NOTES DE COURS

INTRODUCTION AU BASH

Table des matières

1 BASH 4
1.1 DÉMARRAGE DU SHELL 4
1.2 LES SCRIPTS DE CONNEXION 4
1.3 PERSONNALISATION DES COMMANDES BASH 5
1.4 PERSONNALISATION DU LOGIN UTILISATEUR 5
2 FACILITÉS DE SAISIE DES COMMANDES 6
2.1 HISTORIQUE DES COMMANDES 6
2.2 LE CLIC-DROIT 6
2.3 L'OPÉRATEUR TILDE(~) 7
2.4 COMPLÉTER UNE COMMANDE 7
2.5 ENSEMBLE DE FICHIERS 8
2.6 CARACTÈRES SPÉCIAUX 9
2.7 GUILLEMETS (DOUBLE QUOTE) 10
2.8 APOSTROPHE (SINGLE QUOTE) 11
2.9 BACKSLASH \ 12
2.10 BACKQUOTE 13
3 LIGNE DE COMMANDES 14
3.1 ANALYSE DE LA LIGNE DE COMMANDE 14
3.2 ENCHAÎNEMENT INCONDITIONNEL DES COMMANDES 15
3.3 ENCHAÎNEMENT CONDITIONNEL DES COMMANDES 17
3.4 REDIRECTIONS DES ENTRÉES-SORTIES 18
3.5 SUBSTITUTION DE COMMANDE 20
4 PROGRAMMATION BASH 21
4.1 Introduction 21
4.2 CRÉATION D'UN SCRIPT 21
4.3 EXÉCUTION DU SCRIPT 22
4.4 MISE AU POINT (DÉBOGAGE) 23
4.5 ENTRÉES-SORTIES 24
5 LES VARIABLES BASH 25
5.1 VARIABLES USAGERS 25

5.2 VARIABLES D'ENVIRONNEMENT 27
5.3 VARIABLES SYSTÈME 28
5.4 Tableaux 29
5.5 PARAMÈTRES DE POSITION 30
6 COMMANDES IMPORTANTES 31
6.1 Commande shift 31
6.2 tr 31
6.3 set 32
6.4 eval 33
6.5 basename 34
6.6 seq 34
6.7 test 35
6.8 VALEUR DE RETOUR 35
6.9 TESTER UN FICHIER 36
6.10 TESTER UNE CHAÎNE 37
6.11 TESTER UN NOMBRE 38
6.12 OPÉRATIONS BOOLÉENNES 39
7 STRUCTURES DE CONTRÔLE 40
7.1 STRUCTURES CONDITIONNELLES 40
7.2 CONDITIONNELLES IMBRIQUÉES 41
7.3 CHOIX MULTIPLES (case) 43
8 STRUCTURES ITÉRATIVES 45
8.1 BOUCLE FOR 45
8.2 BOUCLE WHILE 47
8.3 SORTIE ET REPRISE DE BOUCLE 48
9 FONCTIONS 49
10 CALCUL SUR LES ENTIERS 50

1 BASH

Un interpréteur de commandes est un programme qui sert d'intermédiaire entre l'utilisateur et le système d'exploitation Sa tâche essentielle est l'exécution de commandes. Le Bash est un

Bash (Bourne-Again shell) est un interpréteur de commande Linux. C'est le shell Linux du projet GNU.

1.1 DÉMARRAGE DU SHELL

- Lors de la création d'un compte usager, un Shell lui est associé.
- Dans le fichier **/etc/passwd**, le dernier champ contient le nom du Shell associé à l'usager correspondant (par défaut **/bin/bash** sur RedHat).
- Le Shell associé est ainsi lancé automatiquement dès l'ouverture de la session.
- Le Shell exécute les scripts globaux à tous les utilisateurs et les scripts liés au compte et qui permettent une personnalisation.
- Enfin, il affiche le prompt et se met en attente de la lecture d'une commande.
- La commande ***exit***, permet de quitter le Shell.

1.2 LES SCRIPTS DE CONNEXION

1) Le script global ***/etc/profile*** est exécuté par tous les usagers y compris **root**
2) Celui-ci cherche à exécuter tous les scripts **/etc/profile.d/*.sh**
3) Puis il y a exécution du script personnel **$HOME/.bash_profile** (la variable $HOME contient le chemin vers le répertoire personnel). Il s'agit ainsi d'un fichier de démarrage personnel et paramétrable.

    - Si le fichier **.bash_profile** n'existe pas alors il cherche à exécuter le fichier **.bash_login**
    - Si le fichier **.bash_login** n'existe pas à son tour alors il cherche à exécuter le fichier **.profile**

4) A son tour il exécute **$HOME/.bashrc** dans lequel il est recommandé de placer toutes les fonctions ou alias personnels (car **.bashrc** est exécuté dans tout shell).
5) Enfin le précédent script exécute **/etc/bashrc**, dans lequel on place les alias globaux et la définition symbolique du prompt **$PS1**
6) Puis le prompt utilisateur s'affiche et le Shell attend une commande.
7) Le script $HOME/bash_logout est exécuté lorsque la session est terminée (exit ou CTL-D)

1.3 PERSONNALISATION DES COMMANDES BASH

/etc/bashrc étant le dernier script d'initialisation du Shell **bash**, **root** peut y définir des alias globaux pour tous les usagers.

```
# vi /etc/bashrc
alias effacer_ecran=clear
```

1.4 PERSONNALISATION DU LOGIN UTILISATEUR

Chaque usager peut ajouter des commandes Shell au fichier de profil personnel, **~/.bash_profile.**

Voici un exemple:

```
clear
salut="Bonjour $USER ! Nous sommes le $(date)"
echo  $salut
```

2 FACILITÉS DE SAISIE DES COMMANDES

Comme les commandes Linux sont souvent longues à saisir, diverses facilités sont offertes:

2.1 HISTORIQUE DES COMMANDES

Cette liste numérotée est accessible en tapant:

```
# history | less
```

Pour relancer la commande numéro **n**, saisir (sans espace):

```
# !n
```

On peut aussi parcourir les précédentes lignes de commandes avec les flèches (comme **doskey**) et les éditer. Ceci permet très facilement de reprendre une précédente commande pour l'éditer et la modifier.

2.2 LE CLIC-DROIT

Dans un terminal console, sélectionner un texte quelconque. Un clic droit recopie ce texte sur la ligne de commande, même dans une autre console.

2.3 L'OPÉRATEUR TILDE(~)

Le caractère tilde ~ (Alt 126) **seul** renvoie au **répertoire personnel** de l'usager actuel.

Le tilde **~** suivi d'un nom d'usager, par exemple **oracle**, renvoie au **répertoire personnel** de l'usager **oracle**.

```
# cd ~oracle
```

2.4 COMPLÉTER UNE COMMANDE

Lorsqu'on tape une commande en ligne ensuite la touche **TAB**, l'interpréteur cherche à compléter le nom du fichier.

```
# less /etc/pass
```

S'il y a plusieurs propositions, il y a attente d'un complément d'informations de la part de l'usager (avec un **beep** sonore).

Un autre **TAB** et l'interpréteur affiche toutes les possibilités où en indique le nombre, s'il y en a beaucoup.

2.5 ENSEMBLE DE FICHIERS

- Travailler avec le Shell nécessite souvent de manipuler des ensembles de fichiers. L'utilisation de caractères spéciaux (appelés aussi **métacaractères**) dans les noms de fichiers, permet de générer des modèles pour désigner ces ensembles.
- Il existe quatre constructeurs de modèles ***, ? , [] et ^**.

Modèle	Signification
*	Remplace une chaîne de longueur quelconque, même vide
?	Remplace un seul caractère quelconque
[ ]	Un caractère quelconque de la liste ou de l'intervalle
[^]	N'importe quel caractère **sauf** ceux de la liste

- Le modèle **[ ]** permet de sélectionner un élément de la liste ou de l'intervalle spécifié.
- Dans le cas d'une suite ordonnée de caractères comme abc...z, on peut utiliser la notation intervalle **a-z**
- On peut mélanger les deux notations, comme dans [a-z].[0-9], ensemble des fichiers a.0, a.1, .., b.0 b.1 etc.

EXEMPLES

- **ll a***
- **ll [a-dA-D]*** liste les fichiers du répertoire courant dont le nom commence par a, b, c ou d minuscule ou majuscule (y compris les répertoires)
- **cp prog1[0-9] /home/java** copie tous les fichiers **prog10** jusqu'à **prog19**

2.6 CARACTÈRES SPÉCIAUX

Caractère	Description
$	Début d'une variable Shell
|	Rediriger la sortie standard sur la prochaine commande
#	Commentaire
&	Exécuter un processus en arrière-plan
?	Remplace un seul caractère quelconque
*	Remplace une chaîne de longueur quelconque, même vide
>	Opérateur de redirection de la sortie
<	Opérateur de redirection de l'entrée
`	(backquote) substitution de commande
>>	Opérateur de redirection de la sortie (append)
[ ]	Liste ou intervalle
[a-z]	Intervalle
[a,z]	Liste
. nom_fichier	Exécuter le fichier *nom_fichier* dans le même Shell.
space	Séparateur de mots
"	Guillemets (double quote)
‘	Apostrophe (single quote)
\	Backslash
;	Séparateur de séquence

2.7 GUILLEMETS (DOUBLE QUOTE)

- Habituellement, une ligne de commande saisie au prompt de la console ou bien écrite dans un script est une phrase composée de mots séparés par des espaces (ou des tabulations). Le premier mot est considéré comme le nom d'une commande et le Shell cherche à l'exécuter; les mots suivants sont des options ou paramètres de cette commande.
- Pour inhiber cette interprétation des espaces, il faut entourer le groupe de mots de "double quotes", ce groupe sera alors interprété comme un seul paramètre.

EXEMPLE

Recherche de la chaîne Jean Jacques (qui constitue un seul paramètre) sur les lignes de **/etc/passwd** (l'option -i pour s'affranchir de la casse)

```
grep -i "Jean Jacques" /etc/passwd
```

```
var="test chaîne"
newvar="valeur de var est $var"
echo $newvar
```

2.8 APOSTROPHE (SINGLE QUOTE)

L’utilisation du "double quotes" permet la résolution des variables contrairement au "single quote".

```
var=’test chaîne’
newvar=’valeur de var est $var’
echo $newvar
```

2.9 BACKSLASH \

```
var=$test
var=\$test
```

2.10 BACKQUOTE

```
var=`date`
```

3 LIGNE DE COMMANDES

3.1 ANALYSE DE LA LIGNE DE COMMANDE

Le Shell commence par découper la ligne en mots séparés par des blancs.

Le premier mot attendu est le nom d'une commande. Les mots suivants sont considérés comme des options ou des paramètres dont la "compréhension" incombe à la commande**.**

EXEMPLE

Supposons que les comptes **java1** à **java20** sont déjà créés.

```
# grep -n java /etc/passwd
```

La commande **grep** attend :

- des options précédées de –
- un modèle (expression rationnelle) des chaînes à chercher
- un ensemble de fichiers où elle doit chercher.

3.2 ENCHAÎNEMENT INCONDITIONNEL DES COMMANDES

- En général, on place une commande par ligne que ce soit en ligne de commande ou dans un script.
- Le point-virgule **;** joue le rôle de séparateur de séquence **inconditionnel**.

Il permet ainsi d'écrire une séquence de plusieurs commandes sur une même ligne.

Toutes les commandes sont inconditionnellement exécutées (même si l'une d'entre elle provoque une erreur), et leurs résultats respectifs sont envoyés sur la sortie standard, séparés par un retour à la ligne "\n".

EXEMPLE

```
# xyz ; ls -l
```

```
# clear ; ls –l
```

EXEMPLE (Exécuter dans un sous-Shell)

```
# (a=2;b=3)
# echo $a
# echo $b
```

EXEMPLE (Exécuter dans le même Shell)

```
# { c=4;d=5; }
# echo $c
# echo $d
```

EXEMPLE

```
# mkdir  /usr/hakimb/{java,delphi,oracle,db2}
```

La commande précédente est équivalente à :

```
# mkdir  /usr/hakimb/java
# mkdir  /usr/hakimb/delphi
# mkdir  /usr/hakimb/oracle
# mkdir  /usr/hakimb/db2
```

EXEMPLE

```
chown  v1000
/root/usr/{prog/{java,delphi},lib/{fi?h?*,how_ex}}
```

La commande précédente est équivalente à :

```
chown   v1000   /root/usr/prog/java
chown   v1000   /root/usr/prog/delphi
chown   v1000   /root/usr/lib/fi?h?*
chown   v1000   /root/usr/lib/how_ex
```

3.3 ENCHAÎNEMENT CONDITIONNEL DES COMMANDES

- Les séparateurs **&& (AND)** et **|| (OR)** sur la ligne de commande sont des séparateurs qui jouent les rôles d'opérateurs **conditionnels**, en ce sens que la 2ème commande sera exécutée en fonction du code de retour de la 1ère commande.
- Dans *commande1 && commande2*, commande2 ne sera exécutée que si le code de retour de commande1 est 0 (exécution correcte)
- Dans *commande1* || *commande2*, commande2 ne sera exécutée que si le code de retour de commande1 est différent de 0 (exécution erronée)

EXEMPLE

Trouver la signification de la commande suivante:

```
$ cd ~/tmp || mkdir $HOME/tmp
```

3.4 REDIRECTIONS DES ENTRÉES-SORTIES

Toutes les commandes sont dotées par le système de **3 canaux de communication**:

```
# ls -l /dev/std*
lrwxrwxrwx  1 root  root  17 Aug 12 09:37 /dev/stderr -> ./proc/self/fd/2
lrwxrwxrwx  1 root  root  17 Aug 12 09:37 /dev/stdin  -> ./proc/self/fd/0
lrwxrwxrwx  1 root  root  17 Aug 12 09:37 /dev/stdout -> ./proc/self/fd/1
```

Voici la signification des périphériques :

Device	file descriptor	Description
/dev/stdin	0	entrée standard (*stdin*=standard input)
/dev/stdout	1	sortie standard (*stdout=standard output*)
/dev/stderr	2	sortie des erreurs (*stdout=standard output*)

EXEMPLE

```
# ls > list
# more list
fichier1
fichier2
```

EXEMPLE

```
# mkdir /root > log
mkdir: cannot create directory '/root': File exists

# more log
```

EXEMPLE

```
# mkdir /root > log 2>&1
# more log
mkdir: cannot create directory '/root': File exists
```

- Par défaut les canaux d'entrées et de sorties communiquent avec le clavier et l'écran. Les commandes et les programmes qui ont besoin de données les attendent en provenance du clavier et expédient leurs résultats pour affichage sur le moniteur.
- Il est possible de les détourner pour les rediriger vers des fichiers ou même vers les entrées-sorties d'autres commandes. Les symboles utilisés sont:

SYMBOLE	DESCRIPTION
<	Redirection de l'entrée standard à partir d'un fichier
>	Redirection de la sortie standard en direction d'un fichier. Le fichier est créé et écrase sans préavis le fichier existant portant le même nom
>>	Redirection de la sortie standard à la fin du fichier s'il existe déjà
|	Enchaînement de commandes (appelé aussi tube ou pipe). La sortie de la commande gauche est envoyée en entrée de la commande droite Fréquemment utilisé avec less (ou more) pour examiner l'affichage La valeur de retour est celle de la dernière commande

EXEMPLE

```
ll --help | less
```

3.5 SUBSTITUTION DE COMMANDE

- Ce procédé permet de substituer au texte d'une commande le résultat de son exécution qui est envoyé sur la sortie standard
- La commande doit être entourée de l'opérateur "backquote" ` ou être placée dans une parenthèse précédée de **$(...)**. D'une manière générale, il est recommandé d'entourer l'expression de " "

EXEMPLES

```
echo "`whoami`, nous sommes le `date` "
```

```
echo "$(whoami), nous sommes le $(date) "
```

4 PROGRAMMATION BASH

4.1 Introduction

- Un script Bash est un fichier de type texte contenant une suite de commandes Shell, exécutable par l'interpréteur (par exemple /bin/bash), comme une commande unique. Un script peut être lancé en ligne de commande, comme dans un autre script.
- Il s'agit bien plus qu'un simple enchaînement de commande : on peut définir des variables et utiliser des structures de contrôle, ce qui lui confère le statut de langage de programmation interprété et complet.
- Le langage **Bash** gère notamment :

  - les entrées-sorties et de leur redirection
  - les variables (système, environnement et usager)
  - le passage de paramètres
  - les structures conditionnelles et itératives
  - les fonctions internes

4.2 CRÉATION D'UN SCRIPT

- Les lignes commençant par le caractère dièse # sont des commentaires.
- Le script débute généralement par l'indication de son interpréteur écrite sur la première ligne : **#!/bin/bash**

EXEMPLE

```
#!/bin/bash
# script bonjour
# affiche un salut à l'utilisateur qui l'a lancé
# la variable d'environnement $USER contient le nom de login
echo ---- Bonjour $USER -----
# l'option -n empêche le passage à la ligne
# le ; sert de séparateur des commandes sur la ligne
echo -n "Nous sommes le " ; date
# recherche de $USER en début de ligne dans le fichier passwd
# puis extraction de l'uid au 3ème champ, et affichage
echo "Ton UID est " $(grep "^$USER" /etc/passwd | cut -d: -f3)
```

4.3 EXÉCUTION DU SCRIPT

Il est indispensable que le fichier script ait la permission **x** (soit exécutable):

```
chmod a+x bonjour
```

Pour lancer l'exécution du script, taper:

```
./bonjour
```

./ indiquant le chemin, ici le répertoire courant. Ou bien indiquer le chemin absolu à partir de la racine. Ceci dans le cas où le répertoire contenant le script n'est pas inclus dans le PATH

Si les scripts personnels sont systématiquement stockés dans un répertoire précis, par exemple */home//bin*, on peut ajouter ce chemin dans le PATH. Pour cela, il suffit d'ajouter la ligne suivante dans */etc/skel/.*

```
# bash_profile
PATH=$PATH:$HOME/bin
```

Si on entre une instruction incomplète en ligne de commande, l'interpréteur passe à la ligne suivante en affichant le prompt **>** et attend la suite de l'instruction (pour quitter Ctrl-C).

4.4 MISE AU POINT (DÉBOGAGE)

Exécution en mode "trace" (-x) et en mode "verbose" (-v)

```
sh -x ./bonjour
```

Pour aider à la mise au point d'un script, on peut insérer des lignes temporaires:

echo $var pour afficher la valeur de la variable

exit 1 pour forcer l'arrêt du script à cet endroit

On peut passer des arguments à la suite du nom du script, séparés par des espaces. Les valeurs de ces paramètres sont récupérables dans le script grâce aux paramètres de position $1, $2 .. mais, contrairement aux langages de programmation classiques, ils ne peuvent pas être modifiés.

EXEMPLE

```
#!/bin/bash
# appel du script : ./bonjour nom prénom
if [ $#  -eq 2 ]
 then
echo "Bonjour $2 $1 et bonne journée !"
 else
echo "Syntaxe : $0 nom prénom"
fi
```

4.5 ENTRÉES-SORTIES

Ce sont les voies de communication entre le programme Bash et la console:

La commande **echo**, affiche son argument texte entre guillemets sur la sortie standard. La validation d'une commande **echo** provoque un saut de ligne.

```
echo "Bonjour à tous !"
```

On peut insérer les caractères spéciaux habituels, qui seront interprétés seulement si l'option **-e** suit echo

- \n (saut ligne)
- \b retour arrière)
- \t (tabulation)
- \a (alarme)
- \c (fin sans saut de ligne)

```
echo "Bonjour \nà tous !"
echo -e "Bonjour \nà tous !"
echo -e "Bonjour \nà toutes \net à tous !"
```

La commande **read**, permet l'affectation directe par lecture de la valeur, saisie sur l'entrée standard au clavier

read *var1 var2 ...* attend la saisie au clavier d'une liste de valeurs pour les affecter, après la validation globale, respectivement aux variables var1, var2 ..

```
echo "Donnez votre prénom et votre nom"
read prenom nom
echo "Bonjour $prenom $nom"
```

5 LES VARIABLES BASH

5.1 VARIABLES USAGERS

Syntaxe : variable=valeur

Le signe = NE DOIT PAS être entouré d'espace(s)

On peut initialiser une variable à une chaîne vide :

```
chaine_vide=
```

Si **valeur** est une chaîne avec des espaces ou des caractères spéciaux, l'entourer de " " ou de ' '

Le caractère \ permet de masquer le sens d'un caractère spécial comme **"** ou **'** faire précéder le nom de la variable du signe **$** pour faire référence à sa valeur.

Pour afficher toutes les variables :

```
set
```

Pour empêcher la modification d'une variable, invoquer la commande **readonly**

```
readonly  pi=3.14
```

ASSIGNER UNE VALEUR A UNE VARIABLE

```
#  total=23
#  nom=hakim
#  nom='Hakim Benameurlaine'
```

LIRE LA VALEUR D'UNE VARIABLE

```
nom='orabec'
nouveau_nom=$nom
```

SUBSTITUTION DE VARIABLE

Si une chaîne contient la référence à une variable, le Shell doit d'abord remplacer cette référence par sa valeur avant d'interpréter la phrase globalement.

EXEMPLES

```
n=abc
echo "la variable \$n vaut $n"
salut="Bienvenue à tous !"
echo "Message1 : $salut"
echo 'Message2 : $salut'
echo "Message3 : \"$salut\" "
readonly salut
salut="simple bonjour"
echo "Message4 : $salut"
```

VARIABLES EXPORTÉES

Toute variable est définie dans un Shell. Pour qu'elle devienne globale elle doit être exportée par la commande :

```
export variable
export variable=valeur
export --> Pour obtenir la liste des variables
exportées
```

EXEMPLE

```
export ORACLE_SID=sapprod
```

OPÉRATEUR {}

Dans certains cas en programmation, on peut être amené à utiliser des noms de variables dans d'autres variables. Comme il n'y a pas de substitution automatique, la présence de {} force l'interprétation des variables incluses. Voici un exemple :

```
user="/home/java"
echo $user
u1=$user1
echo "cas1: u1="$u1
u1=${user}1
echo "cas2: u1="$u1
```

5.2 VARIABLES D'ENVIRONNEMENT

La liste en est accessible par la commande:

```
#  env
```

Les plus utiles sont $HOME, $PATH, $USER, $PS1, $SHELL, $ENV, $PWD ..

La commande echo permet d'obtenir la valeur d'une telle variable. Par exemple:

```
# echo $PATH
# echo $USER
```

EXEMPLE

Ajout d'un nouveau chemin : attention à ne pas écraser la liste des chemins existants (PATH en majuscules !)

Pour ajouter le chemin vers les exécutables de java

```
PATH=$PATH:/home/java/bin
```

Pour ajouter le répertoire courant (non présent par défaut)

```
PATH=$PATH :./
```

La variable **$HOME** contient le chemin du répertoire personnel. La commande cd $HOME est abrégée en cd

La variable **$USER** contient le nom de l'usager

$SHLVL donne le niveau du *Shell* courant

```
$ nom=Albert
$ message="Je m'appelle $nom"
$ echo Aujourd\'hui, quel jour sommes-nous? ; read jour
$ echo aujourd\'hui $jour, $message usager:$USER,
station:$HOSTNAME
```

5.3 VARIABLES SYSTÈME

Elles sont gérées par le système et s'avèrent très utiles dans les scripts. Bien entendu, elles ne sont accessibles qu'en lecture.

Ces variables sont automatiquement affectées lors d'un appel de script suivi d'une liste de paramètres. Leurs valeurs sont récupérables dans $1, $2 ...$9

Variable	Signification
$?	C'est la valeur de sortie de la dernière commande. Elle vaut 0 si la commande s'est déroulée sans problème
$0	Cette variable contient le nom du script
$1 à $9	Les paramètres passés à l'appel du script
$#	Le nombre de paramètres passés au script
$*	La liste des paramètres à partir de $1
$$	Le PID du processus courant
$!	Le PID du processus fils

EXEMPLES

```
if [ $#  -eq  0 ]
then
  echo  "pas de nom"
else
  echo "Votre nom est "$1
fi
```

```
echo "nombre de paramètres :"$#
echo "nom du programme :"$0
echo "liste des paramètres :"$*
```

```
ls -l
echo $?         ----> 0
cd abcdef
echo $?          ---> 1
```

5.4 Tableaux

Déclarer un tableau :

```
declare -a tab1
```

Lister les tableaux :

```
declare -a
```

Initialiser un tableau :

Il n’est pas nécessaire d'initialiser tous les éléments d’un tableau

```
tab2[2]=23
read tab3[2] tab3[4]
tab4=([0]=800 [4]=900)
```

Accéder à un élément (premier élément=index 0)

```
echo ${tab4[0]}
```

Déterminer le nombre d'éléments d'un tableau :

```
echo ${#tab4[*]}
```

Afficher tous les éléments :

```
echo ${tab4[*]}
```

5.5 PARAMÈTRES DE POSITION

On peut récupérer facilement les compléments de commande passés sous forme d'arguments sur la ligne de commande, à la suite du nom du script, et les utiliser pour effectuer des traitements.

Ce sont les variables système spéciales **$1 , $2 .. $9** appelées paramètres de position.

Celles-ci prennent au moment de l'appel du script, les valeurs des chaînes passées à la suite du nom du script.

- le nombre d'argument est connu avec **$#**
- la liste complète des valeurs des paramètres s'obtient avec $*

6 COMMANDES IMPORTANTES

6.1 Commande shift

La commande **shift** effectue un décalage de pas +1 dans les variables $. C'est à dire que $1 prend la valeur de $2 et ainsi de suite.

EXEMPLE

```
a=1 ; b=2 ; c=3
set $a $b $c
echo $1, $2, $3
shift
echo $1, $2, $3
```

6.2 tr

Cette commande de filtre permet d'effectuer des remplacements de caractères dans une chaîne.

Transformer une chaîne en minuscules

```
chaine="BONJOUR"
echo $chaine | tr 'A-Z' 'a-z'
```

Pour permettre l'utilisation de la commande **set**, il est nécessaire que le séparateur de champ sur une ligne soit l'espace comme dans l'exemple suivant.

Créer un fichier passwd.txt qui introduit un espace à la place de ":" á partir de **/etc/passwd**

```
cat /etc/passwd | tr ":" " " > passwd.txt
```

6.3 set

Cette commande interne est très pratique pour séparer une ligne en une liste de mots, chacun de ces mots étant affecté à une variable positionnelle. Le caractère de séparation est l'espace.

```
# soit une chaîne couleurs qui contient une liste de
mots
couleurs="rouge vert bleu noir"
#set va lire chaque mot de la liste et l'affecter aux
paramètres
set $couleurs
echo $1 $2 $3
shift
echo $1 $2 $3
```

```
a=1 ; b=2 ; c=3
set $a $b $c
echo $1, $2, $3
# les valeurs de a, b, c sont récupérées dans $1, $2,
$3
```

6.4 eval

Cette commande ordonne l'interprétation par le Shell de la chaîne passée en argument. On peut ainsi construire une chaîne que l'appel à **eval** permettra d'exécuter comme une commande !

EXEMPLE

```
message="Quelle est la date d'aujourd'hui"
set  $message
echo $#  ---> le nombre de mots est 5
echo $1
echo $4  ---> affiche la chaîne "date"
echo $5
eval $4  ---> interprète la chaîne "date" comme une
commande
```

Il est souvent pratique de construire une chaîne dont la valeur sera égale au libellé d'un enchaînement de commandes. Pour faire exécuter ces commandes contenues dans la chaîne, on la passe comme argument de la commande **eval**

```
liste="date;who;pwd"
#( ' ' ou " " obligatoires sinon le ; est un
séparateur de commandes)
eval $liste  ---> exécute bien les 3 commandes
```

Soit la chaîne $user qui contient des informations sur un compte à créer. S'il utilise un autre séparateur que ";" on fait appel à **tr** en premier:

```
user="login=usager8 ; motdepasse=h34345 ; nom=Jean ;
groupe=java"
eval $user
echo $login $motdepasse $nom $groupe
useradd -G $groupe $login
echo $motdepasse | (passwd --stdin $login)
```

6.5 basename

Permet d'éliminer le chemin d'accès et le suffixe d'un nom de fichier.

Syntaxe : basename nom [suffixe]

EXEMPLES

```
#  basename /etc/lpd.conf  .conf        -->  lpd
```

Si un suffixe est indiqué, et s'il est identique à la partie finale du nom, il est éliminé de celui-ci

```
# basename /etc/named.conf  .conf       -->  named
```

6.6 seq

Permet de générer une séquence de nombres.

Syntaxe:

- seq [options] Last
- seq [options] First Last
- seq [options] First Increment Last

EXEMPLES

```
seq 30
```

```
seq -s : 30
```

```
seq 1 2 40
```

6.7 test

Comme son nom l'indique, elle sert à vérifier des conditions. Ces conditions portent sur des fichiers ou des chaînes ou une expression numérique.

Cette commande courante sert donc à prendre des décisions, d'où son utilisation comme condition dans les structures conditionnelles **if.. then ..else**, en quelque sorte à la place de variables booléennes... qui n'existent pas.

La syntaxe est la suivante:

```
test expression
[ expression ]  attention aux espaces autour de expression
```

6.8 VALEUR DE RETOUR

- Chaque commande transmet au programme appelant un code, appelée valeur de retour (*exit status*) qui stipule la manière dont son exécution s'est déroulée. Cette valeur numérique est stockée dans la variable spéciale **$?**
- Par convention du *Shell BASH*, **la valeur de retour est toujours 0 si la commande s'est déroulée correctement**, sans erreur.
- Une valeur de retour différente de 0 signale donc une erreur, qui peut être éventuellement analysée selon cette valeur.
- Une variable système spéciale $? contient la valeur de retour de la précédente commande. On peut afficher cette valeur avec la commande ***echo***

EXEMPLES

```
$ ll ~
$ echo $?              --> 0
$ ll /abcd
$ echo $?              --> 1, si abcd  n'existe pas
```

6.9 TESTER UN FICHIER

Deux syntaxes possibles (la seconde est la plus utilisée):

```
test option fichier
[ option fichier ]
```

Tableau des principales options :

Option	Signification
-e	Retourne vraie si le fichier existe
-f	Retourne vraie si c'est un fichier normal
-d	Retourne vraie si c'est un répertoire
-r	Retourne vraie si le fichier est accessible en lecture
-w	Retourne vraie si le fichier est accessible en écriture (Modifiable)
-x	Retourne vraie si le fichier est exécutable
-s	Retourne vraie si le fichier n'est pas vide

EXEMPLES

```
[ -s $1 ]
vrai (renvoie 0)     si le fichier passé en argument
n'est pas vide
```

```
[ $#  -eq  0 ]le nombre d'arguments est 0
```

```
[ -w  fichier ]      le fichier est-il modifiable ?
```

```
 [ -r  "/etc/passwd" ]     peut-on lire /etc/passwd ?
 echo $? --> 0 (vrai)
```

```
 [ -r "/etc/shadow" ]      peut-on lire  /etc/shadow ?
 echo $? --> 1 (faux)
```

```
 [ -r "/etc/shadow" ] || echo "échec de lecture"
```

6.10 TESTER UNE CHAÎNE

```
[ option chaine ]
```

Tableau des principales options :

Option	Signification
-z	Retourne vraie si la chaîne est vide
-n	Retourne vraie si la chaîne n'est pas vide
=	Retourne vraie si les chaînes comparées sont identiques
!=	Retourne vraie si les chaînes comparées sont différentes

EXEMPLES

```
[ -n "non vide" ] ; echo $?
```

```
chaine="bonjour" ; [ $chaine   =    "bonjour" ] ; echo $?
```

```
[ $USER != "root" ] && echo "l'utilisateur n'est pas root"
```

6.11 TESTER UN NOMBRE

```
[ nb1 option nb2 ]
```

Il y a une conversion automatique de la chaîne de caractères en nombre

Option	Signification
-eq	Égal
-ne	Différent
-lt	Strictement inférieur
-gt	Strictement supérieur
-le	Inférieur ou égal
-ge	Supérieur ou égal

EXEMPLE

```
a=2 ; [ $a -lt 9 ] ; echo $?
```

6.12 OPÉRATIONS BOOLÉENNES

Option	Valeur
[expression1 -a expression2]	(**and**) 0 si les 2 expressions sont vraies
[expression1 -o expression2]	(**or**) 0 si l'une des 2 expressions est vraie
[! expression1]	Négation

EXEMPLES

```
f="/root" ; [ -d $f -a -x $f ] ; echo $?
```

```
val=80; [ $val -lt 20 -o $val -ge 41 ] || echo "val
entre 20 et 40"
```

7 STRUCTURES DE CONTRÔLE

7.1 STRUCTURES CONDITIONNELLES

```
if suite-de-commandes
then
# séquence exécutée si suite-de-commandes rend une
valeur 0
 bloc-instruction1
else
# séquence exécutée sinon
 bloc-instruction2
fi
```

Attention ! si then est placé sur la 1ère ligne, séparer avec un ;

```
if commande; then …; fi
```

EXEMPLES

On teste la présence d'une ligne commençant par polo dans /etc/passwd

```
grep  "^polo"  /etc/passwd > /dev/null
if [ $? -eq 0 ] ; then
  echo "polo a déjà un compte"
fi
```

```
Si root a eu une bonne note, on le félicite
intra=84
if [ $intra -gt 80 ] ---> test vrai, valeur retournée : 0
then
  echo "Très bien !"
fi
Avant d'exécuter un script, tester son existence.
Extrait de $HOME/.bash_profile
if  [ -f  ~/.bashrc ]
then
 . ~/.bashrc
fi
```

7.2 CONDITIONNELLES IMBRIQUÉES

Pour imbriquer plusieurs conditions, on utilise la construction :

```
if commande1
then
 bloc-instruction1
elif commande2
then
 bloc-instruction2
else
# si toutes les conditions précédentes sont fausses
 bloc-instruction3
fi
```

EXEMPLES

Trouver le maximum de trois nombres a b c:

```
#! /bin/bash
a=$1;b=$2;c=$3
if [ $a -gt $b ]
then
   if [ $a -gt $c ]
   then
     echo "1maximum=$a"
   else
     echo "2maximum=$c"
   fi
elif [ $b -gt $c ]
then
 echo "3maximum=$b"
else
 echo "4maximum=$c"
fi
```

Supposons que le script exige la présence d'au moins un paramètre:

```
#! /bin/bash
if [ $#  -eq  0 ]
then
  echo "paramètre absent"
elif [ $#  -eq  1 ]
  then
    echo "donner le second paramètre:"
    read parametre2
fi
```

7.3 CHOIX MULTIPLES (case)

```
case valeur in
 expr1) commandes ;;
 expr2) commandes ;;
 ...
esac
```

EXEMPLES

Supposons que le script doit réagir différemment selon la valeur de **$USER**

```
#! /bin/bash
case $USER in
root)               echo "mes respects M le $USER";;
usager1|usager2)    echo "Bonjour $USER";;
usager3)       echo "Réveille-toi $USER";;
esac
```

```
Le script attend une réponse oui/non Yes/No de
l'utilisateur
read reponse
case $reponse in
[yYoO]*)       echo "réponse positive";;
[nN]*)         echo "réponse négative";;
*)             echo "réponse incorrecte";;
esac
```

```
read langue
case $langue in
francais)      echo Bonjour ;;
anglais)            echo Hello ;;
espagnol)      echo Buenos Dias ;;
esac
```

```
param=$1
case $param in
0|1|2|3|4|5|6|7|8|9 )  echo $param est un chiffre ;;
[0-9]*)         echo $param commence par un chiffre ;;
[a-zA-Z]*)      echo $param commence par une lettre;;
*)       echo $param commence par un caractère inconnu ;;
esac
```

Un vrai exemple, extrait du script **smb** (/etc/rc.d/init.d/smb)

```
# smb attend un paramètre, récupéré dans la variable
$1
case "$1" in
  start)
echo -n "Starting SMB services: "
deamon smbd -D
echo
echo -n "Starting NMB services: "
deamon nmbd -D
;;
  stop)
echo -n "Shutting SMB services: "
killproc smbd
  ;;
esac
```

8 STRUCTURES ITÉRATIVES

8.1 BOUCLE FOR

```
for variable in [liste]
do
 commandes (utilisant $variable)
done
```

La liste peut être explicite:

```
for nom in root usager1 usager2
do
 echo "changer mot de passe de $nom"
 passwd $nom
done
```

La liste peut être calculée à partir d'une expression:

```
# Recopier les fichiers personnels de root dans /tmp
for ficher in  /root/*
do
 cp $fichier /tmp
done
```

La liste peut être à partir de la variable système **$***:

```
# pour construire une liste de fichiers dans $*
cd  /root
set *
echo $*
for nom in $*
do
 echo $nom
done
```

EXEMPLE

```
for nom in u1 u2 u3
do
useradd  $nom
done
```

```
for i in /home/u[1-3] ; do ls -la  $i; done
```

```
for i in /home/*/* ; do echo  $i; done
```

```
for i in /dev/tty[1-7]
do
setleds -D +num < $i
echo  $i
done
```

```
for x in /home/*
do
echo $x
done
```

8.2 BOUCLE WHILE

while liste-commandes **do** commandes **done**	La répétition se poursuit TANT QUE la dernière commande de la liste est vraie
until liste-commandes **do** commandes **done**	La répétition se poursuit JUSQU'A CE QUE la dernière commande de la liste devienne vraie

EXEMPLES

```
echo -e "Entrez un nom de fichier"
read fichier
while [ -z $fichier ]          #chaîne vide
do
     echo -e "Invalide"
     read fichier
done
```

```
while
echo -e" Entrez un nom de fichier"
read fichier
[ -z $fichier ]
do
     echo -e "Invalide"
done
```

```
# Pour dire bonjour toutes les secondes (arrêt par
CTRL-C)
while true
do
     echo "Bonjour $USER"
     sleep 1
done
```

Noter que la redirection de l'entrée de la commande ***while .. do .. done*** est placée à la fin.

```
#Lecture des lignes d'un fichier pour traitement
fichier=/etc/passwd
while read ligne
do
     echo $ligne
done < $fichier
```

8.3 SORTIE ET REPRISE DE BOUCLE

break placé dans le corps d'une boucle, provoque une sortie définitive de cette boucle.

continue permet de sauter les instructions du corps de la boucle (qui suivent continue) et de "continuer" à l'itération suivante. Pour les boucles ***for***, ***while*** et ***until***, continue provoque donc la réévaluation immédiate du test de la boucle.

EXEMPLES

Boucle de lecture au clavier arrêtée par la saisie de *stop*

```
#! /bin/bash
text=""
while true
do
    read ligne
    if [ $ligne = stop ]
    then
       break
    else
       text="$text \n$ligne"
    fi
done
echo -e text=$text
```

Lecture des lignes d'un fichier:

```
fichier="/etc/passwd"
grep "^usager" $fichier | while true
do
  read ligne
  if [ "$ligne" = "" ]
  then
    break
  fi
  echo $ligne
done
```

9 FONCTIONS

Syntaxe:

```
function nom-fct {
 bloc d'instructions
}
```

```
nom-fct() {
 bloc d'instructions
}
```

EXEMPLES

```
function test_fonction
{
echo Fonction test
}
```

En tant *root*, on doit relancer les "démons", si on a modifié un fichier de configuration.

Par exemple /etc/rc.d/init.d/smb contient la commande deamon smbd -D , pourtant à l'essai deamon est une commande inconnue !

Reportons-nous au début du fichier, le script /etc/rc.d/init.d/functions y est appelé. Celui-ci contient la fonction : daemon()

Dans le corps de la fonction, on peut définir et utiliser des variables déclarées locales, en les introduisant avec le mot-clé **local**

10 CALCUL SUR LES ENTIERS

Ne pas confondre la syntaxe **$((expression arithmétique))** avec la substitution de commande **$(commande)**

```
echo $((30+2))           32
echo $((30+2*10/4))           35
echo $(( (30+2) * (10-7) /4 ))          24
```

Le langage Bash est inadapté aux calculs numériques.

Voici un exemple qui calcul de la factorielle (de 1 à 10)

```
declare -i k
 k=1
 p=1
while [ $k -le 10 ]
do
  echo "$k!=" $((p=$p*$k))
  k= $k+1
done
```