

# **Le Shell (A)**

- A      PRESENTATION**
- B      HISTORIQUE**
- C      EXECUTION**
- D      LES VARIABLES**
- E      SUBSTITUTION DE COMMANDE**
- F      LA COMMANDE EXPR**
- G      LA COMMANDE TEST**
- H      SHELL SCRIPTS**
- I      STRUCTURES DE CONTROLE**
- J      INSTRUCTIONS DIVERSES**
- K      EXEMPLES**
- L      EXERCICES : CORRECTIONS**

## A) Présentation

Ce document est destiné à l'étude des capacités de l'interpréteur de commandes : le **Shell**.

Un Shell peut être utilisé suivant deux modes :

- un mode **interactif** : c'est ce que nous utilisons depuis le début; l'utilisateur frappe au clavier des commandes, que le **Shell interprète**.

En dehors du prompt (le symbole # pour le compte root, \$ pour les autres utilisateurs et % pour le C-Shell), qui donne une indication sur le Shell utilisé, on peut consulter le contenu de la variable d'environnement, qui contient le **nom du Shell courant**, en tapant :

`echo $SHELL`

- un mode **programmé** : dans ce cas, une suite de commandes est enregistrée dans un fichier, et l'utilisateur demande au **Shell d'exécuter ce fichier**. Les commandes forment un véritable langage de programmation. Un utilisateur pourra ainsi combiner des séquences de commandes pour construire de nouveaux programmes, communément appelés des **procédures, ou scripts Shells**.

## B) Historique

Le premier interpréteur de commandes, inventé par un français, Mr PERRET, dans les années 60, et testé avec les premières versions d'Unix, n'a pas laissé de souvenir impérissable.

En 1976, Steve BOURNE crée une version du Shell qui servira de référence : Dès que d'autres versions apparaîtront, on l'appellera (fort logiquement) "*Bourne Shell*".

Les développements de la branche Unix-BSD ont amené une autre version : le **C-Shell** (Bonjour, les cocotiers !), dont la syntaxe reprend celle du langage C.

Ce Shell sera donc très prisé des **développeurs sous UNIX**.

Partant du **Bourne-Shell**, une autre version a été créée par David KORN, qui présentait des atouts formidables par rapport à ses "concurrents".

Citons, par exemple :

- ✗ Possibilité de répétition des commandes, via un historique;
- ✗ Mécanisme des alias;
- ✗ Inclusion des fonctionnalités du Bourne-Shell.

Le **Korn-Shell** a été considéré comme standard, à partir de sa diffusion, sous **UNIX System V R. 4**.

Au début des années 90, S. Bourne a récidivé (le coquin), et développé le **Bash (Bourne Again Shell)**, qui propose évidemment les avantages cités pour le **K-Shell**.

Remarque importante : Petit à petit, la compatibilité ascendante, avec le Shell d'origine, a été considérablement réduite.

## C) Exécution

### 1. Programmes

Par définition, toute saisie **validée en ligne de commande** correspond à une demande d'**exécution** (Bourreau, au travail !!!).

Le premier argument de la ligne de commande désigne le **programme à exécuter**, qu'il s'agisse d'un **programme interne**, d'un **alias** ou d'un **programme externe**.

L'exécution d'un **programme interne** ne génère pas de nouveau processus.

### 2. Exécution inconditionnelle

Pour exécuter **plusieurs commandes**, sur la même ligne, **sans exprimer de condition**, utiliser l'opérateur ; (point-virgule).

Exemple : ls -ail; cd /bin; pwd

### 3. Exécution conditionnelle

Les opérateurs **&&** et **||** permettent d'exprimer des **conditions** à l'exécution des commandes. Ils sont très utilisés, car il permettent une **écriture beaucoup plus concise** des expressions de conditions.

**Commande\_1 && Commande\_2**

Si **Commande\_1** se termine **correctement**,  
**Commande\_2** sera **exécuté**

**Commande\_1 || Commande\_2**

Si la **Commande\_1** se termine **mal**, la  
**Commande\_2** sera **exécutée**

Le traitement est effectué **de la gauche vers la droite**.

La logique de ces caractères particuliers est fondée sur la **survenance d'erreur**.

### 4. Groupement de commandes

L'utilisation des **parenthèses ()** permet le regroupement de commandes, *avec exécution d'un sous-shell*.

Exemple : cd /usr; (cd /usr/bin; pwd); pwd

(-) déexecute ds un sous shell

L'utilisation des **accolades {}** permet le groupement de commandes, dans le Shell courant, donc *sans exécution d'un sous-Shell*.

Exemple : cd /usr; pwd;{ ls -ail; cd /bin; pwd ; }; pwd

## D) Les variables

Le **Shell** permet d'affecter des **valeurs** (*texte ou nombre*) à des mots (des **variables**), puis d'utiliser ces mots à la place des valeurs affectées.

Ceci revient, par exemple, à utiliser la dénomination "age" plutôt que le nombre 20 (ans) qui, lui, peut (malheureusement !) changer.

La commande **set** permet de visualiser toutes les variables connues par le Shell.

## 1. L'affectation

Il s'agit de donner une valeur à une variable.

Le nom de la variable est formé d'une **chaîne** (de préférence en minuscules) de **caractères ordinaires**. Le **symbole d'affectation** est le =

Le **Shell** associe du texte au nom de la variable, sans **présumer de sa signification**.

Par exemple : chiffre=3, chiffre=003 et chiffre=trois n'ont pas la même signification pour le **Shell**.

La commande **read** permet d'affecter un contenu à une *variable*, à partir d'une **lecture au clavier**.

**read var**

bonjour (tapez le texte au clavier)

**echo \$var**

bonjour

La commande **read** se met en attente du clavier (en attente de la validation par *entrée*, en réalité) pour définir le contenu de la variable **var**

## 2. Le contenu

Il s'agit de récupérer le contenu de la variable.

Pour cela, on utilise le **caractère \$** (rien à voir avec le symbole du prompt).

chiffre=trois

**echo chiffre**

chiffre

**echo \$chiffre**

trois

**rep=/usr/bin**

**cd \$rep**

**pwd**

Important : Une variable non définie ne contient **pas de valeur**.

Etudiez, dans ce qui suit, l'emploi des caractères { et } qui permettent de délimiter les noms de variables.

**echo \$vide**

La variable vide n'est pas définie, rien n'apparaît donc à l'écran.

**echo Prem\$videSuite**

**Prem** La variable videSuite n'est pas définie ! rien n'apparaît donc à l'écran, à part **Prem**.

**echo Prem\${vide}Suite**

**PremSuite** Cette fois, on indique au **Shell** précisément le nom de la variable; La variable désignée est **vide**, qui est vide, d'où l'affichage de **PremSuite**

Essayez maintenant la même manipulation, en affectant préalablement une valeur à la variable *vide* (qui ne le sera plus ...).

### 3. L'interprétation

La commande echo permet d'afficher une suite de caractères, placée entre *guillements*, ou entre *simple quotes*.

Exemples : Les commandes :      echo Bonjour  
 et :                                echo 'Bonjour'  
 et :                                echo "Bonjour"  
 fournissent un résultat identique.

- Mais : 1 Les *guillemets* et les *simples quotes* se verrouillent mutuellement, ce qui permettra de placer, sans souci, des *simples quotes* à l'intérieur de *guillemets*;  
 2 Les variables sont interprétées lorsqu'elles sont placées dans des chaînes de caractères entourées de guillemets, pas à l'intérieur de simple quotes.

Illustration :    a=Tomates  
 echo \$a  
 Tomates  
 echo "Ce sont d'excellentes \$a vertes"  
 Ce sont d'excellentes Tomates vertes  
 echo 'Ce sont d'excellentes \$a vertes'  
 ... Erreur ? ... Que se passe-t-il ? ...  
 echo 'Ce sont de tres bonnes \$a vertes'  
 Ce sont de tres bonnes \$a vertes

#### Le problème des guillemets :

Une variable est interprétée de la même façon, à priori, lorsqu'elle est exploitée "telle que", ou lorsqu'on l'entoure de guillements.

Mais, dans un cas bien particulier, une nuance, bien réelle, peut poser problème.

Sans guillemets, une variable vide est substituée par du **vide**, ce qui provoque une erreur (de fonctionnement, ou de syntaxe, suivant les cas) lors des expressions logiques, exprimées avec la **commande test**

Entourée de guillemets, une variable vide sera substituée par une "**Chaîne vide**", ce qui pourra être interprété correctement par le **Shell**.

- Conclusion : 1 Les *guillemets* permettent de changer du **vide** en "**chaîne vide**";  
 2 C'est un très bon réflexe d'entourer les noms de variables de guillemets.

### 4. Commande unset

La commande **unset** permet l'**annulation d'une affectation** et, donc, l'**initialisation du contenu d'une variable** (sauf pour les variables positionnelles, à voir).

Exemple :    unset rep

### 5. Les variables prédéfinies

Le **Shell** utilise lui-même un **certain nombre de variables** qui ont une signification toute particulière. Ces variables-là sont déclarées en **majuscules**.

L'utilisateur est néanmoins apte à en modifier certaines, pour personnaliser son **Shell**.

La commande *set* permet de visualiser toutes les **variables connues par le Shell**.

La commande *env* propose uniquement les **variables exportées** (voir : *portée des variables*).

Les principales variables d'environnement sont :

- ENV : cette variable contient le **nom du fichier** qui est automatiquement exécuté à chaque démarrage de votre *Shell*. Ce nom dépend du *Shell* utilisé. Vous pouvez placer, dans le fichier désigné par la **variable ENV**, toutes les commandes que vous désirez voir exécuter à chaque début de session (**date**, **echo bonjour ...**, par exemple).
- HOME : chemin d'accès absolu du **répertoire de connexion** (paramètre par défaut de la commande *cd*). Par abus de langage, on utilise parfois ce terme pour parler de son répertoire initial.
- IFS : ensemble des caractères interprétés comme **séparateurs de chaînes de caractères** (espace, tabulation, ...).
- LOGNAME : nom de **connexion de l'utilisateur**.
- MAIL : chemin d'accès absolu du fichier utilisé comme **boîte au lettres**.
- PATH : liste des chemins d'accès absolus des **répertoires consultés** lors d'une recherche de commandes. Lorsque la variable PATH possède le caractère . entre deux :, on peut exécuter, en tapant uniquement son nom, un programme qui se trouve dans le répertoire courant.
- PS1 : valeur du **premier prompt** (prompt = "symbole d'accueil").  
Ex. de codes : \t = Heure      \d = Date      \s = Nom du Shell
- PS2 : valeur du **second prompt** (lorsque la ligne de commande dépasse 80 caractères); PS = *Prompt String*.
- SHELL : le nom du programme *Shell* utilisé.
- TERM : type du **terminal** utilisé pour la session courante.

Les variables ci-dessus sont souvent appelées "**variables d'environnement**", car elles sont exploitées pour gérer l'environnement de travail de *l'interpréteur Shell*.

Les variables suivantes sont **automatiquement exploitées par le Shell** lors de l'exécution d'un programme :

- |                 |   |
|-----------------|---|
| \$#             | Contient le <u>nombre de paramètres</u> de la dernière commande tapée.                                      |
| \$*             | Contient la <u>liste des paramètres</u> de la dernière commande.  |
| \$0             | Le <u>nom de la procédure</u> qui l'utilise, avec le chemin d'accès utilisé pour son lancement.             |
| \$1 \$2 \$3 ... | <u>Paramètres positionnels</u> . Contient l'argument N° 1, N° 2, N° 3, ... transmis à la dernière commande. |
| \$?             | Contient le <u>code de retour</u> de la dernière commande.  |
| \$\$            | Contient le <u>numéro du processus (PID)</u> en cours d'exécution.  |
| \$!             | Contient le <u>numéro du dernier processus (PID)</u> lancé en arrière plan.                                 |
| \$-             | <u>Options</u> fournies à l'interpréteur <i>Shell</i> .   |

## 6. Illustration

Créer, avec *cat*, le petit fichier dénommé *liste*, qui contiendra les lignes suivantes :

```
ls $* | more
echo $#
echo $$
echo $0
```

Pensez à donner tous les droits à ce fichier, en faisant : *chmod 777 liste*

Tester ce petit *shell-script*, en lui fournissant les paramètres attendus, puis le tester à nouveau, avec d'autres *variables automatiques*.

Exemple : *./liste /usr /bin*

## 7. Manipulations

a) Lancer un *Shell* secondaire, en tapant : *bash*, puis changer le prompt :

*PS1=Bonjour>* (ou : *PS1="Bonjour">*)

Votre prompt sera désormais *Bonjour>*

Quitter le sous-shell, avec la commande : *exit*

Que se passe-t-il, au retour ? *Retour sur le prompt défini par défaut*

b) Taper : *set | grep PATH*

Fournit la liste des répertoires dans lesquels le *Shell* recherchera les **commandes que vous saisissez**.

La commande *set*, associée à quelques paramètres, permet d'initialiser les **variables positionnelles**. Cela est très pratique pour **récupérer**, et renvoyer dans l'environnement du script, différents champs de sortie d'une commande.

Utilisation : *set a b c d*  
*echo \$1 \$2 \$3 \$4*

*a b c d*

Illustration des variables positionnelles : Ajouter la ligne *echo \$1 \$2 \$3 \$4* à la fin du script proposé au début de cette page : *liste*

## 8. La portée des variables

Une **variable Shell** est une donnée interne au programme **dans lequel elle est définie**.

Lorsqu'un *Shell* exécute un script, il crée un **processus fils**.

Ce **processus fils** ne connaît pas, à priori, les variables définies par le *Shell père*.

Pour qu'une variable soit connue des processus fils, il faudra préalablement *l'exporter*, au moyen de la commande *export*.

Illustration :

<b>VAR=1</b>	( <i>VAR</i> sera exportée : en tant que variable d'environnement, elle est déclarée en majuscules, par convention).
<b>/bin/bash</b>	(On lance un nouveau <i>Shell</i> , fils du précédent).
<b>echo \$VAR</b>	(Le nouveau <i>Shell</i> fils ne connaît pas la variable var).
<b>exit</b>	(On remonte au niveau du <i>Shell</i> père).
<b>export VAR</b>	(Déclaration de la variable VAR à l'environnement).
<b>echo \$\$</b>	(Visualisation du PID du shell courant).
<b>xxxxx</b>	
<b>/bin/bash</b>	(Exécution d'un <i>Shell</i> fils).
<b>echo \$\$</b>	(Visualisation du PID du <i>Shell</i> fils).
<b>yyyyy</b>	
<b>echo \$VAR</b>	
<b>1</b>	(une fois exportée, <i>VAR</i> est connue des processus enfants).

Sur certaines versions, d'origine **BSD** (comme le *C-Shell*), la fonction **setenv** permet de déclarer la valeur d'une variable à un environnement, sous la forme : **setenv VAR 1**

Remarques : Il n'y a aucune possibilité pour qu'une variable définie dans un processus fils soit connue du père ! Donc, attention : toute **modification de variable** dans un script sera perdue, au retour dans le *Shell père*.

Une erreur fréquente résulte de l'utilisation de la commande **cd** dans un script. A la fin d'un script, on se retrouve au même endroit que lorsqu'on l'a lancé (l'environnement du père n'est pas modifié).

## 9. Exercices

- 1) Vérifier si votre variable **PATH** possède le répertoire courant (en tapant, évidemment : **echo \$PATH**).  
Si ce n'est pas le cas rajoutez le par : **PATH=\$PATH:**  
Et vérifiez le résultat, avec : **echo \$PATH**
- 2) Ajoutez, de la même façon, des répertoires de votre cru à la définition de la variable **PATH**
- 3) Sortez à l'écran, par la commande **echo** et un judicieux verrouillage des caractères spéciaux, à l'aide du symbole **\**, le texte suivant :  
**"> je est un autre ! <"**
- 4) Parmi ces noms de variables, lesquels sont corrects ?  
**abc ABC \_123 a.2 a\_2 2a**

## E) Substitution de commande

Il est possible d'affecter, à une variable, la sortie standard d'une commande.

Pour cela, on affecte à la variable la commande placée entre les **caractères `**

## 1. Voici le mécanisme

```
pwd  
/users/jean/bin  
repertoire='pwd'
```

Le résultat de la commande `pwd` (`/users/jean/bin`) sera affectée à la variable `repertoire`, au lieu d'être envoyé à l'écran.

```
echo $repertoire  
/users/jean/bin  
ls $repertoire
```

Le contenu de la variable `repertoire` est exploitable par d'autres commandes.

## 2. Remarques

La substitution de commandes permet de pallier la contrainte liée à la portée des variables.

Ainsi, si les variables définies dans un processus fils ne peuvent pas remonter jusqu'au processus père, celui-ci pourra, par contre, récupérer la valeur d'une variable de son fils.

Illustration :

```
cat > varfils  
var=1  
echo $var  
Ctrl + D  
chmod 777 varfils  
var='./varfils'
```

Le processus père récupère le contenu de `var`, définie dans le processus fils.

La substitution est une technique proche du "*tube*", ou "*pipe*".

En effet, le "*tube*" permet à une Entrée Std de processus de récupérer le contenu de la Sortie Std d'un autre processus, alors que la *substitution* permet à un processus de demander des éléments sur la Sortie Std d'un autre processus.

On peut comparer ces deux outils avec le schéma suivant :

Tube :	Processus A   Processus B
Substitution :	Processus B 'Processus A'

## F) La commande expr (let)

Les variables, en *Shell*, sont toujours traitées sous forme de chaînes de caractères.

Si on manipule une variable numérique, et si on souhaite lui faire subir des opérations arithmétiques, il faudra utiliser la commande `expr`.

Les arguments de `expr` sont traités comme un lot d'opérandes et d'opérateurs.

Ainsi, la commande `expr` évalue l'expression. Son résultat est ensuite envoyé sur la sortie standard.

Les principaux opérateurs sont :

Opérateurs de comparaison : `>` `<` `>=` `<=` `=` (test d'égalité) `!=` (différent de);

Opérateurs arithmétiques : `+` `-` `*` `/` `%` (% = reste de la division entière)

(Penser à utiliser le symbole de protection `\`, chaque fois qu'un opérateur a une autre signification dans Linux : `*`, `|`, `&`, `>`, `<`, ...);

Exemple d'utilisation :

```
nombre=1
expr $nombre + 3      (attention aux espaces, obligatoires)
4
```

Deux opérateurs logiques : | &

- **exp1 | exp2** : a pour valeur l'expression *exp1* si *exp1* n'est pas nulle.  
Sinon a pour valeur *exp2*;
- **exp1 & exp2** : a pour valeur l'expression *exp1* si ni *exp1* ni *exp2* ne sont nulles ou vides, et sinon 0.

Pour les tests arithmétiques, la valeur retour de la commande est :

- ✗ 0 si le résultat est non nul,
- ✗ 1 si le résultat est nul (en principe ...),
- ✗ 2 en cas d'erreur de syntaxe,
- ✗ 3 pour une autre erreur.

Exemple 1 : nb=1  
 a=`expr nb \\* 0`  
*expr: argument non numérique*  
 echo \$?  
 2

Exemple 2 : nb=0  
 tt=3  
 expr \$nb \| \$tt  
 3  
 nb=1  
 expr \$nb \| \$tt  
 1

## G) La commande test

La commande *test* permet de vérifier la réalisation d'une condition.

Elle renvoie la valeur 0 si le résultat du test est **vrai**, un résultat **faux** renvoyant une valeur différente de 0.

La variable spéciale **\$?** est l'outil privilégié pour obtenir, et traiter le résultat.

Voici les options de cette commande :

### Tests sur les fichiers :

-r <i>fich</i>	vrai si le fichier <i>fich</i> existe et s'il est accessible en lecture;
-w <i>fich</i>	vrai si le fichier <i>fich</i> existe et s'il est accessible en écriture;
-x <i>fich</i>	vrai si le fichier <i>fich</i> existe et s'il est exécutable;
-f <i>fich</i>	vrai si <i>fich</i> existe et si ce n'est pas un répertoire;
-d <i>fich</i>	vrai si <i>fich</i> existe et si c'est un répertoire;
-s <i>fich</i>	vrai si <i>fich</i> existe et s'il n'est pas vide;

Tests sur les chaînes :

<b>-z chain</b>	vrai si la longueur de la chaîne <i>chain</i> est zéro;
<b>-n chain</b>	vrai si la longueur de la chaîne <i>chain</i> n'est pas nulle; (A valider : ne fonctionne pas sous <i>bash</i> , ni sous <i>ksh</i> ...)
<b>chain1 = chain2</b>	vrai si les chaînes <i>chain1</i> et <i>chain2</i> sont égales;
<b>chain1 != chain2</b>	vrai si les chaînes <i>chain1</i> et <i>chain2</i> ne sont pas égales;

Tests sur les nombres :

<b>num1 -eq num2</b>	vrai si les entiers <i>num1</i> et <i>num2</i> sont égaux;
<b>num1 -ne num2</b>	vrai si les entiers <i>num1</i> et <i>num2</i> sont différents;
<b>num1 -lt num2</b>	vrai si <i>num1</i> est inférieur à <i>num2</i> ;
<b>num1 -gt num2</b>	vrai si <i>num1</i> est supérieur à <i>num2</i> ;
<b>num1 -le num2</b>	vrai si <i>num1</i> est inférieur ou égal à <i>num2</i> ;
<b>num1 -ge num2</b>	vrai si <i>num1</i> est supérieur ou égal à <i>num2</i> .

La commande *test* peut être combinée avec les opérateurs logiques suivants :

**!** (non), **-a** (and/et), **-o** (or/ou).

Voici quelques exemples :

```
test -d /bin
echo $?
0          Oui, le répertoire /bin existe
```

L'exemple ci-dessous exprime la condition  $((n > 0) \text{ et } (n \leq 5))$  :

```
n=8
test $n -gt 0 -a $n -le 5
echo $?
1          Non, 8 n'est pas compris entre 0 et 5
```

```
nb1=3; nb2=5
test `expr $nb1 > $nb2`      fausse si < sans msg d'erreur
echo $?                         test $nb1 -gt $nb2
1          Non, nb1 n'est pas plus grand que nb2
```

## H) Programmation : Shell Scripts

Les *Shell Scripts* présentent un intérêt évident, lorsqu'une même séquence de commandes doit être exécutée à plusieurs reprises.

Il est même possible d'écrire de véritables programmes.

Voici, ci-dessous, un exemple.

Créez le fichier nommé "*activite.sh*", avec *vi*, contenant les lignes suivantes :

```
#!/bin/bash
# Shell Script fournissant des informations sur un utilisateur
echo -e "Activité de l'utilisateur : $1 \c"
echo -e "connecté depuis \c"
who | grep $1 | cut -c19-38
echo " processus en cours "
ps -Af | grep ^$1 | grep -v "grep"           PS-- fu $1
```

Remarque : L'option *-e* de *echo* permet de traiter les séquences spéciales, comme *\c*

Autorisez l'exécution de votre programme "*activite.sh*" avec :

```
chmod u+x activite.sh
```

et exécuter le programme avec, pour paramètre, le **nom d'utilisateur** d'une personne connectée sur le système.

Repérez l'utilisation faite de la variable système *\$1*

Toute ligne commençant par un *#* représente un **commentaire**, et n'est donc pas exécutée par le **Shell**.

Il est important de toujours placer des **commentaires clairs** dans vos programmes, pour vous en remémorer ultérieurement l'utilité.

## I) Structures de contrôle

Rentrons maintenant dans le vif du sujet de la **programmation** de ces **Shell scripts**.

Il faut, bien évidemment, **tester tous les exemples proposés** pour bien comprendre le fonctionnement de chacune de ces structures.

De même, seuls la **pratique** et le **temps** permettent d'être à l'aise avec ces *notions de programmation*.

### 1. L'instruction conditionnelle (*if...then...else...fi*)

Cette instruction permet d'**exprimer une condition** et d'**effectuer des opérations différentes en fonction du résultat de cette condition**.

Exemple :

```
if test -f "$1"
then
    echo "$1 existe et n'est pas un répertoire"
else
    echo "$1 n'existe pas ou est un répertoire"
fi
```

Si le résultat de la condition est **vrai**, alors les actions indiquées après le *then* seront exécutées, suivies d'un branchement sur le *fi*.

Dans le cas contraire, les actions proposées après le *else* seront exécutées.

Les mots **then** (alors) et **else** (sinon) doivent se trouver seuls sur une ligne.

Format général :

```
if condition
then
    action
else
    action
fi
```

## 2. La boucle (for...do...done)

La boucle **for** permet d'exécuter une liste d'actions, pour une liste de valeurs données.

Exemple :

```
for nom in jean paul jacques
do
    echo "bonjour $nom"
done
```

La boucle **for** a exécuté la commande **echo**, placée entre le **do** et le **done**, pour chaque valeur de la variable **nom**, les valeurs étant prises successivement dans la liste après le **in**.

La variable de **contrôle de la boucle** (dans l'exemple : *nom*) prend successivement les differentes valeurs possibles exprimées avec **in ...**

Exemple :

```
for nom in `who | cut -c1-8`
do
    echo "$nom : " `grep "^$nom" /etc/passwd 2>/dev/null | cut -d: -f3`"
done
```

L'exemple ci-dessus affiche le numéro d'utilisateur: **UID (User IDentity)**, pour tous les utilisateurs **connectés au système**.

Cette valeur est trouvée dans le fichier **/etc/passwd**

Remarques : Voici les possibilités offertes pour construire l'expression de la liste :

- for var in liste**
- for var in \$listevARIABLE**
- for var in \$\***
- for var in `commande`**

La boucle est réalisée jusqu'à **épuisement des variables**, dans la liste après le **in**.

Si la liste de variables est vide, la boucle n'est pas exécutée.

Format général :

```
for chaîne in liste de valeurs
do
    Liste de commandes
done
```

### 3. La répétition (while...do...done)

La structure ***while*** permet de répéter une action tant que le résultat d'une condition est vrai (code de retour égal à 0).

Tant que le résultat de la condition est vrai, exécution de la suite de commandes située entre le ***do*** et le ***done***, puis renouvellement du test exprimé dans la condition.

Exemple :

```
echo -e "entrez un nom de fichier : \c"
read nomfichier
while test -z $nomfichier
do
    echo "chaîne vide interdite"
    read nomfichier
done
```

Tant que ***nomfichier*** est vide, on exécute le ***do***.

Format général :

```
while condition
do
    liste de commandes
done
```

Autre exemple : Les lignes du fichier toto sont transférées une par une dans la variable var

```
while read var; do echo "*** var ***"; done < toto
```

### 4. La sélection multiple (case...in...esac)

Proche du ***if***, le ***case*** permet d'en éviter une longue série, et permet de sélectionner un traitement parmi ***n***.

Voici la procédure ***casein***, à tester en 4 temps :

- a) Taper : ***casein*** et valider;
- b) Taper : ***casein a*** et valider.
- c) Taper : ***casein a b*** et valider.
- d) Taper : ***casein a b c*** et valider.

```
case $# in
    1) echo "un paramètre ca va !";;
    2) echo "deux paramètres ca va encore";;
    *) echo "nombre de paramètre incorrect : bonjour les dégâts";;
esac
```

Le ***Shell*** compare le paramètre donné en entête avec chacun des modèles proposés : dans l'exemple ci-dessus, avec les modèles ***1***, ***2*** et ***\****

Lorsqu'il y a **coincidence** (égalité), les commandes correspondantes sont exécutées, et le ***Shell*** continue son exécution après le délimiteur ***esac*** (***case***, à l'envers ...).

Format général :

```

    case chaine in
        modele1) Liste de commandes;;
        modele2) Liste de commandes;;
        ...
        modelex) Liste de commandes;;
    esac

```

## 5. Boucle until

Cette instruction fonctionne comme le *while*, mais à l'envers.

```

until condition
do
    liste de commande
done

```

La liste de commandes (entre le *do* et le *done*) s'exécute jusqu'à ce que la condition exprimée prenne la valeur vrai.

# J) Instructions diverses

### a) exit

La commande *exit n* termine un Script, avec la valeur *n* comme code de retour.

### b) eval

L'instruction *eval* respecte la syntaxe : *eval arguments*

Où *arguments* = lot de commandes, lues et concaténées en une seule, pour obtenir un code de retour lié à un groupe de commandes.

### c) clear

L'instruction *clear* s'utilise sans argument; Elle efface le contenu de l'écran.

### d) shift

*shift* s'utilise également sans argument.

Cette instruction va décaler le contenu des variables positionnelles.

Après son exécution, \$1 aura reçu le contenu de \$2, qui aura reçu lui-même le contenu de \$3, et ainsi de suite.

Cette instruction fait sauter la limite théorique de 9 variables positionnelles.

### e) break (Structure de contrôle)

L'instruction *break* permet d'interrompre une boucle avant la fin. Les structures *for*, *while* et *until* sont concernées.

Après un *break*, c'est l'instruction qui suit le *done* qui sera exécutée.

Le seul paramètre disponible est une valeur entière, qui précise après combien de boucles l'interruption doit intervenir : *break 3*

Remarque : L'instruction *break* est considérée comme une instruction de branchement, donc critiquée ...

f) **continue** (Structure de contrôle)

*continue* demande au *shell* d'effectuer le **prochain passage** de la boucle, donc remonte au **test de l'entête de boucle**.

*continue* fonctionne comme le *break*, avec la même syntaxe (*continue 1* est égal à *continue*, *continue 3* "saute" 3 boucles).

g) **readonly**

**Utilisé seul :** Affiche la liste des variables en état **readonly**;

**Avec l'argument var :** Affecte l'état **readonly** à la variable var.

**Attention :** Il n'y a pas de **retour possible**, lorsqu'une variable passe en **readonly**.

h) **trap**

Cette instruction permet de **conditionner une action par une autre**.

**Exemple :**

trap \$HOME/fin exit

exécute la procédure *fin* lors de la sortie de la procédure en cours.

## K) Exemples

### 1. if

```
if test -d $1
then
    echo "le répertoire $1 contient :"
    ls -l $1
else
    echo "$1 n'est pas valide"
fi
```

La commande *test\_rep.sh* (ci-dessus) vérifie si **le paramètre** est un **directory**.

Dans l'affirmative, le **Shell** exécute la branche **then**, sinon, il exécute la partie **else**.

Le *fi* est là pour délimiter la fin du *test*.

### 2. for

Le grand classique des **exemples de boucles** consiste à afficher chaque élément d'une liste.

```
for jour in Lundi Mardi Mercredi Jeudi
do
    echo "Aujourd'hui : $jour, vivement dimanche"
done
```

**Le résultat sera :**

*Aujourd'hui Lundi, vivement dimanche*  
*Aujourd'hui Mardi, vivement dimanche*

...

### 3. while

```
v=0
while test $v -le 10
do
    echo "compteur $v"
    v=`expr $v + 1`
done
```

L'exécution de la boucle ci-dessus, nommée *boucle.sh*, fournira :

```
compteur 0
compteur 1
...
compteur 10
```

### 4. case

```
case $1 in
    lundi) echo "nous sommes le " `date`  

            pwd; ls -l  

            echo "bonne semaine";;  

    mardi) echo "bonne journée";;  

    mercredi) echo "bonne journée";;  

    jeudi) echo "bonne journée";;  

    vendredi) echo "bon week-end";;  

    *) echo "pas le samedi, ni le dimanche";;  

esac
```

Rappel : Le ; est le séparateur de commandes.

### 5. trap

Utiliser la commande interne **trap** pour afficher un message vigoureux, lorsque le script reçoit un signal 1.

Script recevant le signal : trap /bin/message 1

Script /bin/message : echo "Maurice, tu pousses le bouchon un peu trop loin !!!"

## L) Exercices : Correction

- Sortez à l'écran, par la commande **echo** et un judicieux verrouillage des caractères spéciaux à l'aide du symbole \, le texte suivant :  
 "> je est un autre ! <"  
**echo "\> je est un autre ! <\\""**

- Parmi ces noms de variables, lesquels sont corrects ?

abc	ABC	_123	a.2	a_2	2a
<i>abc</i>	<i>ABC</i>	<i>_123</i>			
<u>a_2</u>					