Le Bourne Again Shell (BASH)

Introduction

Une connaissance fonctionnelle de la programmation Shell est essentielle à quiconque souhaite devenir efficace en administration système. Pensez qu'au démarrage de la machine Linux, des scripts Shell du répertoire `/etc/rc.d` sont exécutés pour restaurer la configuration du système et permettre la mise en fonctionnement des services. Une compréhension détaillée de ces scripts de démarrage est importante pour analyser le comportement d'un système, et éventuellement le modifier.

Écrire des scripts Shell n'est pas difficile à apprendre car, d'une part, les scripts peuvent être construits par petites sections et, d'autre part, il n'y a qu'un assez petit nombre d'opérateurs et d'options spécifiques au Shell à connaître. La syntaxe est simple et directe, similaire à une suite d'appels de différents utilitaires en ligne de commande et il n'existe que peu de « règles » à apprendre. La plupart des petits scripts fonctionnent du premier coup et le débogage, même des plus longs, est assez simple.

Quand ne pas utiliser les Scripts Shell

- Pour des tâches demandant beaucoup de ressources et particulièrement lorsque la rapidité est un facteur (tri, hachage, récursion...) ;

- Pour des procédures impliquant des opérations mathématiques nombreuses et complexes, spécialement pour de l'arithmétique à virgule flottante, des calculs à précision arbitraire ou des nombres complexes (optez plutôt pour le C++ ou le Java) ;

- Pour une portabilité inter-plates-formes (utilisez le C ou Java à la place) ;

- Pour des applications complexes où une programmation structurée est nécessaire (typage de variables, prototypage de fonctions, etc.) ;

- Pour des projets consistant en de nombreux composants avec des dépendances inter-verrouillées ;

- Si le support natif des tableaux multidimensionnels est nécessaire ;

- Si vous avez besoin de structures de données, telles que des listes chaînées ou des arbres ;

- Si vous avez besoin de générer ou de manipuler des graphiques ou une interface utilisateur (GUI) (Il existe des solutions pour réaliser des dialogues simples avec le Script Shell comme le projet « zenity » par exemple);

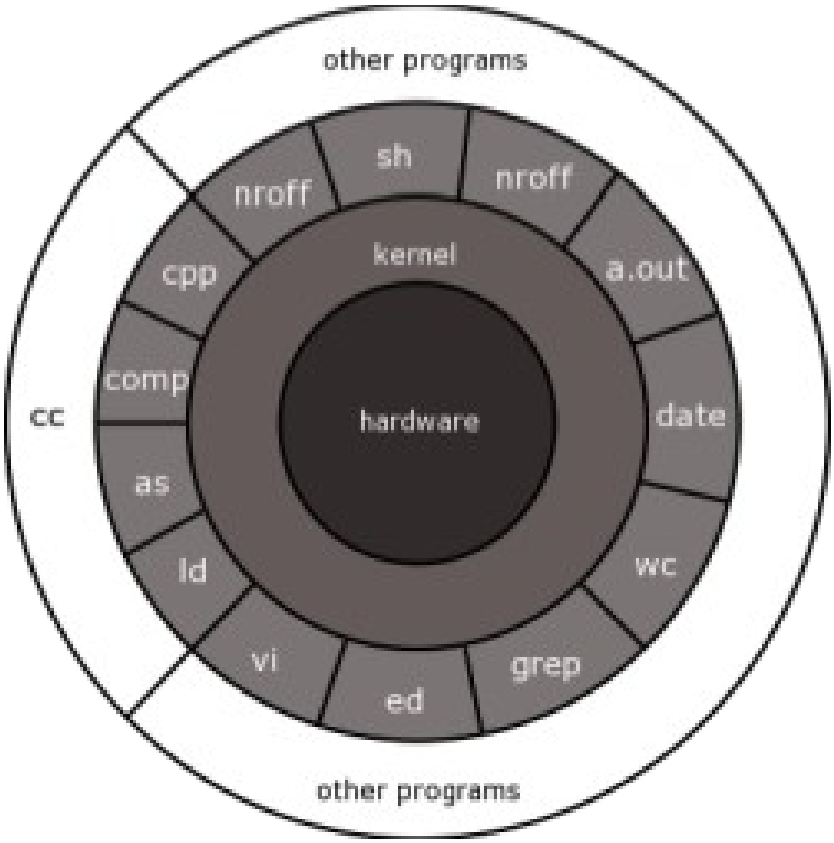
- Lorsqu'un accès direct au matériel est nécessaire ;

- Si vous avez besoin d'utiliser des bibliothèques ou une interface propriétaire ;

- Pour des applications propriétaires, à sources fermées (les sources des Shell sont forcément visibles par tout le monde).

Dans tous les autres cas, le Script Shell est tout a fait conseillé

1) Structure d'un Système UNIX ou Linux



Un Shell est une couche applicative, qui vient se greffer entre le « Kernel »(noyau qui contient toutes les API (sous formes de librairies voir la documentation https://www.kernel.org/doc/Documentation/) permettant de dialoguer avec le HardWare de votre machine) et l'utilisateur (c'est une IHM rudimentaire en mode ligne de commande CLI), effectivement un Shell à pour vocation principale de mettre à disposition de l'utilisateur une console qui attend les ordres via un certains nombres de mots clés d'un langage que l'on nommera langage Shell.
Cette console, va permettre d'exploiter les capacités du hardware au travers de commandes internes au langage Shell, mais aussi de permettre d'exécuter d'autres commandes non incluses dans le Shell et complémentaires du Shell (Applications).

On retrouve d'autres programmes qui utilisent directement les API du noyau comme les outils de compilations (assembleur « as ») et bien d'autres applications qui permettent d'exploiter le hardware de la machine.

l'accès à un Shell se présente généralement sous deux formes, soit sous la forme d'un mode dit console (CLI), ce mode est le plus ancien, ou un mode dit GUI, une fenêtre graphique qui présente les mêmes fonctionnalités que le mode console, mais avec tous les avantages du mode graphique(redimensionnement, copier coller, etc).

2) Les diffèrent Shell

- Bourne Again Shell "bash" (projet gnu) est fondé sur le Bourne Shell "sh" en l'améliorant et reprenant certaines fonctionnalités du Korn Shell et du C Shell.

- Dash (Debian Almquist Shell), petit, rapide et conforme a POSIX, c'est un descendant direct de la version Almquist Shell (ash) de NetBSD . Il à été porté vers Linux par Herbert Xu. Ne possédant à ce jour pas toutes les fonctionnalités de Bash, mais s'y approche très fortement.

- C Shell par Bill Joy (csh) et tcsh Tenex C Shell son équivalent sous Linux utilise une syntaxe proche du C.

- Korn Shell (ksh) ou (pdksh) par David Korn est compatible avec le Bash et reprend un grand nombres de fonctionnalités du C Shell.

Un Shell possède un double aspect :

1 Un aspect *environnement de travail. (mode console interactif)*
2 Un aspect *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.

Attention :**sh** parfois est un lien symbolique sur **bash** ou **dash** et ne transmet pas le contexte des variables système au sous Shell appelé par un script…(toujours vérifier le lien symbolique sh)

Connaître votre Shell par défaut :
`$> echo $SHELL` => donne le chemin du Shell par défaut
`$> /bin/bash`

Obtenir la version du Bash Shell (exemple)

`$> /bin/bash --version` ou `$> echo $BASH_VERSION`

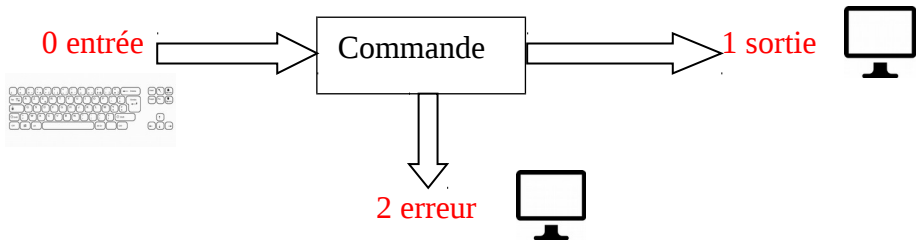
3) Les redirections standards

Un processus UNIX possède par défaut trois voies d'interactions avec l'extérieur appelées *entrées / sorties standard* identifiées par un entier positif ou nul appelé *descripteurs de fichiers* et une *sortie erreur*.

Ces entrées / sorties standard sont :

- une *entrée standard*, de descripteur **0 (stdin)**
- une *sortie standard*, de descripteur **1 (stdout)**
- une *sortie standard pour les messages d'erreurs*, de descripteur **2 (stderr)**.

Toute commande étant exécutée par un processus se vois attribué ces trois descripteurs(*) par défaut, nous dirons également qu'une commande possède une entrées et deux sorties standard et une sortie erreur par défaut.



De manière générale, une commande de type filtre (ex : **cat**) prend ses données sur son entrée standard qui correspond par défaut au clavier, affiche ses résultats sur sa sortie standard, par défaut l'écran, et affiche les erreurs éventuelles sur sa sortie standard pour les messages d'erreurs, par défaut l'écran également.

(*)Un descripteur de fichier est simplement un numéro que le système d'exploitation affecte à un fichier ouvert pour garder sa trace. Considérez cela comme une version simplifiée d'un pointeur de fichier. C'est analogue à un handle vers un fichier en C.

3.1 Symboles des redirections des entrées sorties

cmd < fichier	l'entrée standard est lu à partir d'un fichier
cmd > fichier	La sortie standard est redirigée dans un fichier (RAZ du fichier)
cmd >> fichier	La sortie standard est redirigée dans un fichier (concaténation du fichier)
> fichier **: > fichier**	Créer un fichier vide (voir cmd touch également)
cmd>& fichier **cmd &> fichier**	La sortie standard et les erreurs sont redirigées dans un fichier (RAZ du fichier)
cmd >>& fichier **cmd &>> fichier**	La sortie standard et les erreurs sont redirigées dans un fichier (concaténation du fichier)
cmd 1> fichier	Rediriger stout vers un fichier(RAZ du fichier)
cmd 1>> fichier	Rediriger stdout vers un fichier(concaténation du fichier)
cmd 2> fichier	Les erreurs sont redirigées vers le fichier (RAZ du fichier)
cmd 2>> fichier	Les erreurs sont redirigées vers le fichier (concaténation du fichier)
cmd 2>&1 fichier	Les erreurs sont redirigées vers la sorties standard

cmd >/dev/null	La sortie standard est redirigé vers la poubelle UNIX.
exec df<> fichier	Ouvrir le fichier « fichier » et l'affecter au descripteur n° df
cmd <&df	Lire dans le descripteur n°df (doit être ouvert)
cmd >&df	Écrire dans le descripteur n°df (doit être ouvert)
exec df>&-	Fermer le descripteur n°df
0<&- ou <&-	Fermer le descripteur stdin
1>&- ou >&-	Fermer le descripteur stout

Quelques exemples de redirections :

```
$> cat /etc/profile > profile.back
$> ls -l toto >>& liste.txt

$> exec 3<>profile.back
$> cat <&3
$> exec 3<>&-

$> > fichier_vide.txt
$> :>fichier-vide.txt
```

3.2) Redirection par les pipes

Le signe | est dit pipe, c'est un conduit entre deux ou plusieurs applications (processus).

processus1 | processus2

Exemple : cat fichier.txt | wc -l

Équivaut à faire ceci :
processus1 > fichier
processus2< fichier

proc1 |& proc2 : proc2 reçoit en entrée les sorties standards et les erreurs de proc1.

Exemple :
```
$> find / -name *lsio* |& wc -l
```

4) Mode d'exécution d'un script

4.1) Type de commandes

Une *commande interne* est une commande dont le code est implanté au sein de l'interpréteur de commande. Cela signifie que, lorsqu'on change de Shell courant ou de connexion, par exemple en passant de **bash** au **C-Shell**, on ne dispose plus des mêmes commandes internes.

Exemples de commandes internes : **cd** , **echo** , **for** , **pwd, [,]**

Une *commande externe* est une commande dont le code se trouve dans un fichier ordinaire exécutable.
Le Shell crée un processus pour exécuter une commande externe.

Parmi l'ensemble des commandes externes que l'on peut trouver dans un système, nous utiliserons principalement les *commandes UNIX* (ex : **test**,**ls**, **mkdir**, **vi**, **sleep**) et les *fichiers Shell*.

La localisation du code d'une commande externe doit être connue du Shell pour qu'il puisse exécuter cette commande. A cette fin, **bash** utilise la valeur de sa variable prédéfinie **PATH**. Celle-ci contient une liste de chemins séparés par le caractère **:** (ex : */bin:/usr/bin*).

Remarque : pour connaître le statut d'une commande, on utilise la commande interne **type**.

Ex :
```
$> type sleep
$> sleep is /bin/sleep => sleep est une commande externe
$> type echo
$>echo is a Shell builtin
$>
```

4.2) Exécution

Un Script est un fichier contenant uniquement des caractères ASCII, il est constitué de commandes internes du Shell, mais également de commandes externe au Shell et possédera les logiques classiques que l'on retrouve dans la programmation, comme des boucles d'itérations, des commandes de test, des commandes de sauts, des fonctions etc.

Le nom d'un fichier script ne nécessite pas d'extension, en mettre une n'est pas une erreur, il est possible de donner un nom long qui est plus explicite qu'un nom de fichier court.

Attention aux espaces dans les noms de fichiers !

Exemple
sc1 est moins clair que **Mon_Premier_Script.sh**

Astuce : Il est intéressant de mettre une extension de type « .sh » par exemple ce qui permet de repérer facilement que ce fichier est un script Shell, ce qui n'est pas forcement évident pour un script sans extension et dont les droits d'exécutions sont posés (ceci est assez confus pour les débutants sous Linux).

Exécution d'un script :

Il est possible d'appeler l'interpréteur Shell directement

```
$> bash nom_du_script.sh
```

Dans ce cas on fait appel explicitement à l'interpréteur Shell désiré (ici le bash, on peut utilisé n'importe qu'elle Shell installé sur la distribution utilisée dans la mesure ou les commandes utilisés dans le script Shell sont compatibles avec le contenu de votre script).

Une autre manière de lancer un script est de le rendre autonome (éxécutable).

a) Dans un premier temps il faut changer les droits du fichier script de la manière suivante :

Rendre le fichier exécutable pour l'utilisateur en cours avec l'utilitaire chmod

```
$> chmod u+x nom_du_script.sh
```

b) Puis il suffit de taper le nom du fichier sur la ligne de commande pour l' exécuter (comme un exécutable binaire) :

`$> ./nom_du_script.sh` (+ touche entrée)

Attention : Pour forcer l'exécution du fichier en Shell, le fichier doit commencer obligatoirement par le commentaire spécial ***#!/bin/bash*** (Dite ligne SheBang)

Un script correspond à une suite de commandes écrites dans un fichier.

Exemple :

Prog.sh

```
#!/bin/bash    # interpréteur
commande1
commande2  # même chose que : commande1; commande2
```

4.3) Notion sur les droits des fichiers

Pour voir les différents droit des fichiers, exécutons la commande « `ls -l` » dans un terminal ou une console :

on obtient ceci :

-rw-r--r-- 1 herve herve 670567 2015-02-01 22:32 Freedom.tar.gz
drwxr-xr-x 5 herve herve 4096 2015-02-04 22:30 Roms
drwx------ 2 herve herve 4096 2015-02-05 18:40 ruby
lrwxrwxrwx 1 herve herve 9 2015-02-07 14:26 www -> /var/www/

Nous allons, pour le moment, nous intéresser aux lettres situées à gauche : (*drwxr-xr-x)*

La première lettre désigne le type de fichier :

- - : fichier "classique" régulier
- d : répertoire (directory)
- l : lien symbolique (link)

Passons ensuite aux droits des fichiers à proprement parlé :

- - : aucun droit
- r : read (droit de lecture)
- w : write (droit d'écriture)
- x : execute (droit d'exécuter un fichier ou d'ouvrir un répertoire)

Ces droits sont affichés de la sorte :

Les 3 premiers symboles, sont les droits du propriétaire du fichier (USER), les trois suivants du groupe (GROUP) et les trois derniers des autres utilisateurs (OTHER).

Par exemple, pour cette ligne :
-rw-r--r-- 1 herve herve 670567 2015-02-01 22:32 Freedom.tar.gz

Ici, l'utilisateur (en général, celui qui a créé le fichier ou le dossier) est *herve* et le groupe est également *herve* *(herve:herve <=> user:group)*

Les droits sur ce fichier sont donc les suivants :
user (3 premiers symboles) : rw- (droits de lecture, écriture, pas de droit d'exécution)
group (3 symboles suivants) : r-- (droit de lecture uniquement)
pour les autres (others) (3 derniers symboles) : r-- (droit de lecture uniquement)

Ceci est a retenir sous la forme « **UGO User/Group/Other** »

Notez que pour qu'un dossier puisse être ouvert par un utilisateur, il doit avoir les droits d'exécution.

4.4) Comment modifier les droits d'un fichier

Il y a deux façons de modifier les droits d'un fichier : la manière "relative" qui consiste à modifier les droits existants et la manière "absolue" qui consiste à créer les droits à partir de rien (non vu ici).

La manière "relative"

Elle consiste, à modifier les droits existants.
Par exemple, on souhaite simplement ajouter les droits d'exécution au groupe (en plus des droits existants).
Cela ne modifie donc que le droit d'exécution, les autres droits sont conservés tels quels.

Rappel : Signification des lettres utilisées ci-dessous
- u : user (utilisateur)
- g : group (groupe)
- o : other (autres)
- a: all (tout le monde)

Par exemple, nous souhaitons simplement ajouter les droits d'exécution au groupe. Nous devons donc exécuter cette commande :

```
$> chmod g+x nom_fichier
```

Un autre exemple, nous souhaitons ajouter les droits d'écriture au groupe et supprimer le droit de lecture aux autres (utilisateur non propriétaire du fichier et ne faisant pas partie du groupe du fichier) :

```
$> chmod g+w o-r mon_fichier
```

Enfin, nous souhaitons donner tous les droits à tout le monde (c'est une mauvaise idée) :

```
$> chmod a+rwx mon_fichier
```

5) Modes d'exécution d'une commande

Deux modes d'exécution peuvent être distingués :
1 l'exécution séquentielle.
2 l'exécution en arrière-plan.

5.1) Exécution séquentielle

Le mode d'exécution par défaut d'une commande est l'exécution séquentielle.
Le Shell lit la commande entrée par l'utilisateur, l'analyse, la pré-traite et si elle est syntaxiquement correcte, l'exécute.

Une fois l'exécution terminée, le Shell effectue le même travail avec la commande suivante.

L'utilisateur doit donc attendre la fin de l'exécution de la commande précédente pour que la commande suivante puisse être exécutée : on dit que l'exécution est *synchrone*.

Si on tape la suite de commandes :

`$> sleep 3` ***entrée*** **`date`** ***entrée***

Où ***entrée*** désigne la touche *entrée*, l'exécution de **date** débute après que le délai de 3 secondes se soit écoulé.

Pour lancer l'exécution séquentielle de plusieurs commandes sur la même ligne de commande, il suffit de les séparer par un caractère « **;** »

Ex :
```
$> cd /tmp ; pwd; echo bonjour; cd ~; pwd
```

/tmp => affichée par l'exécution de *pwd*
bonjour => affichée par l'exécution de *echo bonjour*
/home/tondeur => affichée par l'exécution de *pwd*

Pour terminer l'exécution d'une commande lancée en mode synchrone, on appuie simultanément sur les touches *CTRL* et *C* (notées **control-C** ou **^C**).
En fait, la combinaison de touches appropriée pour arrêter l'exécution d'une commande en mode synchrone est indiquée par la valeur du champ *intr* lorsque la commande UNIX **stty** est lancée :

Ex : $> **stty -a**
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
. . .

Supposons que la commande **cat** soit lancée sans argument, son exécution semblera figée à un utilisateur qui débute dans l'apprentissage d'un système UNIX. Il pourra utiliser la combinaison de touches mentionnée précédemment pour terminer l'exécution de cette commande.

Ex :

```
$> cat
     ^C
```

5.2) Exécution en arrière-plan

L'exécution en arrière-plan permet à un utilisateur de lancer une commande et de récupérer immédiatement la main pour lancer « en parallèle » la commande suivante (parallélisme logique). On utilise le caractère **&** pour lancer une commande en arrière-plan.

Dans l'exemple ci-dessous, la commande *sleep 5* (suspendre l'exécution pendant 5 secondes) est lancée en arrière-plan. Le système a affecté le numéro d'identification 696 au processus correspondant tandis que **bash** a affecté un numéro de travail (ou numéro de job) égal à *1* et affiché *[1]*. L'utilisateur peut, en parallèle, exécuter d'autres commandes (dans cet exemple, il s'agit de la commande **ps**). Lorsque la commande en arrière-plan se termine, le Shell le signale à l'utilisateur après que ce dernier ait appuyé sur la touche entrée.

Ex : $> **sleep 5 &**

```
     [1696 ] $ ps
     PID TTY TIME CMD
     683 pts/0 00:00:00 bash
     696 pts/0 00:00:00 sleep
     697 pts/0 00:00:00 ps
```

$> => l'utilisateur a appuyé sur la touche *entrée* mais *sleep* n'était pas terminée
$> => l'utilisateur a appuyé sur la touche *entrée* et *sleep* était terminée

```
     [1]+ Done sleep 5
   $>
```

Ainsi, outre la gestion des processus spécifique à UNIX, **bash** introduit un niveau supplémentaire de contrôle de processus. En effet, **bash** permet de stopper, reprendre, mettre en arrière-plan un processus, ce qui nécessite une identification supplémentaire (numéro de job).

L'exécution en arrière-plan est souvent utilisée lorsqu'une commande est gourmande en temps CPU (ex : longue compilation d'un programme).

Nb : vous pouvez utilisez la commande « fg + numéro de job » pour remettre un processus d'arrière plan en avant plan.

6) Le Scripting Shell

Le langage Script Shell est un langage de programmation qui permet de manipuler les fonctionnalités d'un système UNIX ou UNIX like comme LINUX, configuré pour fournir à l'interpréteur de ce langage un environnement et une interface qui déterminent les possibilités de celui-ci.

Le langage de script peut alors s'affranchir des contraintes de bas niveau
- Prises en charge par l'intermédiaire de l'interface
- Et bénéficier d'une syntaxe de haut niveau.

Le langage Script Shell est exécuté à partir de fichiers contenant le code source du programme qui sera interprété.

6.1) les commentaires

Commentaire mono-ligne
Le signe # sert à introduire des commentaires.
#ceci est un commentaire

Le signe # ! est un commentaire particulier qui doit être introduit à la première ligne et indique à l'interpréteur le nom du Shell à utiliser pour exécuter le script.

Exemple : # !/bin/bash

Commentaires multi-lignes ou documentation interne
Pour introduire des commentaires multi lignes, il est nécessaire d'utiliser une autre syntaxe, qui commence le commentaire par un double chevron <<NomDuCommentaire

Toutes les lignes qui suivent seront des lignes de commentaires, jusqu'à ce que Bash rencontre le même nom de commentaire fermant ce commentaire.

```
#!/bin/bash
echo "Dire quelque chose"

<<COMMENTAIRE6
    Mon commentaire 1
    Mon commentaire 2
    blah blah
COMMENTAIRE6

echo "Faire autre chose!"
```

6.2) Les variables

Les variables utilisateurs

Une variable n'a pas de type en Shell elle est par défaut du type chaîne de caractères.

L'affectation se fait de la manière suivante :

Variable=[valeur] ou set variable=[valeur] **(Pas d'espaces entre la partie gauche et droite du signe =)**

Il est impératif que le nom de la variable, le symbole = et la valeur à affecter ne forment qu'une seule chaîne de caractères sans espaces.

Il est également possible de déclarer une variable avec le mot clef « **declare** » ou « **typeset** ».

Exemple :

```
declare A="Bonjour"
typeset B="Le monde!"
```

On peut donner à titre indicatif le type entier à une variable en passant en paramètre l'option -i au mot clef « declare ». *-i pour integer*

Exemple :

`declare -i NOMBRE=10` => NOMBRE est ici explicitement déclaré comme un entier.
Ou `typeset –i NOMBRE=10`

Constantes :
Pour définir une variable dont la valeur ne doit pas être modifiée (appelée *constante* dans d'autres langages de programmation), on utilise la syntaxe :
declare –r *nom=valeur*

Ex :

```
declare -r mess=bonjour
```

`-r pour readonly`

Manipulation des variables

La manipulation d'une variable impose l'apposition du signe « $ » devant le nom de la variable « $NomVariable ».

Exemple d'utilisation d'une variable utilisateur

```
A="BONJOUR"
```
`echo $A` => affiche BONJOUR

On a la possibilité d'utiliser une autre forme d'écriture pour l'utilisation des variables, celle ci utilise les caractères accolades comme ceci :

${NomVariable}

Cette forme, permet une plus grand souplesse, car apporte des formes d'écritures d'expansion de la variable en fonction de son contenu (voir le tableau ci dessous).

set variable=valeur ou variable=valeur	Affectation d'une variable
$variable	Lecture de la valeur de la variable
${variable}	Lecture de la valeur de la variable
${#variable}	Donne la longueur de la valeur de la variable « variable »
${*paramètre*:-*mot*}	**Utilisation d'une valeur par défaut**. Si le *paramètre* est inexistant ou nul, on substitue le développement du **mot**. Sinon, c'est la valeur du *paramètre* qui est fournie.
${*paramètre*:=*mot*}	**Attribution d'une valeur par défaut**. Si le *paramètre* est inexistant ou nul, le développement du *mot* lui est affecté. La valeur du *paramètre* est alors renvoyée. Les paramètres positionnels, et spéciaux ne peuvent pas être affectés de cette façon.
${*paramètre*:?*mot*}	**Affichage d'une erreur si inexistant ou nul**. Si le *paramètre* est inexistant, ou nul, le développement du *mot* (ou un message approprié si aucun *mot* n'est fourni) est affiché sur la sortie d'erreur standard, et l'interpréteur se termine, s'il n'est pas interactif. Autrement, la valeur du *paramètre* est utilisée.
${*paramètre*:+*mot*}	**Utilisation d'une valeur différente**. Si le *paramètre* est nul, ou inexistant, rien n'est substitué. Sinon le développement du *mot* est renvoyé.
${paramètre:offset}	Permet d'extraire la sous chaîne qui commence à la position offset, jusqu'à la fin de la chaîne paramètre.
${paramètre:d:l}	Permet d'extraire un bout de chaîne d'une certaine longueur « l » à partir du caractère début « d » dans la chaîne d'origine « paramètre ».

6.3) Les variables locales au Shell courant (en général en majuscule)

Définie par la commande **export en général dans le fichier /etc/.profile ou par le système.**

En voici quelques unes ci dessous :

ARGV	les arguments de la ligne de commande (numérotées à partir de 1)
AUTOLOGOUT	temps de déconnexion automatique d'un Shell (en minutes)

CWD	le chemin du répertoire courant
HOME	le home directory
IGNOREEOF	si elle n'est pas initialisée, on ne peut pas se déconnecter par ^D
CDPATH	chemin de recherche pour les commandes cd, pushd, chdir
NOTIFY	si elle est positionnée, l'utilisateur est immédiatement averti à la terminaison d'un processus en tache de fond. Sinon il est averti lors d'un Shell prompt
PATH	chemin de recherche pour l'exécution des commandes
PROMPT	positionne la valeur du prompt
SHELL	indique le Shell courant
STATUS	retourne le statut de la dernière commande exécutée (0 si elle s'est bien exécutée)
BASH_VERSION	Version du bash Shell
PS1/PS2/PS3/PS4	Prompt" utilisé dans le Shell (on peut modifier leur valeurs)
IFS	Séparateur de paramètres sur la ligne de commande.
OLDPWD	Ancien répertoire de travail appeler par la cmd « cd »
RANDOM	Génère un nombre entre 0 et 32767
LINENO	Numéro de la en cour du script Shell
SECONDS	Nb secondes depuis le début de l'exécution du script
HOSTNAME	Nom de la machine
OSTYPE	Nom de l'os installé…

La variable système IFS (Internal Field Separator) est la variable qui contient la valeur du séparateur de mots reconnus par le Shell, par défaut c'est le caractère espace, cette valeur IFS peut être modifiée dans certaines situations pour indiquer que ce séparateur de mots est un caractère ou un groupe de caractères.
Exemple, pour exploiter des fichiers csv (dont les mots sont séparés par le caractère ;), on définira la variable IFS avec le caractère « ; » avec la commande set IFS=';'
Il faudra sauvegarder sont état d'origine, en effet car il n'est pas simple de la rétablir à sa valeur d'origine.
Par défaut, IFS contient le saut de ligne « \n », mais aussi la tabulation (« \t ») et l'espace.
Ne pas oublier de rétablir la situation à l'origine en fin de script car sinon vos commandes risquent de ne plus fonctionner normalement ensuite.

Exemple :

```
$> /tmp$ OLDIFS=$IFS
$> /tmp$ IFS=$';\n'
$> /tmp$ for i in `cat liste`; do echo "$i"; done
$> IFS=$OLDIFS
```

Dans l'exemple ci dessus, on sauvegarde l'état de la variable IFS dans une autre variable OLDIFS, nous modifions la variable IFS en lui faisant accepter comme séparateurs de mots et de lignes les caractères « ; » et « saut de ligne ».
On réalise notre traitement sur le fichier liste, puis on rétabli la valeur de l'IFS.

Exemple d'affichage d'une variable Shell système :

```
$> echo $HOME
```

Il est possible d'obtenir la liste complète des variables d'environnements en exécutant la commande « env » ou « printenv », comme ceci :

```
$> printenv | more
```

6.4) Les paramètres de la ligne de commande (variables positionnelles)

Récupération des paramètres de la ligne de commande

$#	indique le nombre de composants (séparés par des blancs voir variable IFS) dont est composés la variable
$n	équivaut à $argv[n] mais ne sortira jamais d'erreur out of range n compris entre 0 et 9
${n}	Idem a ci-dessus, mais permet d'utiliser des valeurs > 9 exemple : ${12}
$@** **$*	liste des paramètres avec valeur IFS

Quelques variables particulières, et très utiles

$$	le numéro de processus de la dernière commande
$?	Statut de la dernière commande exécutée
$!	Numéro ID du dernier processus exécuté en tache de fond.

Exemple d'utilisation :

```
# !/bin/sh
echo "  nom du script " $0
echo " nombre d'argument : " $#
echo " liste des paramètres :" $*
ls -l
echo "Numéro de processus de la commande ls : " $$
echo "Valeur de sortie de la commande ls : "  $?
```

Utilisation des paramètres positionnels

Un paramètre positionnel est un paramètre issu de la liste des paramètres de la ligne de commande, ou des paramètres d'une fonction Shell.

Il se note $n ou n peut prendre que la valeur 0 → 9 maximum.

$1, $2, $3 ,$4 ,$5 ,$6 jusque $9 permettent de rappeler les 9 premiers arguments de la ligne de commande.

$0 correspond au nom du script

Au delà il faut utiliser la syntaxe suivante :

Soit : effectuer une commande **shift** [n] qui permet de décaler les numéros de paramètre vers la droite de n valeurs.

MonScript.sh	A	B	C	D	E	F	G	H	I	J	K
$0	$1	$2	$3	$4	$5	$6	$7	$8	$9		

shift

MonScript.sh A B C D E F G H I J K
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9

Shift 2

MonScript.sh A B C D E F G H I J K
$0 $1 $2 $3 $4 $5 $6 $7 $8 $9

Sinon il est possible d'utiliser également la syntaxe ${Num}
echo ${10} =>J
echo ${11} =>K

Initialisation des paramètres positionnels avec la commande « set »

La commande set appelée sans option, mais suivie d'arguments affecte ces derniers aux paramètres positionnels ($1 , $2, …,$#,$@,$*) cela permet de manipuler facilement le résultat de substitutions diverses.

Exemple:

```
set `date`
echo $1 => Tue
echo $2=> Mar
echo $4=> 23 :57 :33
echo $# => 6
echo $@ => Tue Mar 23:57:33 MET 2010
```

6.5) Gestion des tableaux en Shell

La gestion des tableaux est très simple et très intuitive.
Les tableaux en Shell sont toujours et uniquement « uni dimensionnel ».

La déclaration d'un tableau doit s'effectuer avec le mot clef « **declare** » grâce à l'option **-a** (pour array), comme ceci :
nb : Un tableau en Shell commence à l'indice 0 (base zéro).

```
declare -a tableau
```

Affectation de valeur aux indices d'un tableau :

```
tableau[0]="premier membre"        # affectation d'une variable
tableau[1]="2eme membre"       # la variable tableau se transforme en tableau
```

Il n'est pas obligatoire d'utiliser des indices contiguës

```
tableau[12]="3ieme élément"
```

Afficher un élément d'un tableau selon sont numéro d'indice :
echo ${tableau[0]} # affiche l'élément 0

Récupérer la longueur du contenue d'une cellule du tableau :
echo ${#tableau[0]} # affiche la longueur de l'élément 0

Afficher tous les éléments d'un tableau :
echo ${tableau[*]} #affiche tous les éléments

Récupérer la taille d'un tableau (nb éléments) :
echo ${#tableau[*]} # affiche le nombre d'élément

6.6) Les caractères spéciaux

\	banalise le caractère suivant (utile pour afficher des caractères comme $ dans une chaîne) echo "la valeur \$A=$A"
"..."	banalise les caractères sauf \ , $ et `
'...'	banalise tous les caractères
`...`	substitution de commande (altgr+7), on peut utiliser la commande $(…) à la place.
%num	préfixe pour les numéros de job (le numéro de job est indiqué lors du lancement d'une commande en background) Pour visualiser la liste des jobs : **jobs**

6.7) Les instructions de contrôles et d'itérations

La structure if

if [condition]
then
[commande]
else
[commande]
fi
==========
if [condition]
then
[commande]
elif [condition]
then
[commande]
else
[commande]
fi

Le *new line* est équivalent au " ; "

```
if [ $USER = Robert ]
then
    echo "Bonjour Robert."
else
    echo "Bonjour, $USER."
    echo "Vous n'êtes pas autorisé a utiliser ce script."
fi
```

La boucle while

La boucle `while` exécute les *commandes* de manière répétée tant que la première *commande* réussit ; en anglais, *while* signifie « tant que ».

La boucle `while` adopte la syntaxe suivante :

while [condition]
do
[commandes]
done

Après l'instruction `do`, vous pouvez mettre autant de commandes que vous le désirez, suivies de retours chariot ou bien de points virgules (« ; »). Le Shell exécutera tout ce qu'il trouve, jusqu'à ce qu'il tombe sur l'instruction `done`.

Exemple :

```
while [ $nb -ge 1 ]
do
nb=$(($nb-1))
done
```

La boucle until

Until signifie « jusqu'à ce que » en anglais. Cette boucle est le pendant de la boucle `while`, à ceci près que **la** *condition* **ne détermine pas la cause de la répétition de la boucle, mais celle de son interruption**.

until [condition]
do
[commandes]
done

Exemple :

```
echo "Tapez le mot de passe :"
```

```
read password

until [ $password = "" ]
do echo "Mot de passe incorrect."
    echo "Tapez le mot de passe"
    read password
done

echo "Mot de passe correct."
echo "Bienvenue sur les ordinateurs secrets de la NASA."
```

La boucle for

La boucle `for` affecte successivement à une variable chaque chaîne de caractères trouvée dans une *liste de chaînes*, et exécute les *commandes* une fois pour chaque chaîne.

for *var* **in** [liste de chaînes]
do
[commandes]
done

Exemple :

```
for var in 1 2 3 4
do
echo $var
done
```

```
% prog
1
2
3
4
%
```

for name in [liste]
do
[commandes]
done

La liste des mots suivants qui suivent la commande "in" est développée, produisant une liste d'articles. Le nom de la variable est affecté sur chaque élément de cette liste, et ceci est exécutée à chaque fois. Si la commande "in" est omise, la commande for exécute le développement pour chacun des paramètres positionnels qui est non vide (affecté). Le code de retour est l'état de sortie de la dernière commande exécutée. Si l'expansion des éléments suivants la commande "in"dans est une liste vide, aucune commande n'est exécutée, et le code de retour est 0.

```
for num in `seq 1 10`
do
echo $num
done
```

for ((expr1 ; expr2 ; expr3))
do
[commandes]
done

Tout d'abord, l'expression arithmétique "expr1" est évaluée selon les règles. L'expression arithmétique "expr2" est évaluée à plusieurs reprises jusqu'à ce qu'elle soit évaluée à zéro. Chaque fois que "expr2" est évaluée à une valeur différente de zéro, la liste est exécutée et l'expression arithmétique "expr3" est évaluée. Si une expression est omise, elle se comporte comme si elle est était évaluée à 1. La valeur de retour est le code de sortie de la dernière commande de la liste qui est exécuté, ou false si aucune des expressions n'est pas valide.

```
for (( I=0; I<=10; I++ ))
do
     for (( J=0; J<=10; J++ ))
     do
          echo $I*$J = $(($I*$J))
     done
     echo " ----------- "
done
```

La structure « case »

Quand préférer `case` à `if` ?

La structure de contrôle `if` est très pratique, mais devient rapidement illisible lorsqu'un aiguillage offre plusieurs sorties, et que l'on doit répéter une condition plusieurs fois sous des formes à peine différentes. Typiquement, un programme comme celui-ci est pénible à écrire, à lire et à débugger :

La structure `case` adopte la syntaxe suivante :

case *chaîne* **in**
  ***motif*)** *commandes* **;;**
  ***motif*)** *commandes* **;;**
esac

La structure `case` commence par évaluer la *chaîne*. Ensuite, elle va lire chaque *motif*. Enfin, et dès qu'une *chaîne* correspond au *motif*, elle exécute les *commandes* appropriées. En anglais, *case* signifie « cas » ; cette structure de contrôle permet en effet de réagir adéquatement aux différents « cas de figure » possibles.

Vous observerez que :

- le *motif* est séparé des commandes par une parenthèse fermante (qui n'est ouverte nulle part !) ;
- la série des *commandes* est close par deux points et virgules successifs (« ;; ») ;
- la structure `case` est close par l'instruction `esac`, qui n'est autre que le mot `case` à l'envers (de même que la structure `if` est terminée par l'instruction `fi`).

Un *motif* (*pattern*) est un mot contenant éventuellement les constructions `*`, `?`, `[a-d]`, avec la même signification que pour les raccourcis dans les noms de fichiers (les *jokers*). Exemple :

```
case $var in
  [0-9]*) echo "$var est un nombre.";;
  [a-zA-Z]*) echo "$var est un mot.";;
  *) echo "$var n'est ni un nombre ni un mot.";;
esac
```

La structure “select”

select identificateur **[in** liste_choix**]**
do
        [commande]
        ...
done

La commande **select** écrit sur la sortie d'erreur standard la liste des choix, chacun étant précédé d'un numéro. Si **in liste_choix** n'est pas spécifié, les paramètres positionnels sont utilisés.
Le contenu de la variable **PS3** s'affiche et l'entrée standard est lu. Si le numéro d'un des mots listés est saisi, le paramètre i**dentificateur** prend la valeur de ce mot.

Si la ligne est vide, la liste s'affiche de nouveau. Sinon, la valeur du paramètre Identificateur est "". Le contenu de la ligne lue à partir de l'entrée standard est sauvegardé dans le Paramètre **REPLY**. La liste est exécutée pour chaque sélection jusqu'à un caractère d'interruption ou de fin de fichier.

```
PS3=" votre choix "
select menu_list in "choix 1"  "choix 2"  "fin"
do
      case $menu_list in
            "choix 1")echo "choix 1";;
            "choix 2")echo "choix 2";;
            "fin") exit 0 ;;
            "") echo "$REPLY est une option invalide "
      esac
done
```

La structure goto <étiquette>

Syntaxe

```
boucle :
        [liste_commandes]
        goto boucle
```

6.8) Les Tests

La commande « **test** » [ce n'est pas une commande interne du Shell].
Syntaxe : **test** *expression*

Exemple :

```
if test $1 -eq qcq
 then
[commandes]
fi
```

Sous Bash , au lieu de la commande test, on préférera utiliser les caractères [ ].

Exemple :

```
if [ $1 -eq qcq ]
 then
[commandes]
fi
```

a) Test sur les fichiers

-a file Vrai si le fichier existe.
-b file Vrai si le fichier existe et, est un fichier bloc.
-c file Vrai si le fichier existe et, est un fichier de type caractères.
-d file Vrai si le fichier existe et, est un répertoire.
-e file Vrai si le fichier existe.
-f file Vrai si le fichier existe et, est un fichier régulier
-g file Vrai si le ficher existe et set-group-id est positionné.
-h file Vrai si le fichier existe et est un lien symbolique.
-k file Vrai si le fichier existe et le ``sticky'' bit est positionné.
-p file Vrai si le fichier existe et c'est un pipe nommé.
-r file Vrai si le fichier existe et est lisible.
-s file Vrai si le fichier existe et à une taille supérieur à zéro.
-t fd Vrai si le descripteur de fichier est ouvert et se réfère à un terminal.
-u file Vrai si le fichier existe et le set-user-id bit est positionné.
-w file Vrai si le fichier existe et les en écriture.
-x file Vrai si le fichier existe et est exécutable.
-O file Vrai si le fichier existe et est possédé par le user id.
-G file Vrai si le fichier existe et est possédé par le group id.

-L file Vrai si le fichier existe et est un lien symbolique.
-S file Vrai si le fichier existe et est une socket.
-N file Vrai si le fichier existe et à été modifié depuis sa dernière lecture.
file1 -nt file2 Vrai si file1 est plus récent (par rapport a la date de modification) que file2, ou si file1 existe et file2 n'existe pas.
file1 -ot file2 Vrai si file1 est plus vieux que file2, ou si file2 existe et file1 n'existe pas.
file1 -ef file2 Vrai si file1 et file2 se référent au même numéro « *d'inode* ».

Exemple de test sur les fichiers :

```
if [ -r $2 ]
then
...
...
else
echo "$0 :vous n'avez pas le droit de lire le fichier $2">&2
fi
```

b) Test sur les chaînes de caractères

- test chaîne (ou [chaîne]) : vraie si chaîne est une chaîne vide
- -z chaîne : vraie si chaîne est une chaîne vide
- -w chaîne : vraie si chaîne est une chaîne non-vide

- chaine1 = chaine2 : vraie si chaine1 est égale a chaine2

- chaine1 != chaine2 : vraie si chaine1 n'est pas égale à chaine2

c) Tests binaires sur les nombres

- n1 -eq n2 : vraie si n1 est égal a n2
- n1 -ne n2 : vraie si n1 est diffèrent de n2
- n1 -gt n2 : vraie si n1 est plus grand strictement a n2
- n1 -ge n2 : vraie si n1 est plus grand ou égal à n2
- n1 -lt n2 : vraie si n1 est plus petit strictement a n2
- n1 -le n2 : vraie si n1 est plus petit ou égal à n2

d) Test logiques

- !n1 non (not)

- n1 -a n2 Test logique booléan AND

- n1 -o n2 Test logique booléan OR

6.9) Sortie de texte sur la sortie standard

echo "Bonjour\n" => Certaine version du Shell accepte l'option –e permettant de prendre en compte les caractères spéciaux suivant :

\a alert (bell) \b backspace \c suppress trailing newline \e an escape character \f form feed \n new line \r carriage return \t horizontal tab \v vertical tab \\ backslash \0nnn the eight-bit character whose value is the octal value nnn (zero to three octal digits) \xHH the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)

Exemple :

```
echo –e "bonjour /n"
```

6.10) Écrire en couleur sur la console :

L'option -e permet également d'utiliser les séquences d'échappement ANSI pour les terminaux qui acceptent l'affichage couleur.
IL est donc possible en utilisant les bonne séquence d'échappement d'écrire en couleur sur la console à partir d'un script Shell.

La capture d'écran ci dessous affiche la liste des valeurs à utiliser pour l'arrière et l'avant plan.

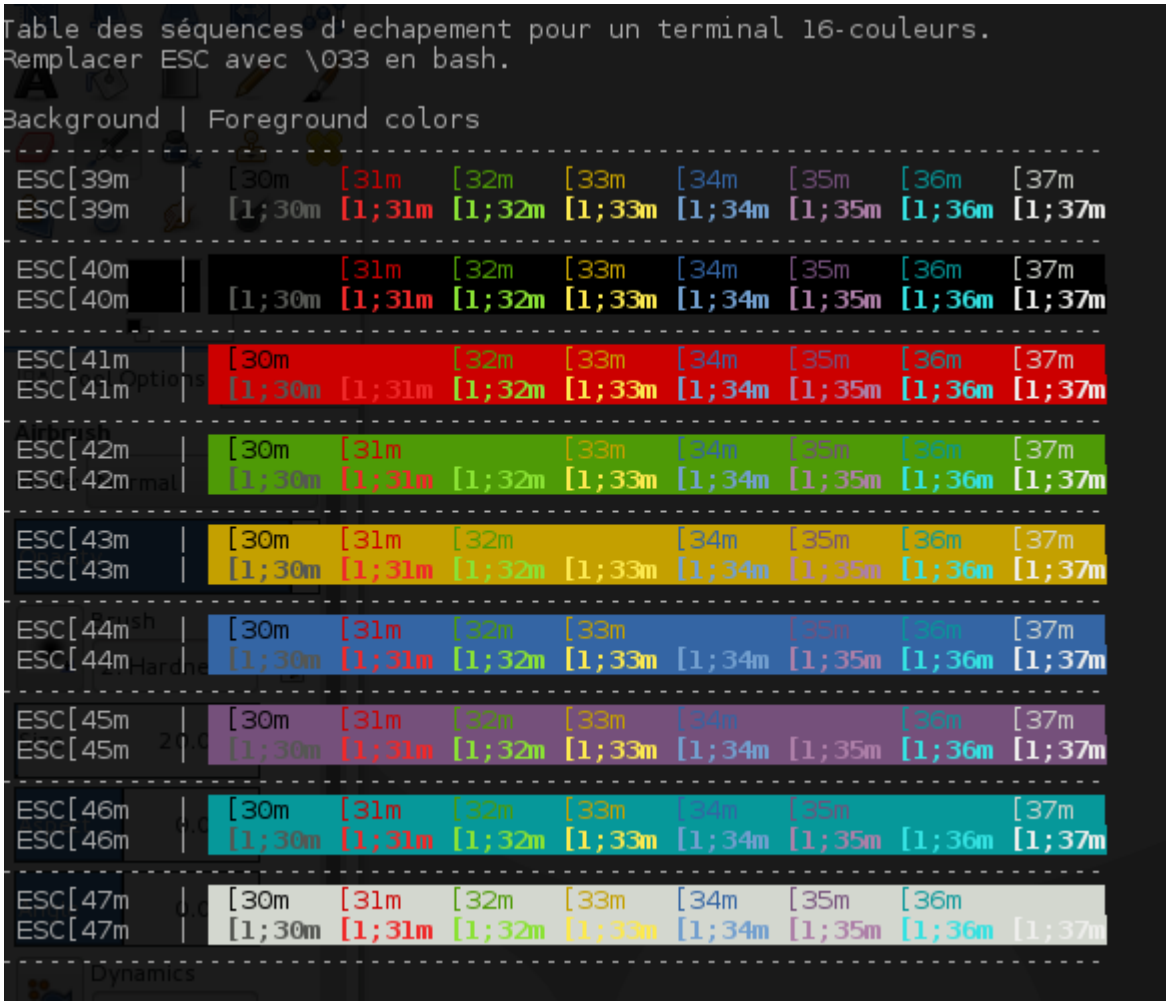
Petit script qui permet d'afficher sur votre console la capture ci dessous :

```
#!/bin/bash

echo
echo Table des séquences d\'échappement pour un terminal 16-
couleurs.
echo Remplacer ESC avec \\033 en bash.
echo
echo "Background | Foreground colors"
echo
"------------------------------------------------------------
-------"
for((bg=39;bg<=47;bg++)); do
     for((bold=0;bold<=1;bold++)) do
          echo -en "\033[0m"" ESC[${bg}m   | "
          for((fg=30;fg<=37;fg++)); do
```

```
                if [ $bold == "0" ]; then
                     echo -en "\033[${bg}m\033[${fg}m [${fg}m  "
                else
                     echo -en "\033[${bg}m\033[1;${fg}m [1;$
{fg}m"
                fi
           done
           echo -e "\033[0m"
     done
     echo
"-----------------------------------------------------------------
------- "
done

echo
echo
```



On définira pour cela une couleur de fond et une couleur d'avant plan, ainsi qu'une surbrillance si nécessaire , exemple :

Couleur de fond noire et d'avant plan rouge en surbrillance :

```
echo -e  "\033[40m\033[1;31m Bonjour!\n"
```

On remarquera que pour utiliser la séquence échappement on écrit \033[BGm et \ 033[BOLD ;FGm

ou BG peut prendre les valeurs de 39 => 47
et FG peut prendre les valeurs de 30 => 37
et BOLD 0 ou 1

6.11) La fonction printf

Il est possible d'utiliser la fonction 'printf' qui est équivalent au printf du langage C.

printf chaîne expr1 expr2 ..... exprn

chaîne représente la chaîne qui sera affichée à l'écran.
Elle peut contenir des formats qui seront substitués par la valeur des expressions citées à sa suite.
Il doit y avoir autant de formats que d'expressions.

Exemple de formats utilisés.

%20s	Affichage d'une chaine (string) sur 20 positions avec cadrage à droite
%-20s	Affichage d'une chaine (string) sur 20 positions avec cadrage à gauche
%3d	Affichage d'un entier (décimal) sur 3 positions avec cadrage à droite
%03d	Affichage d'un entier (décimal) sur 3 positions avec cadrage à droite et complété avec des 0 à gauche
%-3d	Affichage d'un entier (décimal) sur 3 positions avec cadrage à gauche
%+3d	Affichage d'un entier (décimal) sur 3 positions avec cadrage à droite et affichage systématique du signe (un nombre négatif est toujours affiché avec son signe)
%10.2f	Affichage d'un nombre flottant sur 10 positions dont 2 décimales
%+010.2f	Affichage d'un nombre flottant sur 10 positions dont 2 décimales, complété par des 0 à gauche, avec cadrage à droite et affichage systématique du signe

Exemples d'utilisations :

printf « texte »

printf « %s \n » « bonjour »

printf « %s %d \t %s » « Salut vous » 15 « Personnes »

Les caractères d'échappements suivant peuvent être utilisés…

\a alert (bell) \b backspace \c suppress trailing newline \e an escape character \f form feed \n new line \r carriage return \t horizontal tab \v vertical tab \\ backslash \0nnn the eight-bit character whose value is the octal value nnn (zero to three octal digits) \xHH the eight-bit character whose value is the hexadecimal value HH (one or two hex digits)

6.12) Les fonctions en Shell

Les fonctions dans le Shell sont très simple, elles ne prennent pas de paramètres.

Exemple de fonction :

```
Nom_fonction(){
# variable locale non accessible en dehors
locale variable=100 de la fonction
echo $1
echo $2
echo $3
echo $variable
return 0
}
```

La valeur retournée par une fonction correspond a son code d'erreur et non à la valeur de retour de la fonction.

Notez les **parenthèses ouvrante et fermante** : ce sont elles qui indiquent à l'interpréteur du script qu'il s'agit d'une définition de fonction.

Comme l'interpréteur de scripts Shell lit des scripts ligne à ligne, **il faut que la fonction soit définie avant d'être appelée**. Sinon, vous recevez un message de type : « Command not found » (commande introuvable).
Par convention, il est préférable de **placer** *toutes* **les définitions de fonction au début du programme**, avant toutes les instructions d'appels.

L'appel se fait avec le nom de la fonction et les différents paramètres !!!…
En effet ces paramètres seront passés dans les paramètres positionnels de la fonction.

```
Nom_fonction param1 param2 param3 param4
```

Il est possible de déclarer une variable locale dans une fonction :

```
Locale texte_exemple = " Variable Locale "
```

Utilisation des paramètres positionnels dans une fonction.

Dans une fonction, comme dans un script Shell, la position des paramètres est repéré par les paramètres $n ou n est compris entre 0 et 9 ou en utilisant la syntaxe ${n}.

De même les paramètres $# et $@ ou $* on le même rôle ici dans une fonction.

NB : a la sortie de la fonction les paramètres positionnels sont de nouveau initialisé aux valeurs passé au script.

NB : Les fonctions Shell sont récursives.

6.13) Quelques commandes à connaître

break

(permet de quitter une boucle for, while ou until)

continue ou continue [n]

(Force une boucle while, for, until à reprendre à l'itération suivante).
Si on indique une valeur n on peut sauter n itérations.

eval

Permet d'évaluer des arguments

Exemple

```
Foo=10
X=Foo
eval Y='$'$X
echo $Y  → Sortie 10
```

exit [n]

Envoie un numéro de sortie et termine le script en cours

expr

(Évalue ses arguments comme une expression mathématique)

X=` expr $X+1` fonctionne avec les opérations de base + - \ * % (modulo)

expr1 | expr2 renvoie expr1 si <>0 sinon expr2
expr1 & expr2 renvoie 0 si une des 2 expressions est nulle sinon expr1

On peut utiliser également les opérateurs de comparaisons
=, >, >=, <, <=, !=

Il est possible aussi d'utiliser la forme suivante pour évaluer une expression arithmétique

$((…..))

Permet comme pour expr de réaliser des calculs arithmétiques…

Exemple :
```
x=$(($x+1))
```

let

La commande let réalise des opérations arithmétiques sur les variables. Dans beaucoup de cas, elle fonctionne comme une version simplifiée de expr.

```
let a=11        # Identique à 'a=11'
let a=a+5       # Équivalent à  a=$(($a+5))
```

return n

Fin d'une fonction (n correspond à la valeur du retour de la fonction)

shift [n] n=1 par défaut

Permet d'effectuer un décalage des paramètres de la ligne de commande.
Quand n est défini par une valeur >1 alors on effectue le nombre de décalage indiqué vers la droite.

unset var

Retirer ou supprimer de la mémoire une variable qui a été définie.

```
A=10
unset A
```

L'exécution d'une commande ou d'une fonction externe se fait de la manière suivante :

export nom_variable

Exporte une variable d'environnement vers ses processus enfants.

$(commande) ou `commande` (substitution de commandes)

Il est possible de récupérer le résultat de la fonction de la manière suivante :
```
A=$(ls -l)
```

Une commande *cmd* entourée par une paire de parenthèses **()** précédées d'un caractère **$** est exécutée par le Shell puis la chaîne **$(** *cmd* **)** est remplacée par les résultats de la commande *cmd* écrits sur la sortie standard, c'est à dire l'écran. Ces résultats peuvent alors être affectés à une variable ou bien servir à initialiser des paramètres de position.
Ex :

```
A=`ls -l`
```

Toute commande doit être affectée à une variable avoir de pouvoir utiliser le résultat.

Exemple :

for i in $(ls $1) #ne fonctionne pas.

a=$(ls $1)
for i in $a //est par contre fonctionnel

La commande {liste_de_commandes}

Cette commande se nomme « groupe de commandes », elle prendre entre les accolades des commandes UNIX ou Linux séparés par un « ; », ces commandes sont directements éxécuté dans l'environnement du Shell courant.

Exempled'utilisation :

```
{ ls -l ;
cd .. ;
pwd ;
cd ~ ;}
```

La commande « . »

. /script permet d'exécuter le script dans le Shell courant.

exec

Remplace le Shell courant par un autre programme.

```
exec ls -l
```

```
exec 3<>un_fichier
```

La commande « : »

Est une commande nulle elle est toujours vrai

while :

(Boucle infini car toujours vrai)
Utile pour faire des boucles de menu

```
while :
do
case reponse in
   a) echo " bonjour " ;;
   b) echo " le monde ";;
esac
done
```

La commande read

read [option] varx ...

read variable1 variable2 ...variablen
Le premier mot lu sera associé à variable1
Le second mot lu sera associé à variable2
...
Le n-1éme mot lu sera associé à variable (n-1)
...et tous les mots lus après le n-1eme seront associés à la variable n.

Lorsque la commande interne **read** est utilisée sans argument, la ligne lue est enregistrée dans la variable prédéfinie du Shell **REPLY** .
Ex :

```
$> read
$>  bonjour tout le monde
     $>
     $> echo $REPLY
     $> bonjour tout le monde
     $>
```

L'option **–p** de **read** affiche une chaîne d'appel avant d'effectuer la lecture ; la syntaxe à utiliser est : **read –p** *chaîne_d_appel* **[** *var* **…]**
Ex :

```
$> read -p "Entrez votre prénom : " prenom
$> Entrez votre prénom : Eric
     $>
     $> echo $prenom
     Eric
     $>
```

L'option **–d** de **read** permet de changer le délimiteur par défaut qui sépare les valeurs dans une saisie
Sa syntaxe est : **read –d '**caractère' [var…]

L'option **–n** de **read** permet de limiter le nombre de caractères saisies par l'utilisateur.
Syntaxe: **read –n nb** [var…]

L'option **–t** de **read** permet de limiter l'attente du temps de saisie par l'utilisateur.
Syntaxe : **read –t timeout** [var…]

pwd

Affiche le répertoire courant.

times

Affiche le temps accumulé par l'utilisateur et le "system" pour l'exécution du script Shell. La valeur de retour est 0.

type nom_prog

Affiche la commande détaillée liée au nom de la commande.

Exemple de commande interne au Shell

```
$> type echo
```

echo is a Shell builtin

Exemple de commande externe au Shell

```
$ type awk
awk is /usr/bin/awk
```

trap

- trap argument n_1 ...n_i :si les signaux n_1 ...n_i arrivent alors argument sera exécuté.
  Exemple :
  $>trap " INT

  # INT correspond à Ctrl-C
  Après cette commande ,Ctrl-C n'aura plus aucune action.
- trap n_1...n_i :remet les dispositions par défaut.
  Exemple :
  $>trap " INT
- trap : affiche les dispositions associées aux signaux

umask

- umask xyz : permet de définir le masque pour tous les fichiers nouvellement créés. A l'origine les nouveaux fichiers sont dotés des droits : -rw-rw-rw- et les nouveaux répertoires des droits : -rwxrwxrwx. La valeur octale xyz associée à umask est soustraite de la valeur existante.
  Exemple :
  $>umask 022
  Après cette commande tous les fichiers nouvellement créés auront les droits suivants : -rw-r--r--et les nouveaux répertoires les droits : -rwxr-xr-x.
- umask : permet de connaître le masque actif.
- Pour modifier les droits des fichiers existants il faut utiliser la commande chmod.

6.14) Les Listes d'évaluations ET et OU

la liste ET

Permet l'exécution d'une liste de commandes, chacune n'étant exécuté que si les précédentes l'ont été de façon satisfaisante

Inst1 && inst2 && inst3 …..

Exemple

```
if [ -f fichier_un ] && echo « bonjour » && [-f fichier_deux ]
&& echo « le monde »
then
….
fi
```

Hypothèse 1 : fichier_un existe et fichier_deux existe alors le résultat de l'affichage est :

Bonjour le monde

Hypothèse 2 : fichier_un existe mais pas fichier_deux alors le résultat de l'affichage est :

Bonjour

Hypothèse 3 : fichier_un n'existe pas mais fichier_deux existe alors le résultat de l'affichage est :

« Aucun affichage »

Hypothèse 4 : fichier_un et fichier_deux n'existe alors le résultat de l'affichage est :

« Aucun affichage »

la liste OU

Permet l'exécution d'une liste de commandes jusqu'au succès de l'une d'elles, après quoi aucune instruction n'est exécuté

Inst1 || inst2 || inst3 …..

Exemple

```
if [ -f fichier_un ] || [-f fichier_deux ]  echo « bonjour » ||
echo « le monde »
then
….
fi
```

Hypothèse 1 : fichier_un existe et fichier_deux existe alors le résultat de l'affichage est :

« aucun affichage »

Hypothèse 2 : fichier_un existe mais pas fichier_deux alors le résultat de l'affichage est :

« aucun affichage »

Hypothèse 3 : fichier_un n'existe pas mais fichier_deux existe alors le résultat de l'affichage est :

« aucun affichage »

Hypothèse 4 : fichier_un et fichier_deux n'existe alors le résultat de l'affichage est :

Bonjour