# 3. UTILISATION DE LINUX

## 3.1. INTRODUCTION

Linux est un système d'exploitation puissant mais son utilisation n'est pas facile pour les débutants non familiarisés avec l'environnement UNIX. L'utilisation de la plupart des applications peut s'effectuer à partir de l'interface graphique X−Window (ou à partir de sur−couches de X−Window telles que les environnements graphiques KDE et GNOME). Cependant pour certains travaux, il est beaucoup plus pratique et plus souple d'utiliser des lignes de commande depuis un environnement shell plutôt que d'utiliser de lourdes solutions graphiques. De plus, si vous devez intervenir sur votre serveur Linux à distance (c'est à dire depuis un poste connecté à Internet), vous allez inévitablement devoir utiliser des lignes de commande.

Qu'appelle t'on un shell ? Un shell est la liaison la plus élémentaire entre l'utilisateur et le système d'exploitation, c'est à dire le programme de gestion de la ligne de commande. Les commandes saisies sont interprétées par le shell et transmises au système d'exploitation.

De nombreuses commandes du shell ressemblent aux commandes MS−DOS : en utilisant la terminologie UNIX, nous pouvons considérer que le programme command.com correspond au shell de MS−DOS. Dans les environnements de type UNIX, il existe plusieurs shells (bash, tcsh, csh, sh, etc..)

## 3.2. LES COMMANDES DE BASE

Pour toutes les commandes, il est possible d'obtenir de l'aide en tapant `man` suivi du nom de la commande. En tapant une commande suivie du paramètre `−−help`, nous obtenons la liste des paramètres possibles. N'hésitez pas à recourir à la commande `man` ou au paramètre `−−help` dès que vous avez besoin d'aide.

### 3.2.1. Afficher le contenu d'un fichier (cat et more)

La commande cat permet de visualiser le contenu d'un fichier c'est à dire d'envoyer le contenu du fichier vers une la sortie par défaut : l'écran.

Exercice : Repérer un fichier non exécutable et afficher son contenu (vous pouvez vous rendre dans le répertoire /etc).

La commande `more` permet également de visualiser le contenu d'un fichier. L'affichage s'effectue page par page.

Exercice : Afficher le fichier précédent avec la commande more.

La commande `more` permet également de passer en mode éditeur en tapant `vi` pendant la visualisation du fichier.

### 3.2.2. Editer un fichier (vi et emacs)

L'éditeur le plus redouté des informaticiens est `vi`. Cet éditeur est l'éditeur élémentaire que l'on retrouve sur la plupart des systèmes d'exploitation et qui n'utilise pas d'interface graphique. Il prend en charge les commandes et les données en même temps. Une fois `vi` lancé, deux modes de fonctionnement se présentent : le mode commandes et le mode édition.

- pour passer du mode édition au mode commande il suffit d'appuyer sur la touche échappement ;
- pour passer du mode commande au mode édition il faut taper la commande d'insertion (ou équivalent)

Une fois lancé : `vi <nom de fichier>` ; vous pouvez employer quelques unes des commandes ci−après les plus courantes :
**commandes avec passage en mode insertion**

- `i`, `a` : insère une lettre avant ou après le curseur
- `cw` : modifie le mot courant
- `cc` : modifie la ligne courante

**commandes sans passage en mode insertion**

- `x` , `X` : efface le caractère suivant ou avant les curseur
- `dw` : efface un mot
- `dd` : efface une ligne
- `yw` : copie un mot dans le buffer
- `yy` : copie la ligne courante dans le buffer
- `p` : copie le buffer à la position courante
- `/<mot>` : recherche d'instances de 'mot' dans le fichier ; n pour suivante, N pour précedente
- `:<nombre>` : va a la ligne
- `:q` : quitte le fichier
- `:q!` : quitte le fichier sans sauvegarder
- `:w` : sauvegarde le fichier
- `:x` : sauvegarde et quitte

`emacs` est un autre éditeur standard utilisé dans différents systèmes d'exploitation, il dispose d'un langage qui permet de le personnaliser à souhaits. Il dispose néanmoins de menus 'habituels' tels que la gestion des fichiers, la recherche de caractères, etc.. Ainsi que de règles préprogrammées qui permettent aux développeurs une mise en page dépendante du langage utilisé (C, C++, java ...) reconnaissant les commandes courantes, les chaînes de caractères, etc ..

Exercice : Ajouter les noms des serveurs du réseau local dans le fichier /etc/hosts

### 3.2.3. Retrouver un fichier (find et which)

Comme l'on s'en doute bien il arrive que l'on ait à retrouver un fichier dont on ne connaît plus l'emplacement ou même le nom ; Linux comprend quelques outils pour ces recherches.

La commande which permet de scruter les répertoires les plus communément utilisés (dont le chemin est indiqué dans la variable d'environnement `PATH`) pour retrouver le nom de fichier indiqué en argument.

Exemple : `which apachectl` recherchera dans tous les répertoires (du PATH) le fichier apachectl

Cette commande est surtout utile pour vérifier que l'on utilise bien la version souhaitée d'un binaire (exécutable). La commande `whereis` est semblable à la commande `which`.

La commande `find` :

Syntaxe : `find <répertoire> <conditions>` Arguments : le répertoire du début de recherche et les conditions sur des attributs du fichier.

Exemple : `find /home/philippe −name \*.txt −size +100k` recherchera dans le sous le répertoire de l'utilisateur Philippe tous les fichiers qui finissent par ".txt" et qui pèsent de plus de 100 kilo−octets.

Le signe `|` appelé pipe (ou tube) permet de relier avantageusement les commandes. L'introduction du pipe (tube) permet de combiner plusieurs commandes parmi lesquelles la commande `grep`.

On peut ainsi combiner la commande `find` avec la commande `grep` afin de retrouver une chaîne de caractère dans un fichier.

Exemple : `find . −name LISEZMOI | xargs grep −n LINUX` recherchera à partir du répertoire courant (noté .) les fichiers de noms LISEZMOI et affichera les lignes numérotées (option −n) contenant la chaîne de caractères Linux.

**3.2.4. Trouver du texte dans un fichier (grep)**

Syntaxe : `grep [options] <chaîne de caractères> <nom de fichier>`

Exemple : `grep −i −n pacifique *.txt` affichera toutes les lignes numérotées (option −n) contenant le mot pacifique sans prendre en compte les majuscules (option −i) dans les fichiers finissant par .txt

**3.2.5. Les liens (ln)**

La création de liens symboliques (opposition aux liens physiques) évite la copie de fichiers identiques dans différents répertoires. Par exemple, si une application a besoin d'un fichier volumineux contenant des données relatives à un groupe d'utilisateurs, il est possible de l'avoir virtuellement dans les répertoires courant en créant un lien symbolique : `ln −s <source> <destination>`

Il n'est pas nécessaire que la source existe pour cette création au même titre que sa destruction n'altérera pas le lien mais son appel génèrera un erreur de type fichier introuvable.

**3.2.6. Connaître l'espace disque restant (df, du)**

Pour contrôler l'espace occupé et l'espace d'un disque dur (en fait d'une partition), il existe deux commandes très utiles.

La commande `df` renseigne sur l'espace disque total, disponible (disk free). Elle s'utilise sur tous répertoires "montés".

Cette commande s'utilise généralement avec en argument le nom d'un fichier pour vérifier le point de montage de son répertoire.

Exemple : `df ~/essai` nous indiquera la partition sur laquelle est sauvegardé le répertoire essai (le ~ représente /home/).

La commande `du` calcule l'espace occupé (disk usage) pour un répertoire (sous entendu le répertoire et ses sous−repértoires).

L'option `−k` permet un affichage en kilo−octets.

Exemple : `du −k −s essai` affichera la liste des sous−répertoires du répertoire essai récursivement sans indiquer tous les fichiers et leur taille (option −s).

**3.2.7. Se déplacer dans les répertoires (cd)**

Lorsque vous ouvrez une session Unix avec votre login et votre mot de passe, vous vous retrouvez devant le "prompt" du shell. En fonction du shell employé, le prompt peut avoir la forme suivante :

```
[philippe@sirius essai] $
```

Le mot philippe signifie que vous vous êtes "logué" sur le compte de l'utilisateur philippe, @sirius signifie que vous êtes sur la machine qui porte le nom sirius et essai signifie que vous êtes dans le répertoire essai.

La commande `cd` permet de se déplacer dans les répertoires. La command `ls` permet d'afficher la liste des fichiers d'un répertoire.

Attention, sous les systèmes Unix, un répertoire est désigné par le symbole `/` ou slash et non pas par un \ ou anti−slash comme c'est le cas sous DOS.

Exemples d'utilisation de la commande `cd` :

| | |
|---|---|
| `$ cd /` | déplacement à la racine du système |
| `$ cd /essai` | déplacement dans le répertoire essai de la racine |
| `$ cd essai` | déplacement dans le répertoire courant essai |
| `$ cd essai/` | déplacement dans le répertoire courant essai |
| `$ cd /usr/local` | déplacement dans le répertoire apache du répertoire /usr |
| `$ cd ..` | recule d'une branche vers la racine |
| `$ cd ~` | déplacement dans son répertoire personnel |
| `$ cd ~philippe` | déplacement dans le répertoire personnel de l'utilisateur de philipppe |

Etant donné que le système mémorise le répertoire courant (répertoire dans lequel on est), on peut utiliser des noms de chemins relatifs :

| | |
|---|---|
| `$ cd /home/philippe/essai` | chemin absolu |
| `$ cd essai` | chemin relatif |

**3.2.8. Redirections**

On notera pour l'occasion, la possibilité de redirection très utile sous linux. Cette technique permet de rediriger la sortie d'une commande ou d'un programme ailleurs que vers l'écran, c'est à dire dans un fichier ou vers un autre programme. Ainsi, nous pouvons envoyer un fichier dans l'entrée d'une commande, mettre l'affichage d'une commande dans un fichier et même envoyer l'affichage d'une commande dans l'entrée d'une autre. Enfin, dans notre cas, nous pouvons envoyer un fichier dans un autre.

Avec le symbole **>** (signe supérieur), nous pouvons rediriger la sortie d'un programme vers un fichier.

Exemple : `cal > fevrier` Ici, nous envoyons l'affichage de la commande cal − le calendrier pour le mois en cours − dans le fichier nommé fevrier.

Exemple : `cat > essai1.txt` permet de passer dans 'l'editeur' cat que (l'on quitte avec ctrl+d )

Avec le symbole **<** (signe inférieur), nous redirigerons le contenu d'un fichier vers l'entrée d'une commande.

Exemple : `mail marc < courrier` envoie par courrier électronique à marc le fichier nommé courrier.

**>>** (supérieur supérieur) : ajoute à la fin Ce symbole permet d'ajouter l'affichage d'une commande à la fin d'un fichier, sans pour autant écraser ce qu'il y avait déjà dans le fichier. Avec un seul supérieur, le contenu du fichier

serait remplacé par la sortie de la commande et donc ce contenu aurait été perdu.

Exemple : `cat fichier1 fichier2 fichier3 >> fichier_complet`

Nous ajoutons à la fin (concaténons) du fichier fichier_complet le contenu des fichiers fichier1, fichier2 et fichier3.

Rappel : | (symbole barre), réalise un tube entre deux commandes.

Ce symbole permet de faire en sorte que l'affichage d'une commande soit dirigé dans l'entrée d'une autre commande.

**3.2.9. Modification des droits d'accès**

La commande `chmod` permet de changer les droits d'accès d'un fichier.

Vous ne pouvez modifier les droits d'un fichier que si vous en êtes le propriétaire. Il existe une exception : l'administrateur système root peut modifier les droits d'accès de tous les fichiers.

Syntaxe : `chmod <modification des droits d'accès> <nom du fichier>`

La partie est elle−même composée de 3 parties :

- les droits d'accès à modifier : `u` (user) pour le propriétaire, `g` (group) pour le groupe, `o` (others) pour les autres et `a` (all) pour tout le monde ;
- les types de modification à effectuer : − pour rajouter de nouveaux droits et + pour supprimer des droits d'accès déjà mis en place ;
- la modification individuelle des droits : `r`,`w` ou `x`

Exemples :

| | |
|---|---|
| `$ chmod a+r lisezmoi.txt` | Autorise à tout le monde la lecture de lisezmoi.txt |
| `$ chmod ug+rw,o+r lisezmoi.txt` | Le propriétaire et les membres de son groupe peuvent lire et modifier le fichier lisezmoi.txt alors que les autres personnes ne peuvent que le lire. |
| `$ chmod a+x mon−script` | Permet de spécifier que le fichier mon−script peut être exécuté |

La commande `chown` permet de changer le propriétaire d'un fichier. Syntaxe : `chown utilisateur fichier`

La commande `chgrp` permet de changer le groupe d'un fichier (il faut que le propriétaire appartienne au groupe). Syntaxe : `chgrp groupe fichier`

**3.2.10. Lister les fichiers d'un répertoire (ls)**

La commande `ls` permet de lister le contenu d'un répertoire.

```
[philippe@sirius essai] $ cd /bin
[philippe@sirius /bin] $ ls
arch        dd          gzip        nisdomainname su
ash         df          hostname    ping          sync
```

```
awk         dmesg       kill        ps          tar
cp          fgrep       mount       sh          ypdomain
cpio        gawk        mt          sleep       zcat
csh         grep        mv          sort        zsh
date        gunzip      netstat     stty        ls
```

La commande `ls` sans arguments donne un listing brut difficile à exploiter. Pour obtenir des informations plus précises, il est nécessaire d'utiliser l'argument `−l`.

Exercice : Taper la commande ls avec −l en argument.

Avant de continuer, il est nécessaire de fournir quelques explications sur la gestion des fichiers. Sous Linux, un fichier peut représenter :

- un fichier texte ou un fichier exécutable (on parle alors de fichier binaire) ;
- un répertoire ;
- un périphérique ;
- une référence à un autre fichier (on parle alors de lien).

Linux étant un système multi−utilisateur, les utilisateurs doivent par conséquent être administré. Pour faciliter cette administration, les utilisateurs sont réunis en groupes. Ce qui permet de paramétrer des droits spécifiques à chaque groupe : droits en lecture, mais aussi en écriture et en exécution.

Revenons à l'exemple ci−dessus : les informations fournies sont relativement nombreuses et sont regroupées en colonnes :

- la première colonne fournit des informations sur les droits ;
- la colonne suivante est le nombre de liens pointant sur ce fichier ;
- la suivante donne le nom du propriétaire du fichier.
- la quatrième indique le nom du groupe qui peut accéder au fichier selon les actions autorisées par le propriétaire ;
- la cinquième nous donne la taille du fichier en octets.
- la sixième donne la date et l'heure de la dernière modification du fichier.
- et enfin vient le nom du fichier. S'il s'agit d'un lien, la référence du fichier est indiqué par un −> [source]

La première colonne constitué de 10 caractères fournit des informations sur le type de fichier et les droits associés. Pour le premier caractère :

- le caractère `−` représente un simple fichier ;
- la lettre `d` représente un dossier (directory) ;
- la lettre `l` représente un lien ;

Il existe d'autres types de fichiers mais nous ne nous en occuperons pas à ce niveau. Les 9 lettres suivantes sont groupées trois par trois et indiquent les droits associés au fichier, c'est à dire par qui et comment un fichier peut−être utilisé :

- le premier triplet correspond aux droits du propriétaire,
- le second triplet correspond aux droits des membres du groupe (un utilisateur intègre un groupe de travail afin de partager des fichiers) ;
- le troisième correspond aux droits des autres utilisateurs du système.

Pour chaque triplet :

- la première lettre indique si le fichier peut être lu ou pas : `r` (readable) si le fichier peut être lu et un tiret sinon ;
- la seconde lettre indique si le fichier peut être écrit ou pas : `w` (writable) ou un tiret ;
- la troisième lettre indique si le fichier peut être exécuté : `x` pour un binaire exécutable ou un tiret.

Ainsi dans l'exemple précédent, le fichier lisezmoi.txt peut être lu et écrit pas son propriétaire, il peut également être lu par les membres de son groupe mais ne peut être modifié. Pour les autres, la lecture et la modification de ce fichier ne sont pas autorisés.

Pour vous aider à retenir l'ordre de présentation des droits (utilisateur / groupe / autres), vous pouvez utiliser l'astuce mnémotechnique suivante : je, nous, ils.

Notons qu'il existe également des fichiers cachés sous Linux : lorsque le nom d'un fichier commence par un point (caractère .), celui−ci n'est visible qu'avec l'option `−a`.

Exercice : Taper ls avec −la en argument depuis votre répertoire personnel.

Les liens ainsi que la modification des droits associés à un fichier sont abordés un peu plus loin.

### 3.2.11. Retrouver dans quel répertoire je suis (pwd) et créer un répertoire (mkdir)

Lorsque l'on se déplace dans un répertoire, le shell n'affiche que le nom du répertoire dans lequel on se trouve sans préciser le chemin complet. On peut donc très facilement se tromper de répertoire : par exemple penser être dans le répertoire `/bin` alors que l'on se trouve dans le répertoire `/usr/local/bin`. La commande `pwd` permet de connaître le chemin du répertoire dans lequel on se trouve.

```
[philippe@sirius bin]$ pwd
/usr/local/bin
```

Pour créer un répertoire il suffit d'utiliser la commande mkdir avec le nom du répertoire souhaité en paramètre.

Exercice : Créer un répertoire essai dans votre répertoire personnel.

### 3.2.12. Copier (cp), supprimer (rm), déplacer et renommer un fichier (mv)

La copie de fichier s'effectue avec la commande `cp` (copy). La syntaxe de la commande `cp` est la suivante : `cp source destination` La source et la destination pouvant être un fichier ou un répertoire.

Exemples :

| | |
|---|---|
| `$ cp lisezmoi.txt essai.txt` | Duplique le fichier lisezmoi.txt en essai.txt |
| `$ cp lisezmoi.txt essai/` | Copie le fichier lisezmoi.txt dans le répertoire essai |
| `$ cp essai/ essai2/` | Copie les fichiers du répertoire essai dans le répertoire essai2 |
| `$ cp −R essai/ essai2/` | Copie tous les fichiers du répertoire essai − y compris les sous−répertoires dans le répertoire essai2 |

La commande `rm` (remove) permet de supprimer un fichier.

```
$ rm lisezmoi.txt
```

Supprime le fichier lisezmoi.txt

L'option `-R` permet de supprimer récursivement tout le contenu d'un répertoire. Attention, évitez au maximum d'utiliser cette option et surtout ne l'utiliser jamais en tant que root.

La commande `rmdir` (remove directory) permet de supprimer un répertoire.

```
$ rmdir essai
```
Supprime le répertoire essai

La commande `mv` permet de renommer un fichier.

```
$ cp lisezmoi.txt essai.txt
```
Duplique le fichier lisezmoi.txt en essai.txt

```
$ mv lisezmoi.txt essai2.txt
```
Renomme le fichier lisezmoi.txt en essai2.txt

### 3.3. ARCHIVER, COMPRESSER ET DÉCOMPRESSER

**Archivage de fichiers** :

Pour archiver des fichiers, on assemble le groupe de fichiers à archiver :

```
tar <destination> <sources>
```

Assemblage des différents fichiers (fichier i) dans monfichier :

```
$ tar -cf monfichier.tar fichier1 fichier2 ... fichiern
```

Pour assembler en récursif (avec les sous-répertoire) des répertoire :

```
$ tar -cf monfichier.tar rep1 rep2 ... repn
```

**Désassemblage** :

```
$ tar -xf <monfichier.tar>
```

**Compression d'un fichier** :

Une commande de compression permettra ensuite de diminuer la taille totale de ces fichiers assemblés : `gzip`

Compresse monfichier et le remplace par le fichier monfichier.gz : `gzip monfichier`

Pour décompresser un fichier archive essayer la commande suivante : `gzip -d fichier.gz`

Ainsi l'on peut assembler et compresser les fichiers à archiver.

Remarque : la commande `tar xvfz` permet de décompresser en même temps que le désassemblage.

### 3.4. ECRIRE DES SCRIPTS

Ecrit par Frédéric Bonnaud.

Vous aurez envie d'écrire un script (petit programme écrit avec un langage simple : shell, perl ou autre) dès que vous aurez tapez dans un terminal quatre fois la même série de commandes et que vous vous apercevrez que vous êtes ammené à le refaire de nombreuses fois. Un script est une suite d'instruction élémentaire qui sont éxécutées de façon séquencielle (les unes après les autres) par le langage de script Dans cet article nous nous limiterons à l'utilisation du shell comme langage, et en particulier à bash. En guise de première introduction, vous pouvez lire ce qui concerne les commandes du shell dans l'article Le Shell et les Commandes. Attention, n'espérez pas que ce document constitue un manuel complet de programmation ! C'est une courte introduction qui nous l'espérons, vous permettra d'écrire de petits scripts qui vous rendrons de précieux services.

**Notions de base**

Pour commancer, il faut savoir qu'un script est un fichier texte standard pouvant être créer par n'importe quel éditeur : vi, emacs ou autre. D'autre part, conventionnellement un script commance par une ligne de commentaire contenant le nom du langage à utiliser pour interpréter ce script, pour le cas qui nous intéresse : /bin/sh. Donc un script élémentaire pourrait être :

```
#!/bin/sh
```

Évidemment un tel script ne fait rien ! Changeons cela. La commande qui affiche quelque chose à l'écran est echo. Donc pour créer le script bonjour_monde nous pouvons écrire :

```
#!/bin/sh
echo "Bonjour, Monde !"
echo "un premier script est né."
```

Comment on l'éxécute ? C'est simple il suffit de faire :

```
[user@becane user]$ sh bonjour_monde
Bonjour, Monde !
un premier script est né.
[user@becane user]$ _
```

C'est pas cool, vous préféreriez taper quelque chose comme :

```
[user@becane user]$ ./bonjour_monde
Bonjour, Monde !
un premier script est né.
[user@becane user]$ _
```

C'est possible si vous avez au préalable rendu votre script éxécutable par la commande :

```
[user@becane user]$ chmod +x bonjour_monde
[user@becane user]$ ./bonjour_monde
Bonjour, Monde !
un premier script est né.
[user@becane user]$ _
```

Résumons : un script shell commence par : `#!/bin/sh`, il contient des commandes du shell et est rendu éxécutable par `chmod +x`.

Quelques conseils concernant les commentaires Dans un shell−script, est considéré comme un commentaire tout ce qui suit le caractère # et ce, jusqu'a la fin de la ligne.
Usez et abusez des commentaires : lorsque vous relirez un script 6 mois après l'avoir écrit, vous serez bien content de l'avoir documenté. Un programme n'est jamais trop documenté. Par contre, il peut être mal documenté ! Un commentaire est bon lorsqu'il décrit pourquoi on fait quelque chose, pas quand il décrit ce que l'on fait.

Exemple :

```
#!/bin/sh
# pour i parcourant tous les fichiers,
for i in * ; do
# copier le fichier vers .bak
   cp $i $i.bak
# fin pour
done
```

Que fait le script ? Les commentaires ne l'expliquent pas ! Ce sont de mauvais commentaire. Par contre :

```
#!/bin/sh
# on veut faire un copie de tous les fichiers
for i in * ; do
# sous le nom $i.bak
  cp $i $i.bak
done
```

Là, au moins, on sait ce qu'il se passe. (Il n'est pas encore important de connaitre les commandes de ces deux fichiers)

**Le passage de paramètres**

Un script ne sera, en général, que d'une utilisation marginale si vous ne pouvez pas modifier son comportement d'une manière ou d'une autre. On obtient cet effet en "passant" un (ou plusieurs) paramètre(s) au script. Voyons comment faire cela. Soit le script essai01 :

```
#!/bin/sh
echo le paramètre \$1 est \"$1\"
echo le paramètre \$2 est \"$2\"
echo le paramètre \$3 est \"$3\"
```

Que fait−il ? Il affiche, les uns après les autres les trois premiers paramètres du script, donc si l'on tappe :

```
$ ./essai01 paramètre un
le paramètre $1 est "paramètre"
le paramètre $2 est "un"
le paramètre $3 est ""
```

Donc, les variables $1, $2 ... $9 contiennent les "mots" numéro 1, 2 ... 9 de la ligne de commande. Attention : par "mot" on entend ensemble de caractères ne contenant pas de caractères de séparations. Les caractères de séparation sont l'espace, la tabulation, le point virgule.

Vous avez sans doute remarqué que j'ai utilisé les caractères : \$ à la place de $ ainsi que \" à la place de " dans le script. Pour quelle raison ? La raison est simple, si l'on tape : echo "essai" on obtient : essai, si l'on veut obtenir "essai" il faut dire à echo que le caractère " n'indique pas le début d'une chaine de caractère (comme c'est le comportement par défaut) mais que ce caractère fait partie de la chaine : on dit que l'on "échappe" le caractère " en

tapant \". En "échappant" le caractère \ (par \\) on obtient le caractère \ sans signification particulière. On peut dire que le caractère \ devant un autre lui fait perdre sa signification particulière s'il en a une, ne fait rien si le caractère qui suit \ n'en a pas.

Maintenant, essayons de taper :

```
$ ./essai01 *
le paramètre $1 est "Mail"
le paramètre $2 est "essai01"
le paramètre $3 est "nsmail"
$ _
```

(Le résultat doit être sensiblement différent sur votre machine). Que c'est−il passé ? Le shell a remplacé le caractère * par la liste de tous les fichiers non cachés présents dans le répertoire actif. En fait, toutes les substitutions du shell sont possible ! C'est le shell qui "substitue" aux paramètres des valeurs étendues par les caractères * (toute suite de caractères) et [ab] (l'un des caractères a ou b). Autre exemple :

```
$ ./essai01 \*
le paramètre $1 est "*"
le paramètre $2 est ""
le paramètre $3 est ""
$ _
```

Et oui, on a "échappé" le caractère * donc il a perdu sa signification particulière : il est redevenu un simple *.

C'est bien, me direz vous, mais si je veux utiliser plus de dix paramètres ? Il faut utiliser la commande shift, à titre d"exmple voici le script essai02 :

```
#!/bin/sh
echo le paramètre 1 est \"$1\"
shift
echo le paramètre 2 est \"$1\"
shift
echo le paramètre 2 est \"$1\"
shift
echo le paramètre 4 est \"$1\"
shift
echo le paramètre 5 est \"$1\"
shift
echo le paramètre 6 est \"$1\"
shift
echo le paramètre 7 est \"$1\"
shift
echo le paramètre 8 est \"$1\"
shift
echo le paramètre 9 est \"$1\"
shift
echo le paramètre 10 est \"$1\"
shift
echo le paramètre 11 est \"$1\" Si vous tapez :
$ ./essai02 1 2 3 4 5 6 7 8 9 10 11 12 13
le paramètre 1 est "1"
le paramètre 2 est "2"
le paramètre 2 est "3"
```

```
le paramètre 4 est "4"
le paramètre 5 est "5"
le paramètre 6 est "6"
le paramètre 7 est "7"
le paramètre 8 est "8"
le paramètre 9 est "9"
le paramètre 10 est "10"
le paramètre 11 est "11"
$ _
```

A chaque appel de shift les paramètres sont déplacés d'un numéro : le paramètre un devient le paramètre deux et c. Évidemment le paramètre un est perdu par l'appel de shift : vous devez donc vous en servir avant d'appeler shift.

**Les variables**

Le passage des paramètres nous à montrer l'utilisation de "nom" particuliers : $1, $2 etc. ce sont les substitutions des variables 1, 2 et c. par leur valeurs. Mais vous pouvez définir et utiliser n'importe quelle nom. Attention toute fois, à ne pas confondre le nom d'une variable (notée par exemple machin) et son contenue (notée dans cas $machin). Vous connaissez la variable PATH (attention le shell différencie les majuscules des minuscules) qui contient la liste des répertoires (séparés par des " :") dans lesquels il doit rechercher les programmes. Si dans un script vous tapez :

```
1:#!/bin/sh
2:PATH=/bin           # PATH contient /bin
3:PATH=PATH:/usr/bin  # PATH contient PATH:/bin
4:PATH=/bin           # PATH contient /bin
5:PATH=$PATH:/usr/bin # PATH contient /bin:/usr/bin
```

(Les numéros ne sont là que pour repérer les lignes, il ne faut pas les taper).

La ligne 3 est très certainement une erreur, à gauche du signe "=" il faut une variable (donc un nom sans $) mais à droite de ce même signe il faut une valeur, et la valeur que l'on a mis est "PATH :/usr/bin" : il n'y a aucune substitution à faire. Par contre la ligne 5 est certainement correcte : à droite du "=" on a mis "$PATH :/usr/bin", la valeur de $PATH étant "/bin", la valeur après substitution par le shell de "$PATH :/usr/bin" est "/bin :/usr/bin". Donc, à la fin de la ligne 5, la valeur de la variable PATH est "/bin :/usr/bin".

Attention : il ne doit y avoir aucun espace de part et d'autre du signe "=".

Résumons : MACHIN est un nom de variable que l'on utilise lorsque l'on a besoin d'un nom de de variable (mais pas de son contenu) et $MACHIN est le contenu de la variable MACHIN que l'on utilise lorsque l'on a besoin du contenu de cette variable. Variables particulières Il y a un certain nombre de variables particulières, voici leur signification :

- la variable * (dont le contenu est $*) contient l'ensemble de tous les "mots" qui on été passé au script.
- la variable # contient le nombre de paramètres qui ont été passés au programme.
- la variable 0 (zéro) contient le nom du script (ou du lien si le script a été appelé depuis un lien).

Il y en a d'autres moins utilisées : allez voir la man page de bash.

**Arithmétique**

Vous vous doutez bien qu'il est possible de faire des calculs avec le shell. En fait, le shell ne "sait" faire que des calculs sur les nombres entiers (ceux qui n'ont pas de virgules ;−). Pour faire un calcul il faut encadrer celui−ci de : $(( un calcul )) ou $[ un calcul ]. Exemple, le script essai03 :

```
#!/bin/sh
echo 2+3*5 = $((2+3*5))
MACHIN=12
echo MACHIN*4 = $[$MACHIN*4] Affichera :
$ sh essai03
2+3*5 = 17
MACHIN*4 = 48
```

Vous remarquerez que le shell respecte les priorités mathématiques habituelles (il fait les multiplications avant les additions !). L'opérateur puissance est "**" (ie : 2 puissance 5 s'écrit : 2**5). On peut utiliser des paranthèses pour modifier l'ordre des calculs.

**Les instructions de controles de scripts**

Les instructions de controles du shell permettent de modifier l'éxécution purement séquencielle d'un script. Jusqu'à maintenant, les scripts que nous avons créé n'étaient pas très complexe. Ils ne pouvaient de toute façon pas l'être car nous ne pouvions pas modifier l'ordre des instructions, ni en répéter.

**L'éxécution conditionnelle**

Lorsque vous programmerez des scripts, vous voudrez que vos scripts fassent une chose si une certaine condition est remplie et autre chose si elle ne l'est pas. La construction de bash qui permet cela est le fameux test : if then else fi. Sa syntaxe est la suivante (ce qui est en italique est optionnel) :

```
if <test> ; then
   <instruction 1>
   <instruction 2>
   ...
   <instruction n>
else
   <instruction n+1>
   ...
   <instruction n+p>
fi
```

Il faut savoir que tout les programmes renvoie une valeur. Cette valeur est stockée dans la variable ? dont la valeur est, rapelons le : $ ?. Pour le shell une valeur nulle est synonyme de VRAI et une valeur non nulle est synonyme de FAUX (ceci parce que, en général les programmes renvoie zéro quand tout c'est bien passé et un numéro non nul d'erreur quand il s'en est produit une).

Il existe deux programmes particulier : false et true. true renvoie toujours 0 et false renvoie toujours 1. Sachant cela, voyons ce que fait le programme suivant :

```
#!/bin/sh
if true ; then
   echo Le premier test est VRAI($?)
else
   echo Le premier test est FAUX($?)
fi

if false ; then
   echo Le second test est VRAI($?)
else
   echo Le second test est FAUX($?)
```

```
fi
```

Affichera :

```
$ ./test
Le premier test est VRAI(0)
Le second test est FAUX(1)
$ _
```

On peut donc conclure que l'instruction if ... then ... else ... fi, fonctionne de la manière suivante : si (if en anglais) le test est VRAI(0) alors (then en anglais) le bloque d'instructions comprises entre le then et le else (ou le fi en l'absence de else) est éxécuté, sinon (else en anglais) le test est FAUX(différent de 0)) et on éxécute le bloque d'instructions comprises entre le else et le fi si ce bloque existe.

Bon, évidemment, des tests de cet ordre ne paraissent pas très utiles. Voyons de vrais tests maintenant.

**Les tests**

Un test, nous l'avons vu, n'est rien de plus qu'une commande standard. Une des commandes standard est 'test', sa syntaxe est un peu complexe, je vais la décrire avec des exemples.

- si l'on veut tester l'existance d'un répertoire , on tapera : test −d
- si l'on veut tester l''existance d'un fichier , on tapera : test −f
- si l'on veut tester l'existande d'un fichier ou répertoire , on tapera : test −e

Pour plus d'information faites : man test.

On peut aussi combiner deux tests par des opérations logiques : ou correspond à −o, et correspond à −a (à nouveau allez voir la man page), exemple : `test −x /bin/sh −a −d /etc` Cette instruction teste l'existance de l'éxécutable /bin/sh (−x /bin/sh) et (−a) la présence d'un répertoire /etc (−d /etc).

On peut remplacer la commande test par [ ] qui est plus lisible, exemple :

```
if [ −x /bin/sh ] ; then
   echo /bin/sh est executable. C\'est bien.
else
   echo /bin/sh n\'est pas executable.
   echo Votre système n\'est pas normal.
fi
```

Mais il n'y a pas que la commande test qui peut être employée. Par exemple, la commande grep renvoie 0 quand la recherche a réussi et 1 quand la recherche a échoué, exemple :

```
if grep −E "^frederic:" /etc/passwd > /dev/null ; then
   echo L\'utilisateur frederic existe.
else
   echo L'utilisateur frederic n\'existe pas.
fi
```

Cette série d'instruction teste la présence de l'utilisateur frederic dans le fichier /etc/passwd. Vous remarquerez que l'on a fait suivre la commande grep d'une redirection vers /dev/null pour que le résutat de cette commande ne soit pas affichée : c'est une utilisation classique. Ceci explique aussi l'expression : "Ils sont tellement intéressant tes mails que je les envoie à /dev/null" ;−).

**Faire quelque chose de différent suivant la valeur d'une variable**

Faire la même chose pour tous les éléments d'une liste Lorsque l'on programme, on est souvant ammené à faire la même pour divers éléments d'une liste. Dans un shell script, il est bien évidemment possible de ne pas réécrire dix fois la même chose. On dira que l'on fait une boucle. L'instruction qui réalise une boucle est

```
for <variable> in <liste de valeurs pour la variable> ; do
   <instruction 1>
   ...
   <instruction n>
done
```

Voyons comment ça fonctionne. Supposons que nous souhaitions renommer tous nos fichiers *.tar.gz en *.tar.gz.old, nous taperons le script suivant :

```
#!/bin/sh
# I prend chacune des valeurs possibles correspondant
# au motif : *.tar.gz
for I in *.tar.gz ; do
   # tous les fichier $I sont renommé $I.old
   echo "$I -> $I.old"
   mv $I $I.old
# on fini notre boucle
done
```

Simple, non ? Un exemple plus complexe ? Supposons que nous voulions parcourir tous les répertoires du répertoire courant pour faire cette même manipulation. Nous pourrons taper :

```
1:#!/bin/sh
2:for REP in `find −type d` ; do
3:    for FICH in $REP/*.tar.gz ; do
4:        if [ −f $FICH ] ; then
5:            mv $FICH $FICH.old
6:        else
7:            echo On ne renomme pas $FICH car ce n\'est pas un
répertoire
8:        fi
9:    done
10:done
```

Explications : dans le premier for, on a précisé comme liste : `find −type d` (attention au sens des apostrophes, sur un clavier azerty français on obtient ce symbole en appyant sur ALTGR+é). Lorsque l'on tape une commande entre apostrophes inverses, le shell éxécute d'abord cette commande, et remplace l'expression entre apostrophe inverse par la sortie standard de cette commande. Donc, dans le cas qui nous intéresse, la liste est le résultat de la commande find −type d, c'est à dire la liste de tous les sous répertoires du répertoire courant. Donc en ligne 2 on fait prendre à la variable REP le nom de chacun des sous répertoires du répertoire courant, puis (en ligne 3) on fait prendre à la variable FICH le nom de chacun des fichiers .tat.gz de $REP (un des sous répertoires), puis si $FICH est un fichier on le renomme, sinon on afficher un avertissement.

Remarque : ce n'est pas le même fonctionnement que la boucle for d'autres langage (le pascal, le C ou le basic par exemple).

**Faire une même chose tant qu'un certaine condition est remplie.**

Pour faire une certaine chose tant qu'une condition est remplie, on utilise un autre type de boucle :

```
while <un test> ; do
   <instruction 1>
   ...
   <instruction n>
done
```

Supposons, par exemple que vous souhaitiez afficher les 100 premiers nombres (pour une obscure raison), alors vous taperez :

```
i=0
while [ i −lt 100 ] ; do
   echo $i
   i=$[$i+1]
done
```

Remarque : −lt signifie "lesser than" ou "plus petit que".

Ici, on va afficher le contenu de i et lui ajouter 1 tant que i sera (−lt) plus petit que 100. Remarquez que 100 ne s'affiche pas.

**Refaire à un autre endroi la même chose**

Souvant, vous voudrez refaire ce que vous venez de tapez autre part dans votre script. Dans ce cas il est inutile de retaper la même chose, préférez utiliser l'instruction function qui permet de réutiliser une portion de script. Voyons un exemple :

```
#!/bin/sh
function addpath ()
{
   if echo $PATH | grep −v $1 >/dev/null; then
       PATH=$PATH:$1;
   fi;
   PATH=`echo $PATH|sed s/::/:/g`
}

addpath /opt/apps/bin
addpath /opt/office52/program
addpath /opt/gnome/bin
```

Au debut, nous avons défini une fonciton nommée addpath dont le but est d'ajouter le premier argument ($1) de la fonction addpath à la varaible PATH si ce premier argument n'est pas déjà présent (grep −v $1) dans la variable PATH, ainsi que supprimer les chemins vide (sed s/ : :/ :/g) de PATH.
Ensuite, nous éxécuton cette fonction pour trois arguments : /opt/apps/bin, /opt/office52/bin et /opt/gnome/bin.

En fait, une fonction est seulement un script écrit à l'intérieur d'un script. Elles permettent surtout de ne pas multiplier les petits scripts, ainsi que de partager des variables sans se préocuper de la clause export mais cela constitue une utilisation avancée du shell, nous ne nous en occuperons pas dans cet article.

Remarque : le mot function peut être omis.

**Autres types de répétitions**

Il existe d'autres type de répétitions, mais nous ne nous en occuperons pas dans cette article, je vous conseille la lecture forcément profitable de la man page de bash.