T.D. SCRIPTS - AdmiSys 2012-2013

Licence Professionnelle ADMISYS

François FIGAROLA
Développeur
Sàrl I-Consult

L'objet de ces Travaux Dirigés est d'initier (pour certains ou approfondir pour les autres) les étudiants de Licence Professionnelle AdmiSys, aux techniques de scripts afin de leur permettre d'envisager un certain degré d'automatisation des taches d'administration.

Les notions de base seront dans un premier temps envisagées dans un environnement Unix-like, mais pourront être extrapolées vers les systèmes Windows™ plus ou moins récents (des traitements batchs originaux de DOS aux touts récents scripts Windows Power Shell™ de Windows 2008 Server™).

Quoi qu'il en soit, la majeure partie de ces T.D. portera sur l'étude des possibilités de scriptage offertes par le shell Unix (et plus précisément le shell **bash**, maintenant shell par défaut de la plupart des distributions Linux) et des principales commandes de base disponibles sur ces systèmes.

Pour conclure cette courte introduction, voici quelques arguments pour se convaincre de l'utilité du *scripting*, et ce même dans le cadre d'une formation d'administrateurs de réseaux, par essence éloignés, voire allergiques à toute forme de programmation :

- Le premier point est tout simplement historique : utiliser la ligne de commande fût longtemps la seule possibilité de communication avec la machine (soit après la carte perforée, et avant les terminaux graphiques). Dès lors, automatiser des tâches journalières pouvait apparaître comme une belle avancée. A l'identique, un serveur Unix-like n'a aucun besoin d'interface graphique pour exécuter nombre de services [1] ; ou sera hébergé à distance dans un "DataCenter", l'accès via une interface graphique devenant de ce fait compliquée... ou du moins très ralentie ! Il faut également bien garder en mémoire que les applications graphiques en environnement Unix (du moins celle bien écrites !) ne servent jamais qu'à générer syntaxiquement une "commande" qui sera finalement exécutée de façon cachée à l'utilisateur...

- Ensuite, bien que quelques tâches (créer un utilisateur, par exemple) puissent être rapidement affectuées via un outil graphique, si l'on doit les réaliser répétitivement un très grand nombre de fois, celà devient vite fastidieux ; et l'on s'aperçoit que les interfaces graphiques ne sont plus adaptées à cette nouvelle problématique.

En conclusion, ne vaut-il mieux pas perdre (une fois pour toutes) un peu de temps à mettre au point un script, que perdre *souvent* du temps à répéter une même action ?

1. Le shell Unix : historique rapide, compatibilités

Avant d'envisager les scripts-shells d'un point de vue purement technique, il convient également d'en cerner l'évolution historique et les *(in)compatibilités* qui en découlent.

1.1. Historique des interpréteurs de commandes :

Au début fût le Bourne Shell (**`sh`**), écrit au sein des laboratoires AT&T, dans les années 1970 par Steve Bourne. Ce programme sert donc à lancer interactivement des commandes, mais il est également déjà doté de possibilités de programmation.

A peu près à cette même période, Bill Joy [2] écrit pour sa part le C-shell (**`csh`**) qui s'avère incompatible avec le Bourne Shell, mais est doté de capacités supplémentaires, telles que l'historisation des commandes, les alias ou encore le contrôle des tâches. Ce shell équipa par défaut les systèmes *BSD, puis fût remplacé par le plus récent Tenex C-Shell (**`tcsh`**) dans les versions ultérieures de ces systèmes d'exploitation, mais qui finiront par adopter malgré tout le shell bash.

Il faut attendre 1983 pour commencer à voir la nouvelle génération d'interpréteurs de commandes ; qui voit le jour sous la paternité de David Korn et prendra donc le nom de Korn Shell (**`ksh`**). Ce nouveau shell est basé sur l'historique Bourne Shell, lui ajoutant nombre des nouvelles fonctionnalités faisant le succès du C-shell, et prenant ainsi une place prépondérante au palmarès des interpréteurs de commande pour Unix ; à tel point qu'il en deviendra le standard de fait.

En 1988, David Korn en sort une nouvelle révision, le ksh 88 qui servira de base à la normalisation du shell (norme IEEE Posix 1003.0). En 1993, une dernière version est publiée, le ksh 93, c'est un sur-ensemble de la norme POSIX, permettant notamment de manipuler des nombres flottants et des tableau associatifs.

Au milieu des années 1990, la Free Software Foundation propose quant à elle sa propre version du Bourne Shell, dénommée *Bourne Again Shell* (**`bash`**), qui est conforme au standard édicté par la norme POSIX, tout en offrant quelques extensions. Le shell bash est actuellement l'interpréteur fourni en standard par les distributions Linux : c'est donc particulièrement sur celui-ci que portera notre étude.

1.2. Compatibilité entre les différents shells

Il découle de ce court historique que les scripts initialement écrits pour le C-Shell et ceux pour le Bourne Shell seront par essence incompatibles entre eux, impliquant un portage plus ou moins important selon les fonctionnalités utilisées.

Par contre, les scripts écrits pour ksh (du moins ceux n'utilisant pas les fonctionnalités introduites dans la version de 1993) seront à priori compatibles avec le shell bash, et dans une moindre mesure le bourne

shell originel ; seules quelques fonctions avancées ne seront pas reconnues (par exemple la commande **`printf`** de **bash**.

2. Les Fondements : descripteurs de fichiers par défaut, redirections et pipes

L'un des principes fondamentaux d'Unix prévoit de découper les traitements complexes en un certain nombre de commandes simples qui seront coordonnées ou juxtaposées, voire imbriquées, afin de fournir un résultat prévisible et constant pour un contexte donné. [3] :

Pour ce faire, Unix a été conceptuellement doté (et ce au niveau de la librairie C sous-jacente) de mécanismes permettant la collaboration des différentes commandes, unifiées par l'environnement controlant l'exécution de ce flux (généralement un shell), qui vont ainsi transporter d'une commande à l'autre les données textuelles traitées : *l'information est transportée sous forme d'un flux textuel*

Les descripteurs automatiquement créés pour chaque nouveau processus.
Les redirections.
Les tubes (ou pipes).

2.1. Les descripteurs de fichiers par défaut :

Lors de la création d'un nouveau processus, trois descripteurs de fichiers sont automatiquement créés. Pour s'en assurer, il suffira de compiler et exécuter le petit fichier source C suivant :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char **argv)
{
 int i, status;
 pid_t pid, fils;
 char c;
 char  fic[256];
 struct stat fs;

 pid = getpid();
 printf("PID du processus : %d \n",pid);
 for (i = getdtablesize(); i >= 0; --i)
 {
  if ( ! fstat(i, &fs) )
  {
   printf("- Descripteur %d (inode %d%d - ", i, fs.st_dev, fs.st_ino);
```

```
   memset(&fic, 0, sizeof(fic));
   sprintf((char *)&fic, "/proc/%d/fd/%d", pid, i);
   switch (fs.st_mode & S_IFMT)
   {
    case S_IFBLK:  printf("périphérique bloc) :");  break;
    case S_IFCHR:  printf("périphérique caractère) :"); break;
    case S_IFDIR:  printf("répertoire) :");   break;
    case S_IFIFO:  printf("FIFO/tube) :");   break;
    case S_IFLNK:  printf("lien symbolique) :");  break;
    case S_IFREG:  printf("fichier ordinaire) :");  break;
    case S_IFSOCK: printf("socket) :");   break;
    default:       printf("type inconnu...) :");
   }
   printf("\n");
   fils = fork();
   if (fils == 0)
   {
    execl("/bin/ls", "/bin/ls", "-l", fic, NULL);
    _exit (EXIT_FAILURE);
   }
   else if (fils < 0)
    printf("\nErreur lors de l'exécution de la commande ls.\n");
    else
     if (waitpid (fils, &status, 0) != fils)
      status = -1;

    printf("\n");
  }
 }
 printf("Appuyez sur une touche pour terminer l'exécution du programme.");
 c = tolower (fgetc (stdin));
 return 0;
}
```

Compilez le code via une commande du style :

```
td_scripts$ gcc -o descripteurs descripteurs.c
```

Puis exécutez le programme ainsi obtenu :

```
td_scripts$ ./descripteurs


PID du processus : 5085
- Descripteur 2 (inode 110 - périphérique caractère) :
lrwx------ 1 francois francois 64 sep 21 17:24 /proc/5085/fd/2 -> /dev/pts/1

- Descripteur 1 (inode 110 - périphérique caractère) :
```

```
lrwx------ 1 francois francois 64 sep 21 17:24 /proc/5085/fd/1 -> /dev/pts/1

- Descripteur 0 (inode 110 - périphérique caractère) :
lrwx------ 1 francois francois 64 sep 21 17:24 /proc/5085/fd/0 -> /dev/pts/1

Appuyez sur une touche pour terminer l'exécution du programme.
```

Lors de son exécution, ce petit programme parcourt sa table des descripteurs à l'envers, et affiche ceux qui ont été affectés. Pour chacun des descripteurs ouverts (via la fonction *open(...)* de la libc), il procède à l'affichage du fichier correspondant par un appel à la commande unix **`ls`**. Ce qui nous montre bien, que par défaut, un processus lancé depuis un shell dispose de 3 descripteurs de fichiers vers la console d'appel (dans l'exemple, il s'agit du pseudo-terminal */dev/pts/1*) : ces descripteurs sont hérités du shell père qui les a créés lors de son initialisation (login).

En conclusion, par défaut tout nouveau processus dispose de 3 descripteurs de fichiers ouverts sur la même console :

- descripteur *0 : l'entrée standard (stdin en C)*, ouvert en mode lecture, permet d'alimenter le processus en données via le clavier.
- descripteur *1 : la sortie standard (stdout en C)*, ouvert en mode écriture, permet d'afficher du texte à l'écran.
- descripteur *2 : la sortie d'erreur (stderror en C)*, ouvert en mode écriture, permet d'afficher les messages d'erreur rencontrées durant l'exécution à l'écran.

Le fait d'avoir 2 descripteurs distincts ouverts en écriture permettra de traiter distinctement les résultats et les messages d'erreur, même si par défaut ces différents flux utilisent le même terminal.

2.2. Les redirections :

La redirection est le mécanisme offert par le shell permettant d'affecter un flux de données vers un fichier particulier[4] C'est ainsi qu'il sera possible de récupérer le résultat d'une commande dans un fichier, ou encore d'envoyer un fichier vers une commande.

2.2.1. La redirection des sorties (en écriture)

Elle permet donc d'envoyer les affichages normalement affectés à un descripteur donné vers un fichier; et peut prendre deux formes :

La *simple redirection*, qui s'écrit via le symbole : **`>`**, et crée ou écrase le fichier de destination (s'il existe).
La *double redirection*, qui s'écrit via le symbole : **`>>`**, et crée ou concatène le flux sortant vers le fichier de destination.

La syntaxe d'une redirection de sortie est donc la suivante :

`macommande [descripteur]>[>] nomfichier` où les termes entre crochets sont optionnels.

Exemples :

`td_scripts$` **`ls > toto.txt`** envoie le résultat de la commande ls dans le fichier toto.txt.

Qu'il est également possible d'écrire sous la forme :

`td_scripts$` **`ls 1> toto.txt`** dans la mesure où à défaut d'indication, c'est le descripteur 1 qui est utilisé par la redirection de sortie.

`td_scripts$` **`ls > toto.txt; ls -l >> toto.txt`** envoie le résultat de la commande ls dans le fichier toto.txt, puis y ajoute le résultat de la commande ls -l.

`td_scripts$` **`ls > toto.txt; ls -l > toto.txt`** envoie le résultat de la commande ls dans le fichier toto.txt, puis l'écrase par le résultat de la commande ls -l.

`td_scripts$` **`ls -l monfichierquinexistepas > toto.txt`** écrase le fichier toto.txt en n'y écrivant rien dedans (puisque le fichier visé n'existe pas), et affiche le message d'erreur suivant à l'écran :

```
ls: ne peut accéder monfichier: Aucun fichier ou répertoire de ce type
```

`td_scripts$` **`ls -l monfichierquinexistepas > toto.txt 2>/dev/null`** écrase le fichier toto.txt en n'y écrivant rien dedans (puisque le fichier visé n'existe toujours pas), et redirige le message d'erreur vers /dev/null, ce qui en supprime l'affichage.

`td_scripts$` **`ls -l monfichierquinexistepas * > toto.txt 2>toto.txt`** envoie le résultat de la commande ls et le message d'erreur dans le fichier toto.txt.

`td_scripts$` **`ls -l monfichierquinexistepas * > toto.txt 2>&1`** est identique à la précédente forme, si ce n'est que dans ce cas, la syntaxe **`2>&1`** indique au shell de rediriger les flux destinés au descripteur 2 vers le descripteur 1.

2.2.2. La redirection de l'entrée standard (en lecture)

Elle permet donc d'affecter le contenu d'un fichier en tant que saisie d'une commande; et peut également prendre deux formes :

La *simple redirection*, qui s'écrit via le symbole : **`<`**, et lit des données depuis le fichier spécifié.
La *double redirection*, qui s'écrit via le symbole : **`<<`**, et lit les données présentes entre deux occurences d'une étiquette.

> **Astuce :** Cette forme est essentiellement utilisée dans les script pour utiliser une portion de celui-ci en tant qu'entrée d'une commande ou encore pour transmettre une transcription littérale d'une saisie.

Sa syntaxe est la suivante:

`macommande [0]<[<][-][ETIQUETTE] nomfichier` où les termes entre crochets sont optionnel.

Exemples :

`td_scripts$` **`mail -s "un sujet" lepape@vatican.com < toto.txt`** envoie le contenu du fichier toto.txt dans le corps du message électronique à destination du pape.

`td_scripts$` **`mail -s "un sujet" lepape@vatican.com <MSG`**

```
>bla
>bla
>bla
>MSG
```

est équivalente à la précédente, si ce n'est que la commande mail attend cette fois-ci la saisie du message depuis la console. Le saisie se termine par la saisie en début de ligne de l'étiquette, ici le mot MSG (C'est d'ailleurs le comportement normal de cette commande si rien ne lui est spécifiée - rappelez vous la ligne se terminant par . pour finir un message !).

Soit le bout de script suivant dans toto.sh :

```
cat <<FIN
 Ceci est message
 sur plusieurs lignes...
FIN

cat <<-EOF
 Ceci est un message
 sur plusieurs lignes...
 dans lequel les tabulations seront omises
 EOF
```

Son exécution par `td_scripts$` **`. toto.sh`** affiche le contenu :

```
 Ceci est message
 sur plusieurs lignes...
Ceci est un message
sur plusieurs lignes...
dans lequel les tabulations seront omises
```

> **Astuce :** L'utilisation du symbole - suivant la redirection << permet de supprimer les éventuelles tabulations ayant servi à indenter proprement le code du script.

2.3. Les tubes de communication (ou pipes) :

Un tube est un autre mécanisme du shell permettant de faire communiquer entre eux deux processus, sa syntaxe est la suivante :

```
macommande1 | macommande2 [| macommande3] [...]
```

Le tube a donc pour effet d'affecter la sortie standard de la commande située à gauche de l'expression à l'entrée standard de la commande de droite de l'expression ;

> **Note :** Le tube n'a de sens que si les commandes utilisées à droite de l'expression utilisent leur entrée standard, ce qui est notamment le cas pour les commandes ls, echo, cp, rm, mv, etc...

Exemples :

`td_scripts$` **`ls | tee toto.txt`** envoie le résultat de la commande ls vers la commande tee laquelle affiche sur la console ce qu'elle reçoit sur son entrée standard et l'insère également dans le fichier toto.txt.

`td_scripts$` **`ls | wc -l`** wc -l compte le nombre de lignes reçues sur son entrée standard (en l'occurence la sortie de ls).

`td_scripts$` **`dpkg -L coreutils | grep bin | pr -4 | sed -e '/^$/d' | sed -e '/^[1-9]/d'`** affiche sur 4 colonnes les binaires contenus dans le paquet debian coreutils

dont voici le résultat :

```
/usr/sbin     /usr/bin/head  /usr/bin/sort    /bin/pwd
/usr/sbin/chroot  /usr/bin/tsort  /usr/bin/du   /bin/echo
/usr/bin     /usr/bin/chcon   /usr/bin/paste   /bin/ls
/usr/bin/tail    /usr/bin/expand   /usr/bin/cksum   /bin/df
/usr/bin/cut   /usr/bin/tac    /usr/bin/factor   /bin/mv
/usr/bin/users  /usr/bin/dirname  /usr/bin/pr    /bin/chown
```

```
/usr/bin/printenv  /usr/bin/csplit   /usr/bin/pathchk  /bin/cp
/usr/bin/sha384su  /usr/bin/printf   /usr/bin/link    /bin/dd
/usr/bin/sha1sum  /usr/bin/who    /usr/bin/ptx    /bin/readlink
/usr/bin/groups   /usr/bin/fold    /usr/bin/sha256su /bin/chgrp
/usr/bin/mkfifo   /usr/bin/dircolor  /usr/bin/sha224su  /bin/false
/usr/bin/[     /usr/bin/tee    /usr/bin/base64   /bin/rm
/usr/bin/md5sum /usr/bin/tty    /usr/bin/pinky   /bin/true
/usr/bin/whoami   /usr/bin/logname  /usr/bin/stat    /bin/date
/usr/bin/runcon   /usr/bin/od    /usr/bin/install   /bin/sleep
/usr/bin/yes    /usr/bin/nl    /usr/bin/expr   /bin/mkdir
/usr/bin/shuf    /usr/bin/env    /usr/bin/uniq   /bin/rmdir
/usr/bin/seq    /usr/bin/join    /usr/bin/wc   /bin/mknod
/usr/bin/tr    /usr/bin/sha512su  /usr/bin/basename  /bin/sync
/usr/bin/nohup   /usr/bin/sum    /bin/vdir    /bin
/usr/bin/unexpand  /usr/bin/shred   /bin/ln     /bin/dir
/usr/bin/id    /usr/bin/fmt    /bin/uname   /bin/chmod
/usr/bin/hostid   /usr/bin/comm   /bin/stty     /usr/bin/md5sum.t
/usr/bin/nice    /usr/bin/split   /bin/cat     /usr/bin/touch
/usr/bin/unlink   /usr/bin/test    /bin/touch
```

Note : C'est avec l'utilisation des tubes de communication que le notion de *filtres Unix* prend tout son sens (cf. chapite 5) !

2.4. Le regroupement de commandes :

Il peut être parfois utile de regrouper un certain nombre de commandes, qui seront alors séparées par un caractère point-virgule (`;`), soit pour en rediriger les sorties vers un même fichier ou vers un tube, soit pour les faire exécuter dans un même environnement.

2.4.1. Regroupement par des parenthèses :

Ce regroupement permet d'agréger les sorties standard (et les sorties d'erreurs) de chaque commande, comme s'il s'agissait d'un seul et unqiue descripteur. Dans ce cas, les commandes sont exécutées classiquement, dans l'ordre d'apparition dans l'expression, par des shells enfants.

Exemple :

`td_scripts$` **`(echo "DEBUT DU FICHIER" ; cat toto.txt ; echo "FIN DU FICHIER) > titi.txt`** envoie le contenu du fichier toto.txt encadré des lignes "DEBUT..." et "FIN..." dans le fichier titi.txt.

2.4.2. Regroupement par des accolades :

Cet autre type de regroupement permet de faire exécuter la séquence de commandes par le shell courant, et également d'agréger les sorties.

> **Note :** Dans ce cas toutes les commandes (y compris la dernière) doivent être suivies d'un point-virgule (`;`), de plus un espace est nécessaire après l'accolade ouvrante et avant l'accolade fermante.

Exemple :

`td_scripts$` **`{ cd monnouveaurepertoire ; echo "DEBUT DU FICHIER" ; cat toto.txt ; echo "FIN DU FICHIER ; } > titi.txt`** envoie le contenu du fichier toto.txt encadré des lignes "DEBUT..." et "FIN..." dans le fichier titi.txt, en outre l'environnement du shell sera modifié, le répertoire courant étant maintenant ./monnouveaurepertoire/ .

3. Notions de base : le shell, les processus, l'environnement

En guise d'hasardeuse métaphore, il serait possible de comparer le shell à un chef d'orchestre règlant l'ensemble des musiciens (les commandes) pour jouer une partition commune... à ceci près que le chef d'un orchestre ne joue pas réellement (si ce n'est de la baguette) alors que le shell, non seulement commande la symphonie Unix, mais y participe activement via ses commandes internes.

3.1. Les commandes internes / externes

Un certain nombre de commandes acceptées par le shell sont immédiatement exécutées au sein de son propre processus : ce sont les *commandes internes* du shell. Ces commandes ont général pour but de modifier l'environnement courant (ex. cd, pwd, export, etc...) ou encore sont suffisamment simples pour être implémentées au sein du shell (ex. type).

A contrario, *une commande externe* est implémentée par un fichier exécutable présent au sein du système de fichiers. (ex. ls, cp, type, etc...)

> **Note :** A l'instar de la commande pwd, certaines commandes sont implémentées à la fois en tant que commande interne ou externe : comment savoir laquelle va être exécutée ?

> Le principe est simple : la priorité est donnée à l'implémentation interne (et ce pour la simple raison de performances) ! Si vous tenez absolument à invoquer la commande externe, il faudra alors l'invoquer par son chemin d'accès complet (ex. /bin/pwd).

La différence en termes de performances entre les deux types de commandes est notable. En effet, comme il a été dit plus haut, une commande interne est directement exécutée par le shell courant, dans son propre processus ; alors qu'une commande externe sera systématiquement exécutée par un shell fils du shell courant, créé via un appel système fork(...) qui aura dû dupliquer l'environnement du processus père et recopié l'ensemble de ses descripteurs.

3.2. L'environnement utilisateur

Lorsqu'un shell est lancé, un certain nombre de variables sont initialisées, soit de façon automatique par le shell lui-même (valeurs par défaut), soit par des mécanismes automatiques et paramétrables dans des fichiers texte (fichiers de profil ou de ressources) : c'est l'ensemble des valeurs de ces variables qui est appelé *environnement*.

Ces variables sont simplement déclarée en utilisant la commande interne **`declare`** (qui peut également être omise) ; leur destruction pouvant alors s'opérer grace à la commande **`unset`**.

Exemple :

`td_scripts$` **`declare TOTO='un texte'`** est équivalent à `td_scripts$` **`TOTO='un texte'`** pour initialiser la variable TOTO avec la valeur (une chaine de caractères) 'un texte'.

L'accès à la valeur d'une variable particulière, s'obtient simplement en préfixant à son nom du caractère **`$`**.

Exemple :

`td_scripts$` **`echo $PATH`** affiche la valeur de la variable PATH (les chemins de recherche des exécutables).

Pour afficher l'ensemble des variables de l'environnement courant, le shell dispose d'une commande interne : **`set`**

Il est également possible via la commande **`env`** de modifier l'environnement temporairement, afin d'exécuter une commande. Sans aucun paramètre ni argument (ou alors simplement -i), la commande **`env [-i]`** affiche un environnement initial, c'est à dire ne comportant que les variables exportées par le shel courant.

L'environnement est donc controlé via un certain nombre de fichier de paramétrage, certains sont utilisés par l'ensemble du système, d'autres sont particuliers à chaque utilisateur. Il s'agit de scripts-shell destinés à être lus par le shell lors de son démarrage ; et il faut de plus différencier deux types de comportements selon que le shell qui se lance soit un *shell de connexion* (c'est à dire le shell lancé immadiatement après l'identification de l'utilisateur), soit un *shell de session* (shell lancé interactivement en cours de session).

3.2.1. Fichiers utilisés par le shell de connexion :

Le premier fichier lu est un fichier système nommé **`/etc/profile`** qui contient des paramétrages communs à tous les utilisateurs du système. Ensuite, le shell va chercher dans le répertoire de l'utilisateur le fichier **`.profile`** (du moins les shells sh et ksh) ; ou, dans l'ordre, les fichiers **`.bash_profile`**, **`.bash_login`** et **`.profile`** (shell bash).

Ces fichiers permettent donc d'affiner, globalement et pour chacun des utilisateurs, le contenu des variables d'environnement (prompt, chemin d"exécution, etc...) ou les paramètres système (umask, définition du terminal, etc...).

> **Note :** Ne pas définir dans ces fichiers des options du shell ou des alias : ces entités ne se transmettent par définition pas entre processus, et donc seront purement et simplement ignorées par les shells enfants !

3.2.2. Fichiers utilisés par un shell de session :

Les shells ksh et bash (à la différence de sh) voient leur comportement modifié en cas de lancement post-session (ils prennent aussi alors la dénomination de "shells interactifs") : outre l'héritage des variables exportées du shell de session, ils vont lire un fichier supplémentaire, nommé, pour le shell bash **`~/.bashrc`** ; et dépendant de la variable ENV pour le shell ksh (celle-ci est généralement initialisée à la valeur **`.kshrc`**).

> **Note :** Ces fichiers de configuration ne sont lus par le shell que lors de son démarrage. Aussi, en cas de modification en cours de session, pour en tester le résultat, il est inutile de singer l'utilisateur type de Microsoft Windows™ (i.e. redémarrer le système) : la commande interne `.` (ex. **`. ~/.bashrc`**) est la solution !

3.2.3. Transmission de l'environnement entre processus :

Lorsqu'une commande est lancée, celle -ci est généralement exécutée dans un shell (processus) fils via un appel système *fork(...)* lequel va dupliquer les descripteurs ouverts et l'environnement du processus

père : en fait d'environnement, seules les variables *exportées* sont réellement transmises, les autres étant purement et simplement ignorées !

La simple manipulation suivante permet d'en cerner le fonctionnement :

`td_scripts$` **`TOTO='Une chaine de caractères.'`**

`td_scripts$` **`echo $TOTO`** affiche : `td_scripts$` **`Une chaine de caractères.`**

`td_scripts$` **`bash`**

`td_scripts$` **`echo $TOTO`** affiche : `td_scripts$` RIEN ! la variable TOTO n'est pas initialisée dans ce nouvel environnement.

Pour l'exporter, il suffit d'utiliser la commande interne du shell **`export`** après avoir déclaré la variable : `td_scripts$` **`export TOTO`** ; ou lors de sa déclaration : `td_scripts$` **`export TOTO='ma chaine de caractères'`**

> **Note :** L'environnement du processus fils est dupliqué (c'est à dire copié !), donc si dans le shell fils, la variable TOTO est modifiée ou détruite (par exemple via la commande**`unset TOTO`**), l'environnement du père n'est nullement affectée, et une fois de retour dans ce dernier, TOTO aura toujours sa valeur initiale.

3.3. Les options du shell

Les shells permettent de paramétrer certaines de leurs fonctionnalités qui peuvent être activées ou désactivées via l'utilisation des options **`-o`** et **`+o`** de la commande interne **`set`**

Exemple :

`td_scripts$` **`set +o noclobber`** indique au shell d'émettre un avertissement en cas d'existence d'un fichier cible lors d'une redirection simple en sortie.

Pour voir l'ensemble de ces fonctionnalités, se reporter à la page de man du shell utilisé...

4. La programmation

Voilà maintenant la substance même du scripting : les commandes, qui ont jusque là été saisies interactivement, vont être regroupées dans un fichier texte et destinées à être exécutées (ou pas !), séquentiellement (ou pas !) par un shell afin d'obtenir un résultat donné (et de préférence toujours le même !) pour une valeur initiale donnée.

4.1. Les variables, interprétation, substitution et protection des commandes

Au sein d'un script, les variables du shell telles qu'elles ont été décrites infra permettront de mémoriser des données en cours de traitement ou des états du programme durant son exécution : savoir les manipuler, et comprendre comment le shell les interprète ou les substitue est donc primordial.

Au lancement d'un script, le shell chargé de son exécution va commencer à exécuter séquentiellement chacune des lignes le composant, et après une vérification (lexicale et syntaxique) des éventuelles erreurs qu'il pourrait contenir, procéder à son évaluation (phase d'interprétation et de substitution des variables et des commandes), pour enfin l'exécuter et poursuivre le parcours du fichier.

> **Note :** Contrairement à d'autres langages interprétés, et à fortiori aux langages compilés, le shell ne connait pas d'étape de parsing global du code source à exécuter : les erreurs éventuelles ne seront donc détectées qu'au cas par cas, au fur et à mesure de l'exécution.

4.1.1. Nommage, déclaration et suppression de la définition des variables

Pour être valide, le nom de toute variable doit obligatoirement débuter par un des caractères compris dans [a-zA-Z_] : une lettre (majuscule ou minuscule) ou un tiret bas. Les caractères suivants pouvant être compris dans l'ensemble [a-zA-Z0-9_] : c'est à dire les mêmes plus les chiffres.

Exemples de noms valides :

```
_UnNombre
Un_Nombre
Un_Nombre_
```

Exemples de noms invalides :

```
1_Nombre
-Un_Nombre
/Un_Nombre_
```

La déclaration d'une variable est optionnelle (elle est alors implicitement déclarée lors de son affectation), mais peut faire l'objet d'une instruction particulière via les commandes internes **`declare`** ou **`typeset`** qui suivent la syntaxe suivante :

```
declare | typeset [option] [nom[=affectation]]
```

Utilisées seules, les commandes **`declare`** ou **`typeset`** se comportent comme la commande **`set`**, et affichent l'ensemble des variables déclarées dans l'environnement courant. Les options permettent notamment de spécifier un *type* à la variable faisant l'objet de la déclaration : **`-a`** indique un tableau, **`-i`** un nombre entier, l'option particulière **`-r`** marque la variable "en lecture seule" : elle ne peut alors être modifiée, ni supprimée ultérieurement (il s'agira donc d'une constante !).

> **Note :** Par défaut une variable du shell représente une chaine de caractères.

Exemples de déclarations de variables :

`_Unechaine=texte` déclare la variable _UneChaine et lui affecte la valeur 'texte'.
`_Unechaine2="texte comportant des espaces"` lui affecte la valeur 'texte comportant des espaces'.
`typeset _Unechaine` déclare la variable _UneChaine et lui affecte la valeur "chaine vide".
`declare -i Un_Nombre=25` déclare la variable Un_Nombre en tant que valeur numérique et lui affecte la valeur 25.

La suppression d'une déclaration de variable s'effectue par appel de la commande interne **`unset [nom]`**.

Exemples de suppressions de variables :

`_Unechaine=texte; unset _Unechaine` déclare la variable _UneChaine et la supprime aussitôt.
`declare -i -r Un_Nombre=25; unset Un_Nombre` la tentative de suppression déclenche une erreur...

4.1.2. Les variables spéciales du shell

Au cours de son exécution, un shell positionne un certain nombre de variables qui fournissent des informations de différentes natures. La liste exhaustive de ces variables est fournie ci-après :

4.1.2.1. Les paramètres positionnels

- **`$#`** contient le nombre d'arguments reçus par le script en cours d'exécution.
- **`$0`** représente le nom du script.
- **`$1, $2, $3 ... $9`** les 9 premiers arguments passés au script.
- **`${10}, ${11}, ${12} ...`** les arguments suivants passés au script (uniquement pour bash et ksh).

- **`$*`** et **`$@`** contiennent la liste de l'ensemble des arguments passée au script. La différence entre les deux formes si situe au niveau de leur utilisation avec des guillemets : **`"$*"`** ne protège pas les éventuels arguments contenant des séparateurs alors que **`$@`** le fait !

La commande interne **`shift [n]`** permet de décaler (de n positions si n est fourni) vers la gauche l'ensemble des arguments. Elle met à jour l'ensemble des paramètres positionnels à l'exception de $0 qui représentera toujours le nom du script. Cette commande peut être utile pour traiter successivement l'ensemble des arguments fournis à un script.

4.1.2.2. Les variables d'état

- **`$?`** contient le code de retour d'une commande. Ce code est un entier compris entre 0 et 255, et permet de diagnostiquer la réussite (0) ou l'échec d'une commande unix (>0).
- **`$$`** représente le PID du shell en cours d'exécution.
- **`$!`** représente le PID du dernier processus lancé en arrière plan.

La commande interne **`exit [n]`** permet d'arrêter l'exécution et de transmettre le code de retour d'un script. Appelée sans argument, **`exit`** arrête l'exécution du script et retourne 0. A défaut d'appel de **`exit`**, le script se termine en retournant 0.

4.1.2.3. La variable IFS

Cette variable particulière contient en fait les caractères utilisés par le shell comme séparateur (par défaut espace, tabulation et retour à la ligne). Elle peut être modifiée en cours de script.

Exemples d'utilisation :

```
OLDIFS="$IFS"
IFS=$'\n'
...
IFS="$OLDIFS"
```

Par ces quelques lignes, le fonctionnemment du shell est modifié pour ne considérer que le saut à la ligne comme séparateur, afin de permettre de traiter des lignes entières comportant des espaces ou des tabulation.

4.1.3. Interprétation et substitution des variables

La syntaxe du shell prévoit donc l'utilisation de certains caractères spéciaux qui subiront un traitement particulier lors de l'interprétation d'une ligne de commande ... et c'est la principale raison de la restriction des possibilités de nommage des variables à un jeu particulier de caractères.

Ces caractères spéciaux sont les suivants :

- **espace tabulation saut de ligne** sont les séparateurs de mots dans la ligne de commande (cf. **IFS**).
- **;** est le séparateur de commandes.
- **" "" ** sont les caractères de protection.
- **&** signifie l'exécution en arrière plan.
- **| < << > >>** signifient les tubes et les redirections.
- **() {}** signifient les regroupement de commandes.
- *** ? [ ] ?() +() *() !() @()** sont les caractères de génération de noms de fichiers.
- **$ ${}** permettent d'accéder à la valeur d'une variable.
- **` $()** signifient la substitution de commandes.

4.1.3.1. Utilisation des caractères de protection :

Les **'** *apostrophes* ou *simples quotes* permettent de retirer la signification de tous les caractères spéciaux du shell ; elles doivent être appariées dans une même ligne de commande, et ne se protègent pas elles-mêmes.

Exemples d'utilisation :

echo $PATH affiche la valeur de la variable $PATH.
**echo '$PATH'** affiche la chaine '$PATH'.
echo 'erreur d'utilisation' affiche > ... commande non terminée.
echo 'erreur d'\"utilisation' affiche erreur d'utilisation (par appariement des ').

Le caractère **** *anti-slash* permet de retirer la signification du caractère spécial du shell le suivant immédiatement. Pour afficher le caractère \, il faut le doubler !

Exemples d'utilisation :

echo * affiche les fichiers présents dans le répertoire courant.
echo * affiche le caractère *.
echo 'erreur d'\"utilisation' affiche erreur d'utilisation (traitement du ' non apparié).

Les **"** *guillemets* ou *doubles quotes* permettent de retirer la signification de tous les caractères spéciaux du shell sauf **$** et **${}**, **`** et **$()** et enfin **** ainsi qu'elle-même; à l'instar de **'** elles doivent être appariées dans une même ligne de commande, et ne se protègent pas elles-mêmes.

Exemples d'utilisation :

echo "Le contenu de la variable PATH=$PATH" affiche le contenu de la variable $PATH.
**echo "Le contenu de la variable PATH=\"$PATH\""** affiche la chaine '$PATH'.
echo "ls -l * 1>liste 2&>1" affiche > affiche ls -l * 1>liste 2&>1

`echo "$(ls -l * 1>liste 2&>1)"'` n'affiche rien mais crée le fichier liste.

4.1.3.2. Isolation d'un nom de variable :

Il est des cas particulies où il ne sera pas possible, même en utilisation les possibilités d'échappement, d'obtenir le résultat escompté : il faut alors pouvoir isoler le nom de la variable utilisée (c'est à dire forcer le shell à n'utiliser qu'une partie de la commande pour en extraire une valeur de variable). C'est notamment le cas lors de la concaténation du contenu d'une variable à une chaine de caractères débutant par un des caractères appartenant à l'ensemble du jeu de caractères valides pour le nommage d'une variable.

Cette isolation s'obtient en utilisant la syntaxe **`${VARIABLE}`**. Exemple :

On cherche à créer des fichiers nommés FIC001_LISTE.txt à partir d'une variable NUM contenant des valeurs telles que '001', '002'... la syntaxe

`touch FIC$NUM_LISTE.txt` aboutira à créer un fichier FIC.txt, dans la mesure où le shell considère une variable NUM_LISTE qui n'est pas définie.

La solution passe donc par isoler le nom de la variable NOM : **`touch FIC${NUM}_LISTE.txt`** .

4.1.3.3. Substitution de variables :

Sous le terme de substitution de variables, se cache simplement la possibilité d'attribuer une valeur par défaut aux variables (qu'elles soient non initialisées).

- **`${variable:-valeur}`** si la variable n'est pas vide, la substitution retourne sa valeur, sinon elle retourne **`valeur`**.
- **`${variable:=valeur}`** si la variable n'est pas vide, la susbtitution retourne sa valeur, sinon elle retourne **`valeur`** et variable est également affectée à **`valeur`**.
- **`${variable:+valeur}`** si la variable est pas vide, la substitution retourne **`valeur`**, sinon elle retourne sa valeur, c'est à dire vide.
- **`${variable:?message}`** si la variable n'est pas vide, la substitution retourne sa valeur, sinon le shell affiche le nom de la variable suivi de la chaine de caractères **`message`**. De plus, si cette forme de substitution est utilisée dans un script shell, celui s'arrête immédiatement après l'affichage du message.

Exemples d'utilisation :

`VAR=BONJOUR ; echo ${VAR:-'la variable VAR est vide !'}`
affiche "BONJOUR".
`unset VAR ; echo ${VAR:-'la variable VAR est vide !'}`

affiche "la variable VAR est vide !".

```
VAR=BONJOUR ; echo ${VAR:='la variable VAR est pas vide !'}; echo $VAR
```

affiche 2 fois "BONJOUR".

```
unset VAR ; echo ${VAR:='la variable VAR est vide !'}; echo $VAR
```

affiche 2 fois "la variable VAR est vide !".

```
unset VAR ; echo ${VAR:+'la variable VAR est vide !'}
```

n'affiche ... rien.

```
VAR=BONJOUR ; echo ${VAR:+'la variable VAR est pas vide !'}
```

affiche "la variable VAR est pas vide !".

```
unset VAR ; echo ${VAR:?'la variable VAR est vide !'}; echo "suite ..."
```

affiche "la variable VAR est vide !".

```
VAR=BONJOUR ; echo ${VAR:?'la variable VAR est vide !'}; echo $VAR
```

affiche 2 fois "BONJOUR".

4.1.4. Substitution de commandes

La substitution de commandes permet l'utilisation du résultat de l'exécution d'une commande dans une autre commande, ou de l'affecter à une variable. La syntaxe peut indifféremment prendre 2 formes d'écriture :

`$(COMMANDE)` ou bien **`` `COMMANDE` ``**

Exemples d'utilisation :

```
echo "liste des fichiers du répertoire : $(ls)"
nbre=$(ls -l | wc -l); nbre=`expr $nbre-1`; echo "nbre de fichiers : $nbre"
```

4.1.5. Interprétation d'une ligne de commande

L'ensemble des mécanismes vus précédemment peuvent être mis en jeu lors de l'exécution d'une ligne de commande. Ils sont toutefois exécutés dans un ordre précis :

1. Isolation des mots séparés par caractères séparateurs (par défaut : espace, tabulation, saut de ligne).
2. Traitement des caractères de protection (`" "" \`).
3. Substitution des variables (`$ ${}`).
4. Substitution des commandes (`` ` $() ``).
5. Substitution des caractères de génération des noms de fichiers (`'* ? [] ?() +() *() !() @()`).
6. Traitement des tubes et des redirections.
7. Lancement de la commande.

4.2. Les opérateurs et les structures de contrôle

Le flux d'exécution peut (et doit) être contrôlé en fonction des résultats sucessifs des commandes exécutées ou de l'état de variables d'exécution. Les opérateurs du shell permettent de contrôler simplement le lancement ou non d'une commande en fonction du résultat d'une précédente commande ; alors que les structures de contrôle permettent d'agir plus finement sur le flux d'exécution s'un script.

4.2.1. Les opérateurs du shell

Les opérateurs lient entre elles différentes commandes, et sont au nombre de deux :

- **`&&`** signifie ET logique, et exécute la commande suivante si la précédente a retourné VRAI (code retour 0).
- **`||`** signifie OU logique, et exécute la commande suivante si la précédente a retourné FAUX (code retour >0).

La syntaxe d'utilisation est la suivante :

- **`CMD1 && CMD2`** : CMD2 n'est exécutée que si CMD1 a retourné VRAI, l'ensemble ne retourne VRAI que si toutes les commandes ont retourné VRAI.
- **`CMD1 || CMD2`** : CMD2 n'est exécutée que si CMD1 a retourné FAUX, l'ensemble retourne VRAI si au moins une des commandes a retourné VRAI.

Exemple d'utilisation :

```
make bzImage && make modules && make modules_install
```

Cette séquence (typique de la compilation du noyau Linux) va enchainer les compilations :
- du noyau lui-même (make bzImage).
- des modules, si la compilation du noyau a réussi (make modules).
- et enfin, installer les modules (dans /lib/modules/) si leur compilation a réussi (make modules_install).

4.2.2. Les structures de contrôle

Deux catégories de structures programmatiques permettent de contrôler l'exécution des scripts soit conditionnellement, en fonction de *tests* ; ou en opérant via des *boucles* itératives qui permettront donc de répéter une action tant qu'une condition sera vérifiée.

4.2.2.1. Les tests

Le shell offre la possibilité d'exécuter des branchements conditionnels dans le flux d'exécution en fonction du résultat d'un ou plusieurs tests qui sont introduits par différentes syntaxes possibles :

- soit par la commande interne **`[ [val1] op val2 ]`**

- soit par la commande interne **`[[ [val1] op val2 ]]`**
- ou encore par la commande externe **`test [val1] op val2`**
- en ce qui concerne les expressions arithmétiques il faut utiliser soit la commande interne **`(( [val1] op val2 ))`** ou **`let`**
- ou la commande externe **`expr [val1] op val2`**

Tableau 1. Opérateurs de tests portant sur les fichiers :

commande	opérateur	signification	compatibilité
test ou **[** ou **[[**	**-a nomfic** ou **-e nomfic**	VRAI si le fichier existe	ksh ou bash uniquement
test ou **[** ou **[[**	**-s nomfic**	VRAI si le fichier n'est pas vide	**sh** et suivants
test ou **[** ou **[[**	**-f nomfic**	VRAI si le fichier est de type ordinaire	**sh** et suivants
test ou **[** ou **[[**	**-d nomfic**	VRAI si le fichier est un répertoire	**sh** et suivants
test ou **[** ou **[[**	**-h nomfic**	VRAI si le fichier est un lien symbolique	**sh** et suivants
test ou **[** ou **[[**	**-L nomfic**	VRAI si le fichier est un lien symbolique	ksh ou bash uniquement
test ou **[** ou **[[**	**-b nomfic**	VRAI si le fichier représente un périphérique bloc	**sh** et suivants
test ou **[** ou **[[**	**-c nomfic**	VRAI si le fichier représente un périphérique caractères	**sh** et suivants
test ou **[** ou **[[**	**-p nomfic**	VRAI si le fichier est un tube nommé	**sh** et suivants
test ou **[** ou **[[**	**-S nomfic**	VRAI si le fichier est un socket	ksh ou bash uniquement
test ou **[** ou **[[**	**-r nomfic**	VRAI si le fichier est accessible en lecture	**sh** et suivants
test ou **[** ou **[[**	**-w nomfic**	VRAI si le fichier est accessible en écriture	**sh** et suivants
test ou **[** ou **[[**	**-x nomfic**	VRAI si le fichier est possède le droit d'éxécution	**sh** et suivants
test ou **[** ou **[[**	**-u nomfic**	VRAI si le fichier est possède le setuid-bit	**sh** et suivants
test ou **[** ou **[[**	**-g nomfic**	VRAI si le fichier est possède le setgid-bit	**sh** et suivants
test ou **[** ou **[[**	**-k nomfic**	VRAI si le fichier est possède le sticky-bit	**sh** et suivants
test ou **[** ou **[[**	**-x nomfic**	VRAI si le fichier est possède le sticky-bit	**sh** et suivants
test ou **[** ou **[[**	**-O nomfic**	VRAI si l'utilisateur est propriétaire du fichier	ksh ou bash uniquement
test ou **[** ou **[[**	**-G nomfic**	VRAI si l'utilisateur appartient au groupe propriétaire du fichier	ksh ou bash uniquement
test ou **[** ou **[[**	**nomfic1 -nt nomfic2**	VRAI si le fichier nomfic1 est plus récent que le fichier nomfic2	ksh ou bash uniquement
test ou **[** ou **[[**	**nomfic1 -ot nomfic2**	VRAI si le fichier nomfic1 est plus ancien que le fichier nomfic2	ksh ou bash uniquement
test ou **[** ou **[[**	**nomfic1 -ef nomfic2**	VRAI si les fichiers nomfic2 et nomfic2 référencent la même inode	ksh ou bash uniquement
test ou **[** ou **[[**	**-t [descripteur]**	VRAI si le descripteur (1 par défaut) est associé à un terminal	**sh** et suivants

Les tests peuvent également porter sur des chaines de caractères (valeurs de variables ou constantes).

Tableau 2. Opérateurs de tests portant sur les chaines de caractères :

commande	opérateur	signification	compatibilité
test ou [ou [[	**-z chaine**	VRAI si la chaine est vide	**sh** et suivants
test ou [ou [[	**-n chaine** ou **nomfic**	VRAI si la chaine n'est pas vide	**sh** et suivants
test ou [ou [[	**chaine1 = chaine2**	VRAI si les deux chaines sont égales	**sh** et suivants
test ou [ou [[	**chaine1 != chaine2**	VRAI si les deux chaines sont différentes	**sh** et suivants
[[	**chaine1 < chaine2**	VRAI si la chaine1 est lexicographiquement avant la chaine2	ksh ou bash uniquement
[[	**chaine1 > chaine2**	VRAI si la chaine1 est lexicographiquement après la chaine2	ksh ou bash uniquement
[[	**chaine1 = modèle**	VRAI si la chaine correspond au modèle (le modèle est une expression correspondant à la syntaxe des recherche de fichiers)	ksh ou bash uniquement
[[	**chaine1 != modèle**	VRAI si la chaine diffère du modèle	ksh ou bash uniquement
expr ou ((	**chaine1 \& chaine2**	VRAI si les 2 chaines ne sont pas nulles	**sh** et suivants
expr ou ((	**chaine1 | chaine2**	VRAI si l'une des 2 chaines n'est pas nulle	**sh** et suivants
expr ou ((	**chaine : expression régulière**	compare la chaine à l'expression régulière.	**sh** et suivants

Ou enfin sur des nombres. En ce qui concerne les expressions arithmétiques, il faut utiliser soit les commandes `(( ... ))` ou **`expr`** dans la mesure où les précédentes commandes ne peuvent qu'effectuer des comparaison sur des valeurs numériques.

Tableau 3. Opérateurs de tests portant sur des nombres :

commande	opérateur	signification	compatibilité
test ou [ou [[	**nb1 -eq nb2**	VRAI si les nombres sont égaux	**sh** et suivants
test ou [ou [[	**nb1 -ne nb2**	VRAI si les nombres sont différents	**sh** et suivants
test ou [ou [[	**nb1 -lt nb2**	VRAI si nb1 est strictement inférieur à nb2	**sh** et suivants
test ou [ou [[	**nb1 -le nb2**	VRAI si nb1 est strictement inférieur ou égal à nb2	**sh** et suivants
test ou [ou [[	**nb1 -gt nb2**	VRAI si nb1 est strictement supérieur à nb2	**sh** et suivants

commande	opérateur	signification	compatibilité
test ou **[** ou **[[**	**nb1 -ge nb2**	VRAI si nb1 est strictement supérieur ou égal à nb2	**sh** et suivants
expr ou ((	**nb1 + nb2**	addition	**sh** et suivants
expr ou ((	**nb1 - nb2**	soustraction	**sh** et suivants
expr ou ((	**nb1 * nb2**	multiplication	**sh** et suivants
expr ou ((	**nb1 / nb2**	division	**sh** et suivants
expr ou ((	**nb1 % nb2**	modulo	**sh** et suivants
((uniquement	**~nb1**	complément à 1	**ksh** et **bash** uniquement
((uniquement	**nb1 > nb2**	décalage à droite de nb2 bits sur nb1	**ksh** et **bash** uniquement
((uniquement	**nb1 < nb2**	décalage à gauche de nb2 bits sur nb1	**ksh** et **bash** uniquement
((uniquement	**nb1 & nb2**	ET bit à bit	**ksh** et **bash** uniquement
((uniquement	**nb1 | nb2**	OU bit à bit	**ksh** et **bash** uniquement
((uniquement	**nb1 ^ nb2**	OU EXCLUSIF bit à bit	**ksh** et **bash** uniquement
expr ou ((	**nb1 \> nb2**	VRAI si nb1 est strictement supérieur à nb2	**sh** et suivants
expr ou ((	**nb1 \< nb2**	VRAI si nb1 est strictement inférieur à nb2	**sh** et suivants
expr ou ((	**nb1 \>= nb2**	VRAI si nb1 est supérieur ou égal à nb2	**sh** et suivants
expr ou ((	**nb1 \<= nb2**	VRAI si nb1 est inférieur ou égal à nb2	**sh** et suivants
expr ou ((	**nb1 = nb2**	VRAI si nb1 est égal à nb2	**sh** et suivants
((uniquement	**nb1 == nb2**	VRAI si nb1 est égal à nb2	**ksh** ou **bash**
((uniquement	**nb1 [opération arithmétique]= nb2**	assignement (éventuellement précédé de l'opération)	**ksh** ou **bash**
expr ou ((	**nb1 != nb2**	VRAI si nb1 est différent de nb2	**sh** et suivants
expr ou ((	**-nb1**	opposé de nb1	**sh** et suivants
expr ou ((	**-nb1**	opposé de nb1	**sh** et suivants

Afin de compléter cette armada de tests, le shell dispose également d'opérateurs permettant d'influer sur un test ou lier logiquement plusieurs tests :

- L'opérateur **`!`** : signifie la négation du test le suivant.
- L'opérateur **`-a`** : effectue un ET logique entre deux tests.
- L'opérateur **`-o`** : effectue un OU logique entre deux tests.

Ces opérateurs ont un ordre de priorité décroissant, c'est à dire que `!` sera évalué avant `-a`, qui sera lui-même évalué avant `-o`. Toutefois, cet ordre d'évaluation peut être imposé par l'utilisation des caractères de regroupement `\( ... \)`.

La syntaxe de la commande `[[ ... ]]` s'est vue enrichie par rapport à la commande `test` (ou `[ ... ]`) basique, et notamment en ce qui concerne les chaines de caractères pour lesquelles l'usage des guillemets n'est plus nécessaire, certains opérateurs ont été rajoutés (cf. tableau) et les opérateurs de négation et de regroupement ont été alignés sur la syntaxe du C :

- L'opérateur `!` reste tel quel.
- L'opérateur `-a` (ET logique) devient `&&` (idem pour la commande `(( ... ))`).
- L'opérateur `-o` (OU logique) devient `||` (idem pour la commande `(( ... ))`).
- Et les parenthèses de regroupement ne doivent plus être échappées.

Exemples d'utilisation :

`test -a lefichier; echo $?` : teste l'existence du fichier lefichier.
`[ -a lefichier ]; echo $?` : teste l'existence du fichier lefichier.
`[ -b /dev/sda ]; echo $?` : teste si /dev/sda est un périphérique bloc.
`[ "ch1" ="$ch2" ]; echo $?` : teste l'égalité des deux variables chaine ch1 et ch2.
`[ ch1 =$ch2 ]; echo $?` : teste l'égalité des deux variables chaine ch1 et ch2.
`[[ ch1 =$ch2 ]]; echo $?` : teste l'égalité des deux variables chaine ch1 et ch2.
`[ -w $fic1 -a \( -e $rep1 -o -e $rep2 \) ]; echo $?` : devinez ...
`[[ -w $fic1 && ( -e $rep1 || -e $rep2 ) ]]; echo $?` : devinez ...
`X=10; echo \`expr $X + 10\`; echo $?; echo $X` : calcule X + 10, X reste identique.
`X=10; echo $(($X + 10)); echo $?; echo $X` : calcule X + 10, X reste identique.

4.2.2.2. Les branchements : structures if, elif et case

Les tests peuvent donc permettre d'éxecuter conditionnellement un partie d'un programme en fonction de son résultat. La structure pour procéder à ces branchements dans le code est la structure **if** : *si* la condition est vraie, alors le programme exécute un bloc de code. Mais elle peut se compliquer quelque peu en rajoutant les notions de *sinon* (**else**) et de *sinon, si* (**elif**). La syntaxe est la suivante :

```
if test
then
 ... bloc de code ...
[elif test]
then
 ... bloc de code ...
[else]
 ... bloc de code ...
fi
```

ou encore

```
if test ; then
 ... bloc de code ...
[elif test] ; then
```

```
 ... bloc de code ...
[else]
 ... bloc de code ...
fi
```

Exemples d'utilisation :

```
if [ -a ~/.bashrc ] ; then
 echo "J'ai bien un fichier de configuration personnalisé pour le shell bash !"
 echo "En voici son contenu :"
 cat ~/.bashrc
fi
```

```
if [ -z $a ] ; then
 echo "La variable a est indéfinie !"
 exit 1
fi
if (( $a == 10)) ; then
 echo "La variable a vaut $a !"
elif [ $a -eq 20 ] ; then
 echo "La variable a vaut $a !"
elif test $a -lt 20 ; then
 echo "La valeur de la variable a est inférieure à 20 ..."
 echo "La valeur de la variable a est différente de 10 ..."
else
 echo "La valeur de la variable a est supérieure à 20 ..."
fi
```

Dans certains cas, il sera possible de remplacer l'utilisation de nombreuses conditions **if - elif** qui deviennent rapidement illisibles par la structure **case**. Sa syntaxe est la suivante :

```
case $variable in
modele1)
 ... bloc de code ...
 ;;
modele2)
 ... bloc de code ...
 ;;
*)
 ... bloc de code ...
 ;;
esac
```

Le modèle peut contenir des caractères spéciaux qui serviront à construire un modèle générique :

Tableau 4. Caractères génériques servant à forger un modèle :

caractère spécial	signifie	compatibilité
*	**0 à n caractères quelconques**	**sh** et suivants
?	**1 caractère quelconque**	**sh** et suivants
[aAbBc]	**1 caractère parmis ceux figurant entre les crochets**	**sh** et suivants
[!aAbBc]	**1 caractère parmis ceux ne figurant pas entre les crochets**	**sh** et suivants

caractère spécial	signifie	compatibilité
?(expression)	**0 ou 1 fois l'expression entre parenthèses**	ksh ou bash uniquement ; pour bash, il faut activer l'option **shopt -s extglob**
***(expression)**	**0 à n fois fois l'expression entre parenthèses**	ksh ou bash uniquement ; pour bash, il faut activer l'option **shopt -s extglob**
+(expression)	**0 à n fois fois l'expression entre parenthèses**	ksh ou bash uniquement ; pour bash, il faut activer l'option **shopt -s extglob**
@(expression)	**1 fois fois l'expression entre parenthèses**	ksh ou bash uniquement ; pour bash, il faut activer l'option **shopt -s extglob**
!(expression)	**0 fois fois l'expression entre parenthèses**	ksh ou bash uniquement ; pour bash, il faut activer l'option **shopt -s extglob**

Exemple d'utilisation :

```
case "$1" in
start)
 echo "Démarrage du daemon ..."
 ;;
stop)
 echo "Arrêt du daemon ..."
 ;;
*)
 echo "Usage : $SCRIPTNAME {start|stop}"
 exit 3
 ;;
esac
```

4.2.2.3. Les boucles : structures for, while et until

Les boucles sont des structures de contrôle permettant d'itérer un bloc de code autant de fois que nécessaire. L'itération peut être déterminée soit par le nombre de valeurs d'une liste dans le cas de la boucle **for**, soit par le résultat d'un commande ou d'un test dans le cas des boucles **while** et **until** :

- Le bloc code compris dans un boucle **for** sera exécuté pour chaque élement de la liste ; la liste peut être fournie explicitement, par substitution de variable, par susbtition de commande ou par évaluation des caractères spéciaux du shell. Si aucune liste n'est fournie, la boucle porte sur les arguments passés dans la ligne de commande (évaluation de la variable `$*`. Lors de chaque itération, la variable de boucle prend la valeur suivante dans la liste.
- Le bloc code compris dans un boucle **while** sera exécuté tant que la valeur retournée par la commande de contrôle sera VRAI ; dès qu'elle retournera FAUX, le flux d'exécution passe à la commande suivant

immédiatement le mot-clé **done**.

- Le bloc code compris dans un boucle **until** sera exécuté tant que la valeur retournée par la commande de contrôle n'est pas VRAI.

La syntaxe d'une boucle **for** est la suivante :

```
for var [in liste]
do
 ... bloc de code ...
done
```

ou

```
for var [in liste] ; do
 ... bloc de code ...
done
```

La syntaxe d'une boucle **while** est la suivante :

```
while expression
do
 ... bloc de code ...
done
```

ou

```
while expression ; do
 ... bloc de code ...
done
```

La syntaxe d'une boucle **until** est la suivante :

```
until expression
do
 ... bloc de code ...
done
```

ou

```
until expression ; do
 ... bloc de code ...
done
```

Exemples d'utilisation :

```
for i in *; do ls -i; done;
for i in $(cat /etc/passwd | cut -d: -f1); do echo $i; done;
for i in ‘seq 1 10‘ ; do echo $i; done
for i in 1 2 3 4 5 6 7 8 9 10 ; do echo $i; done

while true ; do echo "Saisissez un chiffre"; read CHIFFRE; echo "$CHIFFRE erroné. essayez encore..."; done;
REP=0; while (( $REP < 10 )); do echo $REP; REP=‘expr $REP + 1‘; done;

until false ; do echo "Saisissez un chiffre"; read CHIFFRE; echo "$CHIFFRE erroné. essayez encore..."; done;
REP=0; until [ $REP -eq 10 ] ; do echo $REP; ((REP+=1)); done;
```

Lors de l'exécution d'une boucle, les commandes internes **continue** et **break** permettent de modifier le flux de commandes en, respectivement, passant à l'itération suivante et en interrompant le cours de la

boucle.

Exemples d'utilisation :

```
for i in `seq 1 10`
do
 if (( $i == 5 ))
 then
  continue # n'affichera pas 5
 fi
 if (( $i > 8 )) # interromp la boucle pour i=9
 then
  break
 fi
 echo $i
done

clear
while true ; do         # boucle infinie
        echo "Saisissez un nombre (0 non traité - 999 pour terminer)";
        read NOMBRE;
        if [  ! $(echo $NOMBRE | grep "^[ [:digit:] ]*$") ] ; then
                echo "On vous a demandé un nombre !"
                continue
        fi
        if [ $NOMBRE -eq 0 ] ; then
                continue
        elif [ $NOMBRE -eq 999 ] ; then
                break
        else
                echo "Vous avez saisi le nombre $NOMBRE (999 pour terminer)";
                sleep 5
                clear
        fi
done;
```

4.3. Les fonctions

Dernière forme de structuration du code, le shell prévoit la possibilité de création de fonctions qui permettent de factoriser des portions de code destinées à être exécutées plusieurs fois dans un script, ou bien encore de créer des librairies de code réutilisables dans différents scripts. Une fonction doit être définie avant son utilisation (son appel) ; la syntaxe pour la définition d'une fonction est la suivante :

```
nomdelafonction() {
 ... bloc de code ...
```

}

Une seconde forme est également possible (uniquement avec ksh ou bash)

function nomdelafonction {
... bloc de code ...
}

Quant à la syntaxe d'appel (utilisation) d'une fonction est tout simplement :
nomdelafonction [parametre1 parametre2 ...]
... à la condition que la fonction appelée ait été définie avant son appel, sous peine d'affichage d'une message d'erreur lors de l'exécution du script.

Une function, comme obéit aux mêmes principes que toute commande Unix : elle retourne un code de retour (un entier compris entre 0 et 255) qui vaut par défaut 0, mais cette valeur peut être modifiée en utilisant la commande interne **return [valeur]**. Ce code de retour est transmis à la variable spéciale **$?** dès le retour de la fonction, et peut donc être testé comme de coutume.

Les variables utilisateur sont globales pour l'ensemble d'un script, c'est à dire qu'une variable déclarée dans le script sera accessible et modifiable dans le corps d'une fonction, et qu'à l'identique une variable déclarée ou affectée dans le corps d'une fonction sera accessible au reste du script. Or, dans nombre de cas, les variables de travail d'une fonction n'ont à être accessibles ni publiées à l'extérieur de celle-ci ; la commande interne **typeset** permet de déclarer des variables *locales* dans une fonction : elle sont alors ni accessibles ni publiées en dehors du corps de la fonction.

Quant aux paramètres passés à une fonction, ils sont accessibles à celle-ci via le jeu de variables spéciales *locales* **$1, $2, ..., $*, $@, $#**.

> **Note :** La variable spéciale **$0** reste quant à elle accessible, en lecture-seule, et contient toujours le nom du script en cours d'exécution.

Exemples d'utilisation :

```
#!/bin/bash

function saisie_nombre {
        echo -n "$0> Saisissez un nombre (999 pour terminer) : "
        read NOMBRE;
         if [  ! $(echo $NOMBRE | grep "^[ [:digit:] ]*$") ] ; then
                NOMBRE=""
                return 1
        fi
}

clear
while true ; do         # boucle infinie
        saisie_nombre
        RES=$?
        if [ $RES -ne 0 ] ; then
                echo "On vous a demandé un nombre !"
                continue
        fi
```

```
        if [ $NOMBRE -eq 0 ] ; then
                continue
        elif [ $NOMBRE -eq 999 ] ; then
                break
        else
                echo "Vous avez saisi le nombre $NOMBRE (999 pour terminer)";
                sleep 5
                clear
        fi
done;
```

5. Les commandes - notion de filtre Unix

5.1. Définition d'un filtre Unix

Nombre de commandes Unix permettent de traiter un flux de données textuelles leur parvenant sur leur entrée standard, et restituent le résultat de leur opération sur la sortie standard : elles sont alors nommées *filtres*. Leur but principal est donc d'apporter une traitement particulier dans la chaine des traitements que constitue un script.

5.2. Les principaux filtres Unix

Cette section se bornera à citer les commandes incontournables pour l'écriture de scripts shell, il faudra donc se référer au man de chacune d'entre elles pour obtenir de plus amples renseignement quant à leur fonctionnement. La majorité d'entre elles se trouve dans le paquet debian coreutils !

Tableau 5. Opérateurs de tests portant sur des nombres :

commande	description synthétique
awk	Recherche et traite des modèles (langage de programmation à part entière, développé infra.)
base64	Encoder/décoder des données et les afficher sur la sortie standard.
comm	Compare ligne à ligne deux fichiers triés.
compress uncompress gzip gunzip bzip2 bunzip2	Compressent et décompressent des données.
cpio tar	Archivent des données.

commande	description synthétique
csplit	Découpe un fichier en sections définies par des lignes de contexte.
cut	Affiche des parties sélectionnées des lignes.
expand **unexpand**	Convertit les tabulations en espaces et vice-versa.
fmt	Formate simplement du texte.
fold	Coupe chaque ligne de texte à une longueur donnée.
grep egrep **fgrep rgrep**	Afficher les lignes correspondant à un motif donné (expressions rationnelles).
head	Affiche le début des fichiers.
join	Fusionne les lignes de deux fichiers ayant des champs communs.
nl	Numérote les lignes d'un fichier.
od	Affiche le contenu d'un fichier en octal ou sous d'autres formats.
paste	Regroupe les lignes de différents fichiers.
pr	Met en forme des fichiers de texte pour l'impression.
ptx	Génére un index croisé du contenu de fichiers.
rev	Renverse l'ordre des caractères pour chaque ligne.
sed	Filtre et transforme des textes (commande développé infra.)
shuf	Génère des permutations aléatoires.
sort tsort	Trie les lignes de fichiers texte
split	Découpe un fichier en différentes parties
sum cksum **md5sum** **sha1sum** **sha224sum** **sha256sum** **sha384sum** **sha512sum**	Calculent de sommes de contrôle (CRC).
tac	cat inversé...
tail	Affiche la dernière partie de fichiers.
tr	Convertit ou élimine des caractères.
uniq	Signale ou élimine les lignes répétées.
wc	Affiche le nombre de lignes, de mots et d'octets d'un fichier.
xargs	Construit et exécute des lignes de commandes à partir de l'entrée standard.

5.3. Les principales commandes

Comme la précédente, cette section ne fera que citer les commandes utiles à l'écriture de scripts shell.

Tableau 6. Opérateurs de tests portant sur des nombres :

commande	description synthétique
basename	Éliminer le chemin d'accès et le suffixe d'un nom de fichier.
chroot	
date	Affiche ou configure la date et l'heure du système.
dd	Convertit et copie un fichier.
df	Indique l'espace occupé par les systèmes de fichiers.
dirname	Ne conserve que la partie répertoire d'un nom de fichier.
du	Évaluer l'espace disque occupé par des fichiers.
echo	Affiche une ligne de texte.
env	Exécute un programme dans un environnement modifié.
factor	Affiche les facteurs premiers d'un nombre.
groups	Affiche les groupes auxquels appartient un utilisateur.
hostid	Affiche le numéro d'identification de l'hôte actuel.
id whoami	Affiche les UID et GID effectifs et réels.
install	Copie des fichiers et positionne leurs attributs.
logname	Afficherle nom de connexion de l'utilisateur
nice	Exécute un programme avec un niveau de priorité (politesse) modifié.
nohup	Exécute une commande en la rendant insensible aux déconnexions, avec une sortie hors terminal
pathchk	Vérifie la validité et la portabilité d'un nom de fichier
printenv	Affiche l'ensemble ou une partie des variables d'environnement.
printf	Formate et affiche des données.
readlink	Affiche la valeur d'un lien symbolique.
seq	Affiche une séquence de nombres.
shred	Écrit par dessus un fichier pour en camoufler le contenu, et optionnellement l'effacer.
sleep	Endort un processus pour une durée déterminée.
stat	Affiche l'état d'un fichier ou d'un système de fichiers.
stty	Modifie et afficher la configuration de la ligne de terminal.
sync	Vide les tampons du système de fichiers.
tee	Lit depuis l'entrée standard et écrit sur la sortie standard et dans des fichiers.
test	Vérifie le type d'un fichier, et comparer des valeurs.
touch	Modifie l'horodatage d'un fichier ; le crée s'il n'existe pas.
tty	Affiche le nom de fichier du terminal associé à l'entrée standard.
uname	Affiche des informations sur le système.
users who	Afficher le nom des utilisateurs actuellement connectés sur cette machine.
yes	Affiche indéfiniment une chaîne de caractères jusqu'à ce que le processus soit tué.

5.4. sed et awk

Deux des commandes filtres précédemment énumérées méritent une attention particulière car, outre les services qu'elles peuvent rendre dans le cadre du scripting, ce sont également des langages de programmation à part entière, dont l'étude sortirait toutefois du cadre présent document. Un survol de leur possibilité est toutefois nécessaire.

5.4.1. sed - Stream EDitor

Sed signifie "Stream EDitor" autrement dit "éditeur de flux". De par son mode de fonctionnement, Sed peut être défini comme un éditeur non-interactif.

L'éditeur de flux Sed lit les lignes d'un ou plusieurs fichiers ou depuis l'entrée standard, et y applique des commandes lues elles aussi depuis l'entrée standard sous forme d'expressions (commandes d'édition) ou depuis un fichier texte (script), et écrit le résultat du traitement sur la sortie standard. Il est possible de résumer ainsi le mécanisme de fonctionnement de Sed :

lecture d'une ligne sur le flux d'entrée
traitement de la ligne en fonctions des diverses commandes lues
affichage (ou non) du résultat sur la sortie standard
passage à la ligne suivante

Sed ne se limite toutefois pas à la simple lecture lecture séquentielle du flux d'entrée, il est possible de spécifier les lignes sur lequelles les commandes doivent opérer : elles acceptent des numéros de lignes, des intervalles, ou encore des expressions rationnelles (notées RE ou regex). La syntaxe d'utilisation est la suivante :
sed [-options] [commande] [<fichier(s)>]
ou sa forme plus complète :
sed [-n [-e commande] [-f script] [-i[.extension]] [l [cesure]] rsu] [commande] [<fichier(s)>]

La portée des traitements, bien que facultative, peut être indiquée, c'est *l'adressage d'une ou plusieurs lignes*, qui se traduit par la syntaxe :
[adresse[,adresse]][!]commande[arguments]

Le traitement à opérer n'est pas forcément unique ; plusieurs commandes peuvent être enchainées si elles sont entourées par des accolades :
[adresse[,adresse]][!]{commande1; commande2; commande3 }[arguments]

Les commandes suivent donc un éventuel adressage, et les plus courantes sont :
: Commentaire (pas d'adressage autorisé).
q : Quitter (adressage autorisé) .
d : Suppression de la ligne (adressage autorisé) .
p : Affichage (adressage autorisé).
n : Ligne suivante (adressage autorisé).
s/ch1/ch2/ : substitution (adressage autorisé).
y/ch1/ch2/ : Transposition (adressage autorisé).
a\texte : Ajout sur la ligne(une adresse autorisée).
i\texte : Insertion (une adresse autorisée).
c \texte : Echange le contenu de la ligne par le texte (adressage autorisé).

r fichier : Lecture (et insertion) du fichier (une adresse autorisée).
w fichier : Ecriture dans un fichier (une adresse autorisée).
= : Affiche le n° de ligne courante (une adresse autorisée).
l [largeur] : Affiche les caractères non imprimables et tronque éventuellement à N la ligne (adressage autorisé).

En outre, la commande de substitution peut se voir spécifier des drapeaux indiquant sa portée par rapport à ligne traitée :
g :susbtitution globale.
[0-9]* : N° d'occurence à traiter.
p : Affichage de la ligne(nécessite l'option -n).
w fic_sortie : Enregistrement dans le fichier spécifié.
e : Evaluation et exécution par le shell.
I : Insensibilité à la casse.

Exemples d'adressages et de commandes :

for i in $(seq 1 20); do echo "Ligne $i" >> fic.txt; done : résultat ???

sed -n 3p fic.txt : affiche la ligne 3 du fichier.
sed -n 1~2p fic.txt : affiche toutes les deux lignes à compter de la ligne 1 du fichier.
sed -n '3,6 p' fic.txt : affiche les lignes 3 à 6 du fichier.
sed -n '/Ligne [123789][0]*/p' fic.txt : affiche les lignes correspondant au motif (ER) fourni entre les délimiteurs '/'.
sed -n '/Ligne 3/,/Ligne 12/ p' fic.txt : affiche les lignes comprises enter deux correpondances.
sed -n '2,/Ligne 6/ p' fic.txt : affiche les lignes 2 jusqu'àt la correspondance.
sed -n '/Ligne 5/,+5 p' fic.txt : affiche les 5 lignes suivant la correspondance.

sed '10,19 d' fic.txt : Supprime les lignes 10 à 19.
sed '/1[0-9] d' fic.txt : Supprime les lignes 10 à 19.
sed '2,/Ligne [569]/ s/Ligne/Bonjour/' fic.txt : Substitue Ligne par Bonjour sur les lignes 2 à 5 et affiche les autres sans
sed -n '2,/Ligne [569]/ s/Ligne/Bonjour/' fic.txt : Substitue Ligne par Bonjour sur les lignes 2 à 5 les affiche.
sed -e "s/\([0-9][0-9]*\)/--\1--/" fic.txt : Substitue le n° suivant Ligne par -- numéro - via l'extraction de sous-chaine ciblé
sed -e "s/\(Ligne\).*\([0-9][0-9]*\)/--\2 \1--/" fic.txt : Affiche le -- n° Ligne -- via l'extraction de 2 sous-chaines ciblées.
sed -n '1~2 s/Ligne/Bonjour/ p; $a\LA FIN' fic.txt : Substitue Ligne par Bonjour une ligne sur 2 et ajoute 'LA FIN' en f

Exemples d'utilisation commune de sed (mono-lignes) :

- Supprimer les lignes vides : **sed '/^\t*$/d'** .
- Supprimer les commentaires : **sed 's/#.*$//'** .
- insérer une ligne vide avant la ligne repérée par le motif : **sed '/ER/{x;p;x;}'** .
- insérer une ligne vide après la ligne repérée par le motif : **sed '/ER/G'** .
- insérer une ligne vide avant et après la ligne repérée par le motif : **sed '/RE/{x;p;x;G;}'** .
- compter les lignes d'un fichier : **sed -n '$='** .
- Eliminer les espaces blancs en début de ligne : **sed 's/^[\t]*//'**.
- Eliminer les espaces blancs en fin de ligne : **sed 's/[\t]*$//'**.
- Aligner du texte à droite (ici 79° colonne) : **sed -e :a -e 's/^.\{1,78\}$/ &/;ta'** .
- Centrer du texte à droite (ici sur 80 colonnes) : **sed -e :a -e 's/^.\{1,77\}$/ &/;ta' -e 's/\(*\)\1\1/'** .
- Renverser l'ordre des caractères : **sed '/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//'** .
- Formater des nombres décimaux : **sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1.\2/;ta'** .

Sed est autrement plus riche que les fonctions basiques qui ont été décrites dans cette section, et mérite sûrement une étude plus poussée, ne serait-ce que par la lectrure attentive d'un des nombreux tutoriels qui vent être trouvés sur internet.

5.4.2. awk

awk est un autre outil particulièrement adapté au traitement de données textuelles, son nom provient des initiales de ses 3 concepteurs : AHO, WEINBERGER et KERNIGHAN. Il s'agit en fait d'un véritable langage de script mais ses fonctionnalités de base en font également un outil incontournable pour l'écriture de scripts.

La syntaxe de base est la suivante :

awk [options] '{actions [; action]}' [fichier]
ou
awk [options] -f script [fichier]

Si le fichier à traiter est omis, awk utilise les données arrivant sur son entrée standard.

Lors du parcours d'un fichier, **awk** procède au découpage de chacune des lignes (enregistrement) pour en extraire des variables (champs) sur lesquels il sera possible ensuite travailler. Ce découpage est effectué en fonction des valeurs d'un certain nombre de variables internes à **awk** :

Tableau 7. Opérateurs de tests portant sur des nombres :

variable	valeur par défaut	rôle
RS	\n (newline)	Record Separator : caractère séparateur d'enregistrement (lignes).
FS	Caractères blancs (espaces ou séparations)	Field Separator : caractère séparateur de champs.
OFS	espace	Output Field Separator : caractère séparateur de champ utilisé pour l'affichage.
ORS	\n (newline)	Output Record Separator : caractère séparateur d'enregistrement utilisé pour l'affichage.

Pour chacun des enregistrements lus, **awk** génère donc un certain nombre de variables utilisables dans le corps du script :

Tableau 8. Opérateurs de tests portant sur des nombres :

variable	valeur
$0	L'enregistrement en cours de traitement (la ligne).
NF	Nombre de champs de l'enregistrement courant.
$1, $2, ... $NF	Les champs découpés dans l'enregistrement courant.
NR	Numéro de l'enregistrement courant (1 pour la première ligne, etc...)
FNR	Indice de l'enregistrement courant relatif au fichier en cours de traitement.
FILENAME	Nom du fichier en cours de traitement.

Partant de là, il est déjà possible d'utiliser les fonctionnalités de base de **awk** pour récupérer et afficher certaines valeurs :

ps -edfal | awk '{print $3, $15}'
ps -edfal | awk '{print "Commande", $15, "lancee par", $3, "- durée :", $14}'
ls -l | grep -v total | awk '{ print "La taille du fichier",$9,"est",$5," octets."}'

Dans le dernier exemple, il est fait appel à la commande grep pour supprimer une ligne indiquant le total, alors que **awk** sait parfaitement procéder à la sélection des enregistrement sur lesquels le traitement doit porter en spécifiant une expression rationnelle qui devra être en concordance (**~**) ou non (**!~**) avec l'enregistrement entier (si rien n'est spécicifé) ou un champ particulier :
ls -l | awk '!/total/ { print "La taille du fichier",$9,"est",$5," octets."}'
ls -al | awk '$9 ~ /[a-w]+/ { print "La taille du fichier",$9,"est",$5," octets."}'
ls -al | awk '! /total/ && $9 !~ /^\./ { print "La taille du fichier",$9,"est",$5," octets."}'

La sélection de l'enregistrement peut également être conditionnée au test d'une variable interne à awk :
ls -l | awk 'NR==2 { print "La taille du fichier",$9,"est",$5," octets."}'
ls -l | awk 'NR==2 || NR==4 { print "La taille du fichier",$9,"est",$5," octets."}'
ls -l | awk 'NR==2, NR==4 { printf("La taille du fichier %s est %s octets.\n",$9, $5)}'

Il est également possible de faire exécuter un traitement particulier avant et après le traitement du flux via l'utilisation des mots clés **BEGIN** et **END** :

**ls -l | awk 'BEGIN {print "Les fichiers correspondants :"} **
**NR==2 || NR==4 { print "La taille du fichier",$9,"est",$5," octets."} **
END {print "============= FIN ================="}'

En définitive, lorsque le traitement commence à se complexifier de la sorte, mieux vaut le coder dans un fichier de script à part qui sera appelé via l'option **-f** lors de l'invocation de awk :
ls -l | awk -f ls.awk avec le fichier ls.awk contenant :

```
# Section BEGIN  => Appelée avant le traitement du flux
BEGIN {
 print "Les fichiers correspondants :"
}
# Un section traitant le flux
NR==2 || NR==4 {
 printf("La taille du fichier %s est %d octets\n",$9,$5)
}
# Section END invoquée après la lecture du flux de données
END {
 print "============= FIN ================="
}
```

> **Note : awk** permet de définir plusieurs sections de traitement pour le flux, lesquelles seront (conditionnellement) invoquées séquentiellement.

Ce survol de **awk** s'arrêtera là, mais pas les possibilités de cet incontournable outil, avec lequel il est possible de définir des variables de travail, des structures de contrôle (tests, boucles), de rediriger les entrées-sorties, d'appeler des fonctions prédéfinies ou créer ses propres fonctions...

6. Autres langages de scripts (rapide survol)

Bien entendu, le shell, s'il est probablement le précurseur, n'est pas le seul moyen d'ecrire des scripts permettant d'automatiser des taches d'administration système. De nombreux autres langages permettent, avec plus ou moins de bonheur et de facilité, d'y procéder.

6.1. TCL

Le langage TCL *Tool Command Language* a été écrit par John Ousterhout et son équipe à l'université de Berkeley à partir 1988. Il est conçu pour être facile à apprendre (il repose uniquement sur douze règles syntaxiques), extensible et facilement interfaçable avec le langage C (embarquable). Ces qualités ainsi que sa maturité en ont fait un langage très utilisé en administration système, et notamment avec l'utilisation de son extension TK qui permet de générer des interfaces graphiques.

Un exemple de code :

```
set message "Bonjour, Monde"
puts $message
```

6.2. PERL

Le langage PERL *Practical Extraction and Report Language*, à été l'origine développé Larry Wall, et est né si l'on en croit wikipédia : « du besoin de disposer d'un langage optimisé pour l'extraction d'informations de fichiers textes et la génération de rapports. Avant la naissance de Perl, les traitements sur le texte devaient être faits au moyen de scripts shell, en utilisant les programmes sed, awk, grep, cut, test et expr. Beaucoup de limites apparaissaient quand on utilisait cette programmation : format des données d'entrée pas toujours souple, difficulté de passer des données d'un processus à l'autre, lenteur due au lancement de multiples programmes (le coût du lancement d'un processus n'était pas négligeable), dépendance à une mise en oeuvre particulière d'une commande, bogues intrinsèques à certains langages (awk ne différencie pas la comparaison de nombres et de chaînes de caractères). Perl regroupe et emprunte sa syntaxe concrète à tous ces mini langages, dont le shell, en ajoutant une partie de la syntaxe du C et les fonctions des bibliothèques système en C. » De fait, perl a connu un très grand succès, notamment auprès des administrateurs système qui l'ont grandement utilisé pour écrire nombre d'utilitaires (à l'origine de DEBIAN, les outils d'installation étaient écrits en perl).

L'une de ses grandes forces réside dans l'utilisation massive des expressions rationnelles mais il est desservi par une syntaxe absconse, due en grande partie à sa grande richesse, et qui le rend rapidement illisible pour un non-spécialiste.

Pour preuve l'une des devises favorites de ses utilisateurs est « There Is More Than One Way To Do It (TIMTOWTDI) », ce qui pourrait se traduire par *Il y a plus d'une façon de le faire !*

Malgré ce, un très grand nombre de bibliothèques sont disponibles pour PERL via des dépôts de code spécialisés : les CPAN.

Un exemple de code :

```
$message = "Bonjour, Monde.\n";
print $message;
$message =~ s/Monde/Université de Perpignan Via Domitia/;
print $message;
exit 0
```

6.3. PYTHON

PYTHON a été développé par Guido van Rossum en 1989 avec pour but avoué d'obtenir un langage lisible, qui vise à être visuellement épuré, et utilise des mots anglais fréquemment là où d'autres langages utilisent de la ponctuation et sa structuration s'obtient par l'indentation du code source, et il est résolument orienté objet.

Il est actuellement très utilisé pour du prototypage et des scripts, d'autant qu'il dispose d'une très importante bibliothèque de modules se rapportant à d'innombrables domaines, comprenant notamment des *wrappers* pour la quasi-totalité des librairies les plus en vogue (qt, gtk, etc...).

Un exemple de code :

```
print "Bonjour, Monde !"
print "Bonjour, Monde !".replace("Monde", "Université de Perpignan Via Domitia")
```

6.4. RUBY

RUBY a été à l'origine conçu par Yukihiro "Matz" Matsumoto qui, ne trouvant pas dans les langages de programmation déjà existants (dont Perl et Python) de quoi le satisfaire, en commença l'écriture en 1993

et publia une première version en 1995. Ce nouveau langage est entièrement basé sur le paradygme objet : tout y est objet, y compris les types dits "de base" dans les autres langages.

Un exemple de code :

```
puts "Bonjour, Monde !"
puts "Bonjour, Monde !".upcase
```

6.5. ... et les autres

On n'aura cité ici que les langages de scripts les plus connus ou utilisés, mais il en existe une multitude d'autres, chacun plus ou moins spécialisés dans un domaine particulier, tels que LUA (forte ressemblance au C, facilement embarquable), PHP (développé originellement pour le développement web, mais l'interpréteur en ligne de commande permet d'écrire des scripts à vocation "système" utilisant l'immense bibliothèque de modules existante), etc...

En guise de conclusion, il reste évident qu'au vu de la multitude des possibilités offertes par le monde open source, il est possible de trouver un langage qui corresponde à chaque besoin particulier ou aux capacités du programmeur !

Bons scripts !!!

7. ... Et Microsoft Windows™ dans tout ça ?

Les systèmes d'exploitation vendus (loués ???) par Microsoft™ comportent, depuis les premières versions de Microsoft Windows™ une certaine compatibilité ascendante avec la génération précédente, à savoir PC|MS|DR/DOS™ laquelle permettait de créer des traitements batchs (scripts). Au fur et à mesure de l'évolution de ses systèmes d'exploitation, Microsoft™ semble vouloir permettre aux administrateurs gérant ses systèmes d'exercer leurs tâches de plus en plus via des scripts : pour preuve tous les outils permettant l'accès ou la configuration d'Active Directory™ existent en version "ligne de commande"...

Un indice supplémentaire est l'effort de développement fourni pour l'interpréteur de commande des systèmes récents, à savoir cmd.exe, qui intègre l'historisation activée par défaut, et (à compter de Microsoft Windows XP™ la possibilité de complétion des commandes.

7.1. Au début, (PC|MS|DR)/DOS™ et command.com

L'originel PC-DOS™ commandé par IBM™ à Microsoft™, provenait en fait du développement de Q-DOS™ racheté à Tim Paterson, lui-même grandement inspiré de CPM™, et disposait pour toute interface utilisateur de l'ineffable shell dénommé **command.com**. A l'instar des shells Unix, il comportait des commandes internes et permettait de chainer des commandes externes (dans son propre espace mémoire, évidemment).

Ce command.com permettait également d'interpréter des fichiers de scripts (nommés fichiers batch) portant l'extension **.BAT** afin de réaliser des tâches d'administration du système, ou de préparer l'environnement de travail et lancer des applications. Ce shell permettait également la redirection vers des fichiers.

D'un point de vue programmation, les fichiers .BAT permettaient de re-router le flux d'instruction via des aiguillages **GOTO etiquette** et des tests sur le code de retour de commandes **ERRORLEVEL** via la commande interne **IF [NOT]**. La boucle **FOR IN ... DO** était également disponible pour procéder à des intérations sur des noms de fichiers.

7.2. Premier tournant : Windows NT4™

Avec l'avênement de Windows NT4™, le shell command.com connait une évolution majeure, et devient **cmd.exe** qui ouvre un "terminal en mode texte" dans l'interface graphique. Du point de vue intrinsèque de sa programmation, il ne connait guère d'évolutions par par rapport à la génération précédente, mais il est adapté au contexte multi-processus, et à l'environnement graphique.

Par contre, Microsoft™ fournit d'autres interpréteurs, plus riches en fonctionnalités, puisque directement issus de leurs outils de développement phares : VBScript™ et surtout l'infrastructure Window Script Host™ qui se veut plus une plateforme de script qu'un simple langage car, en effet, différents langages frontaux lui sont disponibles (*Wscript* pour le mode fenêtré, *Cscript* pour le mode console texte, ou encore VBScript™, *Javascript* ou Active Perl™).

Enfin, les dernières briques apportées par Microsoft™ à son système de scripting est l'infracstructure *Windows Management Instrumentation* ou *WMI* qui permet un accès à la quasi-totalité des objets du système et *Active Directory Service Interface* ou *ADSI* plus spécialisé pour la gestion d'Active Directory (bien que permettant également l'accès aux données des bases SAM de NT4 ou 2000, ou encore aux annuaires d'Exchange™) sont représentés dans le système de scripts comme de nouveaux objets.

7.3. Et après...

L'évolution vers la ligne de commande se poursuit encore avec le dernier des systèmes d'exploitation Microsoft™ : Windows 2008 Server™, qu'il est possible de déployer en *"Mode Core"* (c'est à dire sans interface graphique), et dans lequel le shell Cmd.exe, s'il existe toujours pour les raisons de

compatibilité, se voit mis en concurrence avec un nouvel environnement de ligne de commande : Windows Power Shell™ qui, si l'on en croit les fichiers de déploiement fourni, ne travaille plus sur des données textuelles mais sur des objets de la platforme et dont les commandes externes sont appelées applet de commande.NET™.

Notes

1. D'aucuns pourraient même considérer qu'il s'agit là d'une pure perte de ressources (en temps processeur et occupation mémoire)...
2. Principal responsable des publications de l'Unix Berkeley (donc de ce qui deviendra la famille des *BSD) et co-fondateur de Sun MicroSystems™.
3. Une commande ne fait qu'une seule chose... mais le fait bien !
4. Or, sous Unix, tout est fichier ...