. Mais ceci est inutile puisque la variable est de toute façon connue de la fonction et modifiable par celle-ci.

9.3. Retour de fonction

Syntaxe :

```
nom_de_fonction()
{
                    return [n]
}
…
nom_de_fonction
echo $?
```

L'instruction "***return [n]***" met immédiatement fin à la fonction dans laquelle cette instruction est exécutée.
Le paramètre "***n***" facultatif vaut "***0***" par défaut, mais ne peut pas dépasser "***255***". Il correspond au "***statut***" de la fonction et est, de ce fait, retransmis à la variable "***$?***" du programme ayant appelé cette fonction. Cette instruction peut donc rendre une fonction "vrai" ou "faux" selon les conventions du *Shell*. Ce comportement est similaire à celui de l'instruction "*exit»,* mais ; au contraire de cette dernière, ne fait pas sortir su programme.

Remarque : Il ne faut pas confondre la notion "*retour de fonction*" en *Shell* et la notion de "*valeur calculée par la fonction*" telle qu'on la comprend dans d'autres langages, comme le C ou le PASCAL. Cette dernière notion ne peut-être recréée en *Shell* que par l'utilisation d'un "***echo***" unique et final dans la fonction ; ce qui permet au programmeur de récupérer dans une variable ce que la fonction affiche en utilisant les "*accents graves*" de la sous-exécution.

10. LES COMPLÉMENTS

Le chapitre précédent marquait la fin de l'apprentissage du *Shell* de base. Il reste néanmoins un certain manque dans le langage, comme "incrémenter une variable", "récupérer des informations dans un fichier", etc.
Tous ces manques ont été comblés par les nombreux programmeurs du monde Unix par la création de diverses commandes répondant aux besoins.

10.1. La commande "expr"

Elle répond à beaucoup de besoins dans l'analyse des "**expressions régulières**". Avant d'examiner les possibilités de cette commande, on va d'abord parler de son statut d'exécution, car il reste indépendant du but de l'emploi de la commande.
Statut de la commande :

- si l'expression est valide et si la valeur que "*expr*" affiche n'est pas "*0*", le statut est "***0***"
- si l'expression est valide et si et si la valeur que "*expr*" affiche est "*0*", le statut est "***1***"
- si l'expression est invalide, le statut vaut en général "***2***"

Remarque : Cette commande ne peut pas être utilisée comme booléen *vrai/faux* puisque le Shell la considérera comme "*faux*" si elle affiche "*0*". Mais rien n'empêche d'analyser la valeur de son statut "*$?*".

a) Arithmétique

Syntaxe :

```
expr opérande opérateur opérande [opérateur opérande]
```

La commande va réaliser l'opération mathématique demandée sur les opérateurs cités et afficher le résultat sur la sortie standard (l'écran). Les opérandes ne peuvent être que des nombres entiers et le résultat de l'opération sera lui aussi sous forme d'un nombre entier.
La priorité mathématique des opérateurs est respectée (multiplication, division et modulo prioritaires sur addition et soustraction). Voici leur symbolique :

- + (addition)
- - (soustraction)
- * (multiplication ; attention l'étoile est un méta caractère Shell donc il doit être neutralisé)
- / (division euclidienne ou entière)
- % (modulo, qui correspond au reste d'une division entière)

Exemple : Incrémenter une variable pouvant jouer le rôle d'un compteur

```
[user1 ]$  i=5                          # Affectation initiale
[user1 ]$  i=`expr $i + 1`              # Utilisation du résultat de l'affichage de la commande
[user1 ]$  echo $i                      # Affichage final
```

b) Comparaison

Syntaxe :

```
expr opérande comparaison opérande
```

La commande va réaliser la comparaison mathématique demandée sur les opérateurs cités et afficher le résultat sur la sortie standard (l'écran). Les opérateurs ne peuvent être que des nombres entiers et le résultat de la comparaison sera "***0***" si la comparaison est "faux", "***1***" si elle est "vrai".
Les comparaisons possibles sont :

- = (égal)
- != (différent)
- \< (strictement inférieur à; attention le signe "inférieur" est un méta caractère de redirection)
- \<= (inférieur ou égal à; attention le signe "inférieur" est un méta caractère de redirection)
- \> (strictement supérieur à; attention le signe "supérieur" est un méta caractère de redirection)
- \>= (supérieur ou égal à; attention le signe "supérieur" est un méta caractère de redirection)

c) Travail sur chaînes de caractères

Syntaxe :

```
expr chaîne : argument
```

La commande va indiquer si la chaîne demandée débute par l'argument indiqué. Si c'est le cas ; la commande affichera soit :

- le nombre de caractères de la chaîne qui correspondent à la totalité de l'argument
- la partie de l'argument demandé s'il a été mis entre parenthèses

Exemples :

```
[user1 ]$  expr "abcd" : "f"                # affiche "0" ("abcd" ne commence pas par "f")
[user1 ]$  expr "abcd" : "a"                # affiche "1" ("abcd" commence par "a")
[user1 ]$  expr "abcd" : "ab"               # affiche "2" ("abcd" commence par "ab")
[user1 ]$  expr "abcd" : "a\(bc\)"  # affiche "bc" ("abcd" commence par "abc" et "bc" est entre
parenthèses)
[user1 ]$  expr "abcd" : "abcde"    # affiche "0" ("abcd" ne commence pas par "abcde")
[user1 ]$  expr "abcd" : ".*"               # affiche "4" (méta caractère "*" permettant d'avoir la
longueur de la chaîne)
```

10.2. La commande "grep"

Syntaxe :

```
grep [option] expression [fichier1 …]
```

La commande "grep" (Global Regular Expression Printer) permet d'extraire et d'afficher toutes les lignes contenant l'expression demandée. Les lignes sont prises dans l'entrée standard (clavier), mais peuvent être cherchées dans un ou plusieurs fichiers.
L'expression à chercher peut être une simple chaîne de caractères ; ou bien composée de méta caractères spécifiques à la commande "*grep*" :

- accent circonflexe ("^") : placé en début d'expression, il indique à "*grep*" de ne chercher l'expression qu'en début de chaque ligne
- dollar ("***$***") : placé en fin d'expression, il indique à "*grep*" de ne chercher l'expression qu'en fin de ligne
- point ("**.**") : il permet de représenter un caractère quelconque, mais non nul dans l'expression
- étoile ("*****") : il indique que le caractère précédent peut se trouver entre zéro et un nombre infini de fois
- accolades("*{x,y}*") : elles permettent de spécifier que le caractère précédent doit être présent entre *x* et *y* fois
- crochets ("*[]*") : ils permettent de spécifier des classes de caractères recherchés

Les options modifient le fonctionnement de la commande (ne pas tenir compte des majuscules, n'afficher que le nom des fichiers contenant l'expression, n'afficher que le nombre de lignes contenant l'expression, etc.)

D'autres commandes similaires existent.

- fgrep (fast grep) : plus rapide en exécution, mais ne permettant pas de méta caractères dans l'expression à chercher
- egrep (ehanced grep) : élargit les possibilités de recherche en donnant accès à d'autres méta caractères pour spécifier l'expression
- awk : utilise un script de programmation dans son propre langage pour traiter les données entrantes

Statut de la commande :

- si la commande trouve l'expression demandée (elle affiche au moins une ligne), le statut est "***0***"
- si la commande ne trouve pas l'expression demandée (rien n'est affiché), le statut est différent de "***0***" (en général, il vaut "***1***")

Exemple :

```
[user1 ]$  cat /etc/passwd |grep "root"          # Extraction dans l'entrée standard

[user1 ]$  grep "root" /etc/passwd               # Extraction dans un fichier
```

10.3. La commande "cut"

Syntaxe :

```
cut –fn [-dc] [-s] [fichier1 …]

cut –cn [fichier1 …]
```

La commande "cut" (couper) est un filtre vertical qui sélectionne le n ième champ (option "***-f***" comme field) ou le n^{ième} caractère (option "***-c***") de chaque ligne. Les lignes sont prises dans l'entrée standard (clavier), mais peuvent être cherchées dans un ou plusieurs fichiers.
Les champs de l'option "***-f***" sont découpés suivant le caractère **tabulation**. Ce réglage par défaut peut être changé en mettant l'option "***-d***" pour spécifier le caractère de séparation de champs (délimiteur). Il peut alors être demandé d'ignorer les lignes ne contenant pas le délimiteur spécifié avec l'option "***-s***".

Statut de la commande : toujours "***0***" sauf en cas d'erreur de syntaxe

Exemple :

```
[user1 ]$  cat /etc/passwd |cut –f3-6 –d:          # Sélection du champ 3 au champ 6
[user1 ]$  cut –f1,6 –d: /etc/passwd                    # Sélection des champs 1 et 6
```

10.4. La commande "sort"

Syntaxe :

```
sort [-n] [-r] [-o output] [-k pos] [-tc] [fichier1 …]
```

La commande "sort" va trier les lignes de façon alphabétiquement croissante pour les afficher à l'écran. Les lignes sont prises dans l'entrée standard (clavier), mais peuvent être cherchées dans un ou plusieurs fichiers.
Le tri peut être inversé (option "***-r***"), les lignes triées sur plusieurs champs (option "***-k***"), le délimiteur de champs peut-être spécifié (option "***-t***") et les champs peuvent être considérés comme étant des nombres (option "***-n***").
Enfin, il est possible de spécifier le fichier dans lequel sera écrit le résultat du tri (option "***-o***") ce qui permet de demander à réécrire un fichier désormais trié sur lui-même (ce qui n'est pas possible avec un pipe).

Statut de la commande : toujours "***0***" sauf en cas d'erreur de syntaxe

Exemple :

```
[user1 ]$  cat /etc/passwd |sort –k3 –n –t:                  # Tri numérique sur le 3ième champ
[user1 ]$  sort –r –k4 –t: /etc/passwd –o /etc/passwd        # Tri inversé reversé dans le fichier
```

10.5. La commande "sed"

Syntaxe :

```
sed [-e script] [-f fichier_script] [fichier1 …]
```

La commande "sed" (Stream Editor) est un éditeur de flux. Elle permet de filtrer un flux de données au travers d'un ou plusieurs scripts basés sur l'éditeur "*ed*" (ex. : "s/x/y/g") pour avoir en sortie un flux de données modifiées. Le script peut être pris dans la ligne de commande (option "***-e***") ou dans un fichier externe (option "***-f***").
L'éditeur "ed" a été la base de travail qui a donné naissance à l'éditeur "vi".

Statut de la commande : toujours "***0***" sauf en cas d'erreur de syntaxe

Exemple :

```
[user1 ]$  cat /etc/passwd |sed –e "/root/d"                    # Suppression de ligne
[user1 ]$  sed –e "s/toto/titi/g" –e "s/tata/tutu/g" /etc/passwd  # Double substitution
```

10.6. La commande "tr"

Syntaxe :

```
tr [-c] [-s] [-d] chaîne1 chaîne2
```

La commande "tr" va transposer l'entrée standard où chaque caractère correspondant à un de ceux trouvés dans la chaîne 1 sera transformé en caractère pris dans la chaîne 2.
Il est possible de demander la suppression de chaque caractère de la chaîne 1 (option "***-d***") ; d'éliminer les caractères répétés (option "***-s***") et de complétion (option "***-c***").

Statut de la commande : toujours "***0***" sauf en cas d'erreur de syntaxe

Exemple :

```
[user1 ]$  cat /etc/passwd |tr "[a-z]" "[A-Z]"       # Transformation minuscules en majuscules
```

10.7. La commande "wc"

Syntaxe :

```
wc [-c] [-l] [-w] [fichier1 …]
```

La commande "wc" va compter le nombre de lignes, de mots et de caractères de l'entrée standard. Il est possible de ne demander que le nombre de lignes (option "***-l***") ; le nombre de mots (option "***-w***") ou le nombre de caractères (option "***-c***").

Statut de la commande : toujours "***0***" sauf en cas d'erreur de syntaxe

10.8. La commande "eval"

Syntaxe :

```
eval arguments
```

La commande "eval" va regrouper les arguments en une commande simple qui sera exécutée par le shell. Le premier mot des arguments doit donc être une commande valide.
La commande "eval" renverra le statut de la commande exécutée.
L'utilisation de la commande "eval" permet d'utiliser une indirection de variable puisqu'il y a deux fois interprétation ; une fois par la commande et une fois par le shell.

Statut de la commande : statut de la commande demandée

Exemple :

```
[user1 ]$  var=valeur
[user1 ]$  ptr=var
[user1 ]$ eval echo "\$$ptr"                  # Affichera "valeur"
```

10.9. La gestion de l'écran (codes "Escape")

Syntaxe :

```
echo code particulier

tput argument_tput
```

L'utilisation de certains codes appelés "codes Escape" permet d'influer sur l'affichage de l'écran. Ces codes portent ce nom, car ils commencent tous par la valeur "*033*" et le caractère "*Escape*" porte le numéro "*ascii*" 27 (33 en octal).
Bien souvent, ces codes varient en fonction de l'écran que l'on veut gérer, c'est pourquoi il vaut mieux utiliser la commande "*tput*" qui envoie elle-même les codes appropriés en fonction de l'écran utilisé.
Il vous est proposé ici une liste non exhaustive de certains codes avec leur signification.

Codes échappements	**Commande TPUT**	**Signification**
echo "\033[2J"	tput clear	efface l'écran
echo "\033[0m"	tput smso	aucun attribut (blanc sur noir)
echo "\033[1m"		gras
echo "\033[4m"		souligne
echo "\033[5m"	tput blink	clignote
echo "\033[7m"	tput rmso	vidéo inverse
echo "\033[8m"		invisible
echo "\033[30m"		noire (avant plan)
echo "\033[31m"		rouge
echo "\033[32m"		vert
echo "\033[33m"		jaune
echo "\033[34m"		bleu
echo "\033[35m"		magenta
echo "\033[36m"		cyan
echo "\033[37m"		blanc
echo "\033[40m"		noire (arrière plan)
echo "\033[41m"		rouge
echo "\033[42m"		vert
echo "\033[43m"		jaune
echo "\033[44m"		bleu
echo "\033[45m"		magenta
echo "\033[46m"		cyan
echo "\033[47m"		blanc
echo "\033[#A"		déplace le curseur de # ligne en haut
echo "\033[#B"		déplace le curseur de # ligne en bas
echo "\033[#C"		déplace le curseur de # colonne à droite
echo "\033[#D"		déplace le curseur de # colonne à gauche
echo "\033[s"		sauvegarde la position du curseur
echo "\033[u"		restaure la position du curseur
echo "\033[K"		efface la ligne courante

echo "\033[lig;colH\c"		positionne le curseur en lig - col
echo "\033[r" echo "\033[12;19’r’" echo "\033[?6h"		réinitialise les attributs définit une fenêtre lig 12 à 19 active la fenêtre
echo "\033[r"		désactive la fenêtre
echo "\033(B"		mode texte
echo "\033(0"		mode graphique

INDEX