Notions de shell

Xavier Serpaggi

Mastère Spécialisé en Génie Logiciel

Table des matières

Chapitre 1

Objectifs

L'objectif de ce cours est que vous soyez capable d'écrire des scripts shell pour résoudre vos problèmes d'administrateurs au jour le jour.

Le chapitre 12 de l'excellent ouvrage *"Unix Text Processing"* a un titre qui pourrait être un résumé de ce que l'on désire obtenir ici : *Let the Computer Do the Dirty Work.*

En voici le début, qui explique ce que cela signifie :

> Let the Computer Do the Dirty Work
>
> Computers are very good at doing the same thing repeatedly, or doing a series of very similar things one after another. These are just the kinds of things that people hate to do, so it makes sense to learn how to let the computer do the dirty work.

Ou, comment faire faire à l'ordinateur les tâches ingrates.

Pour arriver à ce but, vous devez être capables de comprendre les scripts que d'autres personnes ont écrit et vous devez également écrire des scripts compréhensibles et réutilisables (par vous et par d'autres).

En tant qu'administrateur système, on se rend compte que l'on refait très souvent les mêmes actions : analyser un fichier de *log*, déplacer un ensemble de fichiers d'un endroit à un autre, supprimer un ensemble de fichiers répartis dans toute l'arborescence, ...Les scripts doivent donc être, comme tout programme, documentés et facilement améliorables. Programmer en shell demande peut-être plus de rigueur que l'utilisation d'un autre langage du fait de l'absence de type, de la non nécessité des déclarations, ...

Pour atteindre ces objectifs, il est nécessaire de connaître la syntaxe du shell lui même, mais également les commandes du système.

L'utilisation de scripts shell est omniprésente dans le monde Unix : une immense partie des mécanismes de démarrage du système et de sa configuration sont réalisés par ce biais.

TAB. 1.1 – Conventions de notations

Notation	Signification
`ls`	une commande dont le nom est `ls`
`mkdir(1)`	une commande dont le nom est `mkdir` et dont le manuel est en section 1 des pages de manuel.
ll	une commande que nous avons définie dont le nom est `ll`
cd	une commande interne dont le nom est `cd`
`/etc/passwd`	un fichier dont le nom est `/etc/passwd`
`HOME` ou `$HOME`	une variable de nom `HOME`
`-o toto`	l'option (`-o`) d'une commande et son argument (`toto`)
[Entrée]	représente une touche du clavier, ici la touche `Entrée`
`^C`	une séquence de touche **[Contrôle]**+**[C]**
`<argument>`	désigne un argument obligatoire
`[option]`	désigne un argument ou une partie optionnelle

1.1 Pré-requis

Avoir des notions sur l'organisation des fichiers sur un système Unix et savoir ce que sont les droits sur les fichiers.

Les notations trouvées dans ce document sont présentées dans la table 1.1.

Chapitre 2

Présentation

2.1 Origines

Shell est un mot anglais qui signifie coquille. Ce nom n'a pas été choisi au hasard. Le shell est la coquille qui entoure le noyau du système d'exploitation (généralement un unix). C'est par lui que l'utilisateur passe pour commander son système. C'est donc l'interface privilégiée entre l'utilisateur et le système.

2.2 Principe

Un shell est en fait un interprète[1]. Ces commandes peuvent être internes au shell ou externes. Ces dernières seront alors des commandes du système ou des commandes que l'utilisateur a pu écrire lui même (un script shell par exemple).

De fait, le travail du shell est de gérer la façon dont vont s'exécuter ces commandes puis d'en connaître le statut pour pouvoir en assurer une gestion minimale (leur fonctionnement a-t-il été celui prévu ? S'il y a eu des soucis pendant l'exécution, de quelle nature sont-ils ? ...) Il a donc un rôle de gestion de processus.

2.3 Différents Shells

Le Shell original des systèmes unix est `sh`. Au début, c'était le Thompson Shell, puis il a été remplacé par le Bourne Shell mais a conservé le même nom. Il est toujours utilisé actuellement, de concert avec d'autres interprètes :

– `ksh`, le Korn Shell (de M. Korn) ;

[1] Certaines personnes utilisent l'anglicisme *interpréteur* de commandes

– `csh`, le C Shell, nommé ainsi car sa syntaxe ressemble à celle du langage C ;
– `bash`, le *Bourne again Shell*, qui est une évolution de `bsh` et qui est trés employé actuellement.

Chacun a des spécificités que l'on retrouve au niveau de la façon de les programmer ou au niveau de l'interface utilisateur (complétion, correction d'erreurs, nom et manipulation des variables, ...) – on peut voir une comparaison des fonctionnalités de certains Shells sur Wikipédia. Dans tous les cas, le rôle qu'ils jouent est le même, celui de permettre à l'utilisateur de passer des commandes au système d'exploitation. Il peut-être plus pratique pour telle ou telle tâche d'utiliser un Shell en particulier, comme il peut-être préférable dans certaines circonstances d'utiliser un langage de programmation plutôt qu'un autre. Pour ce faire, il est bon de les connaître. Ici nous allons plutôt nous attacher à la description de `bash` qui contient les bases qui vous permettront de facilement appréhender les autres interprètes.

Chapitre 3

Bases d'utilisation

Pour pouvoir utiliser un shell, il faut disposer d'un terminal, qu'il soit physique ou virtuel. Par exemple, des terminaux virtuels couramment utilisés sont `xterm`, `rxvt` ou `gnome-terminal`.

Le terminal est un support au shell, une interface entre le shell et l'utilisateur.

Une fois un terminal lancé, on fait en général face à une invite de commande (on utilisera plus facilement le terme anglais *prompt*) qui peut ressembler à cela :

```
[P]:~>
```

[P] est ici pour vous rappeler que c'est un *prompt*, le **:** est un séparateur et **~** représente le répertoire dans lequel on est (ici, le répertoire racine de l'utilisateur – le *home dir*). Bien entendu, ceci n'est qu'un exemple et varie en fonction de votre système, de votre shell et de bien d'autres paramètres.

Le *prompt*, comme son nom l'indique, invite l'utilisateur à entrer une commande. Et, par exemple, la première commande que l'on peut utiliser est celle nous permettant de savoir ce qui se trouve dans le répertoire courant : `ls`.

```
[P]:~> ls
Desktop        Documents         Enseignements
Mail
[P]:~>
```

3.1 Syntaxe

Chaque ligne entrée par l'utilisateur est analysée, découpée en morceaux et exécutée. De ce fait, les commandes passées au Shell doivent respecter une certaine syntaxe :

- Le séparateur de mot est l'espace ou tout ce qui peut-être considéré comme tel (séquence d'espaces, tabulations) ;
- les commandes sont séparées soit par un retour à la ligne, soit par le caractère **;** ;
- une commande s'écrit sous la forme `commande options arguments`
- toute ligne ou partie de ligne commençant par **#** est ignorée (c'est un commentaire) ;
- chaque commande a un code de retour qui n'est pas affiché (nous verrons plus loin comment l'obtenir) et qui est 0 si tout s'est bien passé, non nul dans le cas contraire.

3.2 Options

Les options sont actuellement présentes sous deux formes, longues ou courtes. Jusqu'à récemment seules les options courtes, de la forme `-x` existaient, où `x` est une lettre ou un chiffre représentant le nom de l'option, choisi par le créateur de la commande.

Devant, d'une part le manque de lettres/chiffres et, d'autre part le côté relativement obscur du choix des lettres/chiffres pour certaines options, les versions longues sont apparues. Ces dernières sont sous la forme `--nom-de-l-option-longue`. En général, à une option courte correspond une options longue, l'inverse n'étant pas toujours vérifié.

Les noms de options ainsi que la manière de les utiliser sont en général détaillés dans le manuel de la commande.

Il est possible de rencontrer également des options courtes avec une syntaxe longue, de la forme `-option`. C'est le cas, par exemple, du programme gcc :

```
[P]:~> gcc -Wall -ansi -pedantic -o prog prog.c
[P]:~>
```

Les options `-Wall`, `-ansi` et `-pedantic` sont considérées comme des options courtes au même titre que `-o`. Cette dernière par contre, admet un argument, qui est ici **prog**, le nom du fichier que va créer `gcc`.

3.3 Exécution conditionnelle

Le dernier point de la syntaxe fait que l'on peut séparer plusieurs commandes avec les opérateurs booléens **&&** ou **||** pour agir de manière conditionnelle. En effet, il est possible d'écrire une séquence de commandes qui sera évaluée de gauche à droite et dont chaque

morceau ne s'exécutera que si le précédent a bien fonctionné ou pas.

Par exemple, nous voulons créer un répertoire tmp et, si la création a été effective, y copier des fichiers. Nous utiliserons pour cela le **&&** (ET logique) dont la syntaxe générale est
`commande1 && commande2 && ... && commandeN`
(fin de l'exécution dès qu'une commande a un code de retour non nul).

En pratique :

```
[P]:~> mkdir /home/dupont && cp /etc/skel/.* /home/dupont
[P]:~>
```

Si la commande `mkdir` a échouée pour une raison ou pour une autre, alors la commande `cp` ne sera pas exécutée.

L'utilisation de `||` (OU logique) est semblable, mais le comportement est inversé. Dans une suite de commandes
`commande1 || commande2 || ... || commandeN`
(fin de l'exécution dès qu'une commande a un code de retour égal à 0), le traitement s'interrompt dès qu'une commande a été exécutée avec succès. Par exemple, nous désirons créer un répertoire et, s'il existe déjà, afficher un petit message d'erreur :

```
[P]:~> mkdir tmp || echo "Attention, le répertoire tmp existe déjà !"
[P]:~>
```

Notez que cet exemple est loin d'être parfait, la commande `mkdir` affichant déjà un message d'erreur dans ce cas.

Chapitre 4

Entrées / Sorties

Traditionnellement, l'utilisateur entre les commandes via le clavier et observe le résultat sur l'écran. Même si dans la plupart des cas ce comportement est celui souhaité, il est parfois nécessaire de redéfinir le flux d'entrée et/ou le flux de sortie.

Les Shells permettent de faire cela de manière relativement simple, comme nous allons le voir ci-dessous.

Quel que soit le Shell utilisé il existe en fait deux types de sortie qui sont différenciées par un identifiant. En général, elles sont toutes les deux visualisées via l'écran du terminal. Ces deux sorties sont :
– `1` la sortie standard ;
– `2` la sortie d'erreur.
Il existe plusieurs types de redirections :

– `>` et `<` qui redéfinissent respectivement la sortie et l'entrée standard (si le fichier existe il est remplacé) ;
– `>>` qui redéfinit la sortie standard, mais qui ajoutent le nouveau contenu à la suite d'un éventuel contenu existant ;
– `&>`, `>&` et `<&` qui permettent de dupliquer les flux d'entrée et de sortie standard (ces redirections seront abordées au chapitre 11) ;
– `<<` qui permet de passer plusieurs données à une commande et qui est également abordée au chapitre 11.

Avant de nous intéresser aux redirections proprement dites, examinons le processus de *piping*.

4.1 Enchaîner les commandes

Nous avons vu dans une section précédente que les Shells sont capables de traiter plusieurs commandes les unes à la suite des autres de manière conditionnelle. Ils sont également capables d'enchaîner les commandes en faisant en sorte que la sortie de l'une soit l'entrée de l'autre. C'est cela que l'on nome *piping*, du terme anglais *pipe* qui signifie tube. C'est un tube de communication entre les différentes commandes. Pour l'utiliser, il suffit de séparer les commandes que l'on désire cascader avec le symbole `|`. Il faut bien entendu que ces commandes soient capables de réagir à ce traitement.

La syntaxe générale est donc :

`commande1 | commande2`

Les commandes sont toujours évaluées de gauche à droite et les unes à la suite des autres (la $n^{ième}$ commande ne s'exécutera que quand la $n-1^{ième}$ aura terminé son exécution).

Par exemple, si l'on désire compter le nombre de fichiers présents dans un répertoire, on peut utiliser la commande `wc` (word count) pour compter le nombre de lignes que retourne la commande `ls` :

```
[P]:~> ls | wc -l
12
[P]:~>
```

L'exemple ci-dessus va compter (`wc`) le nombre de lignes (`-l`) que produit (`|`) la sortie de la commande `ls`.

Il est bien entendu possible de cascader plus de deux commandes et si, par exemple, nous recherchons, dans notre répertoire personnel de l'utilisateur `dupont`, combien il y a de fichiers portant l'extension `.txt` qui contiennent le mot `rapport` dans leur chemin complet, on peut utiliser la séquence suivante :

```
[P]:~> find /home/dupont -name "*.txt" -print | grep -i rapport | wc -l
9
[P]:~>
```

Il y en a 9 !

4.2 Rediriger la sortie

La redirection de la sortie standard permet de ne plus afficher le résultat de la commande sur le terminal, mais, par exemple dans un fichier.

La syntaxe est la suivante :

`commande > fichier`

où `commande` est un exécutable (commande système, commande interne du Shell ou même script Shell) et `fichier`, un flux sur lequel on dispose de droits d'écriture. Nous remarquons que `fichier` peut tout à fait être un flux particulier tel que `/dev/null`, ce qui permet, en fait, d'ignorer la sortie de `commande`.

Si `commande` est un enchaînement de commandes (*piping*), c'est bien entendu le résultat de la dernière commande de la séquence qui sera redirigé.

4.2.1 Redirections simples

L'exemple suivant montre comment il est possible de sauvegarder la liste des documents d'un répertoire dans un fichier texte :

```
[P]:/tmp> ls
com.sun.swp.client.LOCK  hsperfdata_serpaggi       ogl_select882
[P]:/tmp> ls > /home/serpaggi/tmp.txt
[P]:/tmp> cd /home/serpaggi
[P]:/tmp> ls
Desktop        Documents         Enseignements
Mail           tmp.txt
[P]:/tmp> cat tmp.txt
com.sun.swp.client.LOCK
hsperfdata_serpaggi
ogl_select882
[P]:/tmp>
```

La commande `ls` permet d'afficher le contenu d'un répertoire, comme nous l'avons vu au tout début de ce document et la commande `cat` permet d'écrire le contenu d'un fichier sur le terminal courant.

Comme cela a déjà été précisé dans l'introduction, si le fichier existe déjà, il est écrasé, c'est à dire qu'il est remplacé et que l'ancien contenu est perdu. Il existe cependant un moyen pour écrire à la suite de ce qui existe déjà dans fichier, comme nous allons le voir ci-dessous.

4.2.2 Redirections avec ajouts

Pour ajouter la sortie d'une commande à un fichier existant, il faut utiliser `>>` en lieu et place de `>`. De la même manière qu'avec la redirection simple, si le fichier destination n'existe pas, il sera créé.

Nous pouvons par exemple créer la liste des fichiers présents dans tous les répertoires temporaires du système :

```
[P]:~> ls /tmp >> tmp.lst
[P]:~> ls /var/tmp >> tmp.lst
[P]:~> ls ~/tmp >> tmp.lst
[P]:~> cat tmp.lst
...
...
...
[P]:~>
```

4.2.3 Redirections multiples

Dans l'exemple ci-dessus, nous avons fait l'hypothèse qu'un répertoire `tmp` était présent dans notre répertoire racine (`~`). Cependant, si ce n'est pas le cas, la commande `ls` va retourner une erreur :

```
[P]:~> ls ~/tmp >> tmp.lst
/bin/ls: tmp: Aucun fichier ou répertoire de ce type
[P]:~>
```

Dans le cadre, par exemple, d'un script qui va s'exécuter automatiquement à intervalle régulier, il peut-être gênant de voir de tels messages d'erreur (ceci est d'autant plus gênant que l'on sait très bien que cela peut arriver et que cette "erreur" est volontaire).

Il est possible de "supprimer" ce message d'erreur en utilisant les redirections multiples. Mais cela ne fonctionne que parce que le programme `ls` a été programmé de telle sorte que les messages d'erreur qu'il produit soient dirigés non pas vers la sortie standard (1), mais vers la sortie d'erreur (2). De ce fait, il est possible de rediriger cette sortie d'erreur vers un fichier et la sortie standard vers un autre fichier. Par exemple :

```
[P]:~> ls ~/tmp >> tmp.lst
/bin/ls: tmp: Aucun fichier ou répertoire de ce type
[P]:~> ls ~/tmp >> tmp.lst 2> ~/error_log
[P]:~> ls
Desktop        Documents         Enseignements
Mail           error_log         tmp.lst
tmp.txt
[P]:~> cat ~/error_log
...
/bin/ls: tmp: Aucun fichier ou répertoire de ce type
[P]:~>
```

Ou bien, pour ignorer complètement le message d'erreur :

```
[P]:~> rm error_log
[P]:~> ls
Desktop         Documents          Enseignements
Mail            tmp.lst            tmp.txt
[P]:~> ls ~/tmp >> tmp.lst 2> /dev/null
[P]:~> ls
Desktop         Documents          Enseignements
Mail            tmp.lst            tmp.txt
[P]:~>
```

L'exemple de création d'un répertoire vu plus haut n'était en fait pas parfait. En effet, la commande `mkdir` écrit un message d'erreur si le répertoire existe déjà. Il est possible, en utilisant les redirection de supprimer le message par défaut et de mettre le notre à la place :

```
[P]:~> mkdir tmp 2>/dev/null || echo "Attention, le répertoire existe déjà !"
[P]:~>
```

4.2.4 Limitations des redirections

Le Shell a un mécanisme permettant de limiter certaines actions – nous aborderons ce point plus tard. Il est possible via ce mécanisme d'empêcher l'utilisateur d'écraser des fichiers avec les redirections, en général pour lui éviter une mésaventure.

Cependant, il se peut que l'utilisateur sache ce qu'il fait et que ce soit volontairement qu'il décide d'écraser un fichier. Il faut donc qu'il puisse passer outre cette limitation et la redirection spéciale `>|` est là pour ça. L'exemple ci-dessous montre comment l'utiliser :

```
[P]:~> ls > toto
[P]:~> ls > toto
[P]:~> set -C # mise en place de la protection contre l'écrasement
[P]:~> ls > toto
-bash: toto : impossible de ré-écrire sur le fichier existant
[P]:~> ls >| toto
[P]:~>
```

Nous verrons plus tard comment obtenir des comportements encore plus évolués avec les redirections.

4.3 Rediriger l'entrée

Alors qu'il semble naturel de vouloir rediriger la sortie d'une commande (besoin de conserver le résultat ou au contraire de l'ignorer complètement), il peut paraître moins naturel de vouloir en rediriger l'entrée.

Habituellement, l'entrée, c'est l'utilisateur, via le clavier. Cependant certains programmes traitant des données ne savent pas lire des fichiers et considère que tout ce qu'ils doivent traiter arrive sur l'entrée standard. Imaginons que `compute` soit un tel programme. Il est alors impossible d'écrire

```
[P]:~> compute data.big
[P]:~>
```

le programme sera incapable de traiter les données dans le fichier `data.big`.

Au lieu de lancer le programme puis d'entrer le contenu de data.big à la main, ligne après ligne, on peut utiliser le mécanisme des redirections et plus précisément la redirection d'entrée `<` :

```
[P]:~> compute < data.big
Computing...
305.17 -29.45 450.92
124.18 -89.41 434.52
84.21 59.99 598.49
173.50 -3.06 675.48
65.73 38.18 304.96
149.04 -82.64 621.83
229.56 31.31 695.99
219.28 -56.76 347.12
118.22 41.54 627.16
331.53 -53.60 740.44
92.62 27.57 684.76
93.78 -74.22 532.45
OK!
[P]:~>
```

Dans ce cas, le Shell considère que l'entrée standard n'est plus le clavier, mais le fichier `data.big`. Notre programme `compute` peut alors faire son travail.

Dans l'exemple ci-dessus, le programme `compute` retourne le résultat de ses traitements sur l'écran du terminal. Il peut-être intéressant de sauvegarder ces résultats dans un fichier. Il suffit pour cela, comme nous l'avons vu dans la section précédente, de rediriger la sortie standard (on considère que les textes `Computing...` et `OK !` sont écrits sur la sortie d'erreur) :

```
[P]:~> compute < data.big > result.big
Computing...
OK!
[P]:~> cat result.big
305.17 -29.45 450.92
124.18 -89.41 434.52
84.21 59.99 598.49
173.50 -3.06 675.48
65.73 38.18 304.96
149.04 -82.64 621.83
229.56 31.31 695.99
219.28 -56.76 347.12
118.22 41.54 627.16
331.53 -53.60 740.44
92.62 27.57 684.76
93.78 -74.22 532.45
[P]:~>
```

Chapitre 5

Variables

Les variables sont aussi souvent appelées variables d'environnement du fait qu'elles permettent, dans certaines limites, de modifier l'environnement d'exécution des programmes et donc le comportement de ceux-ci.

Même si leurs comportements sont très entrelacés, il est possible de diviser, de manière informelle, ces variables en trois catégories :

– celles qui sont relatives au système ;
– celles qui permettent de modifier le comportement de certaines applications ;
– celles définies par les utilisateurs pour leurs usages propres.

Les variables sont en général globales, ne sont en général pas typées et il n'est pas nécessaire de les définir avant de les utiliser.

5.1 Déclaration

La syntaxe pour définir une variable est des plus simples :

```
VAR=valeur
```

La définition d'une variable se fait en lui affectant une valeur et il est important de noter qu'il ne faut pas ajouter d'espace autour du signe `=`.

En général, les noms des variables sont en majuscule, mais ce n'est pas une obligation. Si `valeur` est une chaîne de caractère comportant des caractères spéciaux, il est nécessaire, soit de les échapper en les faisant précéder d'un \, soit de mettre la totalité de la chaîne entre guillemets :

```
PEER=Jean\ Dupont
RESERVED="?()&:[]"
```

Dans le cas de la variable `RESERVED`, les guillemets doubles (`"`) ne font pas partie du contenu, mais ne sont là que pour marquer la limite et protéger ce contenu.

5.2 Utilisation

Pour utiliser une variable, il faut faire précéder son nom du caractère `$`. Par exemple :

```
[P]:~> PAGER=/bin/less
[P]:~> echo $PAGER
/bin/less
[P]:~> IDENTITY="Xavier Serpaggi"
[P]:~> echo ${IDENTITY}
Xavier Serpaggi
[P]:~> if [ "${IDENTITY}" == "Gérard Lambert" ]
> then
> ACCESS=1
> else
> ACCESS=0
> fi
[P]:~> echo $ACCESS
0
[P]:~>
```

Les accolades ({}) ne sont pas nécessaires dans la majorité des cas ; le shell arrive le plus souvent à se débrouiller pour ne pas faire de confusion. Cependant, pour des raisons de facilité de lecture et de non-ambiguïté, il est souhaitable de les utiliser.

5.2.1 Variables et guillemets

Lors de **l'utilisation** des variables, on peut avoir besoin de les mettre entre guillemets, qu'ils soient simples (`'$VAR'`) ou doubles (`"$VAR"`).

En général, ceci est fait pour "protéger" des caractères réservés dans le contenu de la variable, comme nous l'avons vu précédemment. Cependant, il est important de noter que ces deux types de guillemets n'ont pas la même signification et ne donnent pas le même résultat.

Les guillemets simples (*simple quotes*) font que les variables contenues à l'intérieur ne sont pas évaluées. Par exemple :

```
[P]:~> MAVAR="plic ploc"
[P]:~> echo $MAVAR
plic ploc
[P]:~> echo "$MAVAR"
plic ploc
[P]:~> echo '$MAVAR'
$MAVAR
[P]:~>
```

De cet exemple, on comprend facilement l'intérêt des guillemets simples.

L'exemple suivant illustre l'intérêt de l'utilisation de guillemets doubles :

```
[P]:~> MAVAR="plic ploc"
[P]:~> for i in $MAVAR ; do echo $i ; done
plic
ploc
[P]:~> for i in "$MAVAR" ; do echo $i ; done
plic ploc
[P]:~> for i in '$MAVAR' ; do echo $i ; done
$MAVAR
[P]:~> unset $MAVAR
[P]:~>
```

Même si vous n'avez pas encore abordé la syntaxe de la boucle **`for`** (voir section 8.1.3 pour une description) , vous ne devriez pas avoir de mal à comprendre ce que cela fait : pour chaque `$i` prenant ses valeurs dans ce qui contient `$MAVAR`, afficher la valeur de `$i`.

La présence de guillemets double permet de considérer le blanc comme étant partie intégrante de la valeur de la variable et non pas comme étant un séparateur de mots.

Il n'y a donc pas de bonne ou de mauvaise façon d'utiliser les guillemets simples ou doubles. Il faut simplement adapter cette utilisation au contexte.

5.3 Portée

Les variables ont une portée, c'est à dire qu'elles n'existent que dans le shell dans lequel elles ont-été définies.

Ceci a une implication importante puisque, chaque processus lancé par un shell est en fait exécuté dans un sous-shell. Les variables ne sont donc pas "transmises".

Pourtant, il existe une mécanisme permettant d'étendre la portée de ces variables pour qu'elles soient connues des sous-shells. C'est le rôle de la commande interne *`export`* . Elle permet d'exporter les variables aux sous-shells.

```
[P]:~> PAGER="/bin/cat"
[P]:~> echo $PAGER
/bin/cat
[P]:~> bash
[P]:~> echo $PAGER
[P]:~> exit
exit
[P]:~> echo $PAGER
/bin/cat
[P]:~> export $PAGER
[P]:~> bash
[P]:~> echo $PAGER
/bin/cat
[P]:~> PAGER="/bin/less"
[P]:~> exit
exit
[P]:~> echo $PAGER
/bin/cat
[P]:~> unset $PAGER
[P]:~>
```

Lorsqu'on lance un shell dans un shell, on a ce que l'on appelle un sous-shell. Dans le premier cas, ce sous-shell n'a pas connaissance de la variable `$PAGER`. Par contre, une fois exporté dans le premier shell, le second en a connaissance.

La troisième partie permet de se rendre compte que l'export ne fait que "descendre" l'information vers les sous-shells et en aucun cas elle ne la "remonte" vers les shells parents.

Nous aborderons une autre vision de la portée des variables dans la section 6, décrivant l'utilisation des fonctions.

5.4 Variables système

5.4.1 Non modifiables

Les variables ci-dessous ne devraient pas être modifiées par l'utilisateur car elles sont utilisées par un ensemble de programmes.

Elles ont donc, pour nous, un caractère informatif.
- `HOME` donne le chemin absolu du répertoire utilisateur (celui de l'utilisateur qui exécute ce shell) ;
- `PWD` représente le chemin absolu du répertoire courant. C'est la même chose que la sortie de la commande `pwd` ;
- `SHLVL` représente le niveau (*level* en anglais) d'imbrication de shells.

5.4.2 Modifiables

Les variables ci-dessous sont définissables et modifiables par l'utilisateur. Il est même parfois nécessaire de les définir correctement pour que les programmes fonctionnent comme attendu.

- `TERM` permet de définir le type de terminal que l'utilisateur utilise. Cela permet en fait de définir les capacités du terminal en terme d'affichage (couleurs, nombre de lignes, de colonnes, possibilité de faire du défilement avant et/ou arrière, ...) ;
- `LANG` (et toute la famille `LC_`) permet de définir la façon dont les programmes vont afficher tout ce qui est localisable (c'est à dire tout ce qui peut changer quand on parle une langue ou une autre). Cela comprend, par exemple, la façon d'afficher les devises, l'heure, ...Il existe plusieurs variables pour chacune des parties localisable qui peuvent être définies séparément (voir la page de manuel de **`localedef(1)`**).

5.5 Variables propres aux programmes

Par définition, elles sont utilisées par les différents programmes qui déclarent les utiliser et donc, les définissent. En général, c'est en lisant la page de manuel d'une application particulière que l'on apprend quelles sont les variables qu'il est possible d'utiliser pour en modifier le comportement. Nous avons vu dans la section 5.2 comment la variable `PAGER` modifie le comportement de la commande **`man`**. Une autre illustration serait, par exemple, le programme **`less`**, qui permet d'afficher un fichier sous forme de texte dans un terminal, et qui utilise les valeurs de plusieurs variables si ces dernières ont été définies :

- `LESSCHARSET` pour définir un charset à utiliser pour afficher les fichiers ;
- `LESS` qui contient les options que l'on désire passer à l'application à chaque fois qu'elle est invoquée (donc, sans avoir à les retaper à chaque fois) ;
- ...

Toutes sont documentées dans la page de manuel de **`less(1)`**.

Il existe également des variables communes à plusieurs applications (si ce n'est à toutes). Elles permettent aussi de modifier le comportement de cette application, mais de manière identique à toutes les autres applications qui l'utilisent.

C'est par exemple le cas de `DISPLAY`, qui, dans un environnement fenêtré, permet de définir sur quel serveur X (en gros quel écran de quelle machine) l'interface du programme va s'afficher.

C'est également le cas de la variable `EDITOR` qui permet de définir un éditeur de texte par défaut. C'est celui-là qui sera utilisé à chaque fois qu'une application utilisant `EDITOR` en aura besoin.

5.6 Variables utilisateur

En tant qu'utilisateur vous pouvez définir et/ou redéfinir n'importe quelle variable. Le comportement des programmes par la suite va dépendre de ce que vous faites, mais vous êtes libre, il n'existe pas de mécanisme de protection sur les variables.

Chapitre 6

Fonctions

Il est possible, au sein même du shell, de définir des fonctions, c'est à dire des parties de code réutilisables plusieurs fois dans un ou plusieurs scripts.

6.1 Définition

En shell, une fonction est déterminée uniquement par son nom. Il n'est fait mention, ni de ses arguments, ni de ce qu'elle peut retourner. Ceci n'empêche bien entendu pas ces fonctions d'avoir effectivement des arguments et de retourner des valeurs.

La *définition* d'une fonction peut se faire de différentes manières, illustrées dans les exemples ci-dessous :

```
function mkcd ()
{
    [[ -z "$1" ]] && echo "Usage: mkcd <directory>" && return 1

    [[ -f "$1" ]] && echo "File already exists!" && return 2

    if [ ! -d "$1" ] ; then mkdir "$1" ; fi

    [[ -d "$1" ]] && cd "$1"

    return 0
}

cd ()
{
    if [ -z "$1" ] ; then builtin cd
    else builtin cd "$1"
    fi
    [[ -f .README ]] && cat .README
}
```

Dans le premier exemple, on veut définir une fonction qui combine les commandes `cd` et `mkdir`. La définition commence par le mot clé optionnel **`function`** puis est suivie du nom de la dite fonction et des parenthèses décrivant le fait que c'est d'une fonction dont on parle. Dans le corps de la définition (entre les accolades), on reconnaît l'utilisation de structures à présent bien connues, mais également celle du mot clé **`return`**, permettant de faire retourner une valeur à cette fonction.

Dans le second exemple on défini, de manière tout à fait légitime, une fonction qui porte le même nom qu'une commande interne du shell : ***cd***. Le mécanisme d'évaluation des commandes par le shell faisant qu'il teste les fonctions avant ses commandes internes, c'est bien cette fonction qui sera invoquée lorsque l'on voudra changer de répertoire. Le but de cette fonction est, de tout manière, de se substituer à la commande interne *cd* en l'enrichissant ; ici en testant l'existence d'un fichier `.README` dans le répertoire dans lequel on arrive et l'affichant le cas échéant.

Il est toujours possible de faire appel à la commande interne *cd* du shell via le mot clé *builtin* et c'est ce qui est fait dans la redéfinition de ***cd***. Si le mot clé *builtin* n'avait pas été précisé, nous aurions fabriqué une fonction récursive. De plus, dans le corps de la

fonction ***mkcd*** on fait appel à `cd`. Si les deux fonctions ont été définies, c'est bien notre ***cd*** qui sera alors invoqué.

De ces deux exemples on note que les arguments passés à la fonction sont notés `$1 ... $N`. Il ne peut y avoir qu'un maximum de 9 arguments. Leur traitement se fait de la même manière que dans le cas des arguments des scripts décrit à la section 8.2.

6.2 Utilisation

Tout comme il est extrêmement simple de définir une fonction, il est très simple de l'utiliser.

Il suffit de donner le nom de la fonction, sans parenthèse, suivi de ses éventuels arguments. La valeur retournée par la fonction se retrouve dans la variable prévue à cet effet : `$?`.

Voici des exemples d'utilisation, dans un shell, de la fonction ***mkcd*** définie au dessus :

```
[P]:~> mkcd rep
[P]:rep> echo $?
0
[P]:rep> pwd
/Users/serpaggi/rep
[P]:rep> touch exists
[P]:rep> mkcd exists
File already exists!
[P]:rep> echo $?
2
[P]:rep>
```

6.3 Variables

Il est possible de définir des variables à l'intérieur des fonctions qui ont une portée limitée. La limitation est telle que ces variables ne sont visibles que depuis la fonction et ses enfants.

Pour déclarer une telle variable, il suffit d'utiliser la commande interne `local`.

L'exemple suivant défini une fonction qui permet de compter la taille totale des fichiers présents dans un répertoire. Deux variables locales sont définies : `t` et `tot`.
On remarque également sur cet exemple qu'il est tout à fait possible que la fonction retourne la valeur d'une variable locale.

```
function taille()
{
    local t=0
    local tot=0

    for i in "$1"/*
    do
        if [[ -f "$i" ]] && [[ -r "$i" ]]
        then
            t=`wc -c "$i" | cut -b 1-8`
            tot=`expr $tot \+ $t`
        fi
    done

    return $tot
}
```

Chapitre 7

Gestion de processus

Pour comprendre la gestion du processus par le Shell, il est nécessaire d'avoir un minimum de connaissance du mécanisme de fonctionnement de ces processus, ainsi qu'un peu de vocabulaire.

7.1 Processus

Un processus, quel qu'il soit, a un numéro d'identification unique qui n'est, pour une phase de fonctionnement de l'ordinateur donnée, jamais réutilisé ; c'est le pid (*process ID*).

Un processus a, en général, un père, qui est le processus qui lui a permis d'exister. Ce dernier est repéré par son pid qui, du point du vue de ses fils, porte le nom de ppid (*parent pid*).

Un processus a plusieurs états dont les plus importants sont R (*runnable*), S (*Sleeping*) et Z (*Zombie*).

Un processus est à l'écoute des divers signaux qu'il peut recevoir. Certains de ces signaux sont destinés à la gestion de son comportement (recharger le fichier de configuration, interruption de l'exécutions, ...) d'autres sont destinés à en arrêter le fonctionnement avec plus ou moins de savoir vivre. Les signaux sont envoyés aux processus par la commande `kill(1)`.

Un processus utilise de la mémoire qui lui est réservée. Cependant, il est parfois difficile de connaître exactement la mémoire utilisée par un processus en particulier à cause de la présence de bibliothèques partagées dont l'occupation mémoire est comptabilisée plusieurs fois.

Un processus est exécuté avec une certaine priorité, ce qui détermine le pourcentage du temps processeur qu'il va monopoliser.

7.2 Gestion

Le Shell a une vocation naturelle de gestion des processus. Il mémorise plusieurs informations sur les processus dont il est à l'origine. Il est également capable, via la commande `kill(1)`, mais aussi par des procédés internes, de manipuler ces processus.

Au niveau du système, il est possible de connaître tous les processus qui sont actuellement en cours d'exécution grâce à la commande `ps(1)` (*processes list* comme ls mais pour les processus ?). On voit grâce à cette commande beaucoup d'information relative aux processus.

```
[P]:~>ps -axlc -U serpaggi
UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
719   100     1   0  31  0    78348   4328 -      Ss    ??    0:01.83 ATSServer
719   101     1   0  63  0   362880   8152 -      Ss    ??    1:34.50 loginwindow
719   159   101   0  31  0    55600   2788 -      Ss    ??    0:00.29 pbs
719   166    86   0  46  0   335688   9104 -      S     ??    0:01.45 Dock
719   168    86   0  63  0   376320   9532 -      S     ??    0:11.52 SystemUIServer
719   169    86   0  46  0   381144  12680 -      S     ??    0:02.06 Finder
719   172    86   0  46  0   346312   2808 -      S     ??    0:00.06 iTunesHelper
719   173    86   0  46  0   350852   4452 -      S     ??    0:00.15 iCalAlarmScheduler
719   174    86   0  46  0   356072   4592 -      S     ??    0:25.28 UniversalAccess
719   310    86   0  62  0   401164  31320 -      S     ??    0:06.56 Mail
719   314    86   0  46  0   352308   5772 -      S     ??    0:00.23 GrowlHelperApp
719   316    86   0  46  0    38296   2460 -      S     ??    0:00.14 AppleSpell
719   327    86   0  47  0   369236  11360 -      R     ??    0:11.16 Terminal
719 22021     1   0  97  0   995164 258160 -      Ss    ??   20:47.08 firefox-bin
719 22076     1   0  13 18    39844   4408 -      SNs   ??    0:00.24 mdimportserver
719   330   329   0  31  0    27748   1344 -      S     p1    0:00.32 -bash
[P]:~>
```

Voir également la commande `top(1)`.

7.2.1 Envoyer un signal à un processus

Comme nous l'avons évoqué plus haut, un processus est en général à l'écoute de signaux qui peuvent lui parvenir. Le but de ces signaux est de modifier le comportement du processus de manière plus ou moins radicale.

Pour envoyer un signal à un processus depuis le shell, il existe la commande `kill(1)`. Sa syntaxe est assez simple :

```
kill [-signal] <pid>
```

où `signal` peut être représenté par son code (numérique) ou son nom (alphanumérique).

Voici une liste des signaux les plus couramment utilisés :

code	nom	description
1	HUP	*hang up*, demande au processus de se terminer (coupure de ligne).
2	INT	*interrupt*, demande au processus de se terminer (interruption du processus).
3	QUIT	*quit*, demande au processus de quitter (création d'un fichier core).
6	ABRT	*abort*, demande au processus de quitter (création d'un fichier core).
9	KILL	*non-catchable, non-ignorable kill*, fin brutale du programme.
15	TERM	*software termination signal*, demande au processus de quitter comme s'il l'avait fait normalement.
17	STOP	*non-catchable, non-ignorable stop*, pause le processus immédiatement.

7.2.2 Modifier la façon dont va s'exécuter un processus

En général (il existe des exceptions), un programme lancé par un shell monopolise ce dernier jusqu'à ce qu'il se termine. Cela signifie qu'il n'est plus possible à l'utilisateur d'interagir avec l'ordinateur via cet interprète de commande particulier. Ce comportement peut être gênant si l'on n'a pas la possibilité, par exemple, d'utiliser un autre terminal virtuel avec un autre shell.

La solution pour reprendre la main consiste à envoyer au processus bloquant un signal lui demandant de s'interrompre, le signal `SIGSTOP`. Ce signal est envoyé au processus en cours grâce à la séquence de touches **[Contrôle+Z]**. Une fois le processus `STOP`pé, il est possible d'utiliser le shell normalement. Cependant, le processus existe toujours et occupe toujours des ressources système. À ce stade, plusieurs choix sont possibles :

- reprendre le fonctionnement du processus là où il s'est arrêté en bloquant à nouveau le shell (commande `fg` pour foreground) ;
- reprendre le fonctionnement du processus, mais en le reléguant en tâche de fond, c'est à dire qu'il s'exécute mais ne bloque pas l'utilisation du terminal (commande `bg` pour background) ;
- tuer le processus avec la commande `kill`.

Les processus qui sont passés en tâche de fond sont appelés des jobs. Il est possible de connaître l'état des différents jobs, ainsi que leur numéro (différent du pid) en utilisant la commande `jobs`.

Il est possible de lancer directement un processus en tâche de fond. Pour cela, il faut terminer sa ligne de commande par le caractère **&** :

```
[P]:serpaggi>find / -name "*org*" -type d > /tmp/org-files 2>/dev/null
^Z
[1]+  Stopped         find / -name "*org*" -type d >/tmp/org-files 2>/dev/null
[P]:serpaggi>bg
[1]+ find / -name "*org*" -type d >/tmp/org-files 2>/dev/null &
[P]:serpaggi>ls -lR > lslR &
[2] 22470
[P]:serpaggi>jobs
[1]-  Running         find / -name "*org*" -type d >/tmp/org-files 2>/dev/null &
[2]+  Running         /bin/ls -GF -lR >lslR &
[P]:serpaggi>
```

Nous pouvons commenter l'exemple ci-dessus. L'utilisateur désire trouver (`find`) tous les répertoires (`-type d`) dans son système de fichier (`/`) qui contiennent la chaîne de caractère org (`-name "*org*"`), stocker cette liste dans le fichier `/tmp/org-files` (`> /tmp/org-files`) et ignorer les éventuelles erreurs (`2>/dev/null`).

Il se rend compte que cela va prendre beaucoup de temps et décide donc de continuer l'exécution de cette recherche en tâche de fond. Il envoie alors au processus en cours le signal `SIGSTOP` via la séquence de touches **[Contrôle+Z]** (représenté sur la sortie par **^Z**), ce qui a pour effet de l'interrompre, puis reprend son cours en tâche de fond avec la commande interne *bg*.

Il peut alors tranquillement lancer un autre processus, qu'il sait prendre du temps, directement en tâche de fond (`ls -lR > lslR &`).

Enfin, il contrôle l'état des ses jobs grâce à la commande éponyme.

Au moment où le premier processus a été stoppé, on a pû remarquer que le shell lui avait attribué un numéro de job ([1]). Il en va de même pour le second processus lancé directement en tâche de fond ([2]). Ce sont ces numéros que l'on retrouve en début de chaque ligne de la commande jobs et qui permet de les contrôler via, par exemple, la commande `kill(1)`.

Les caractères **+** et **-** dans la sortie de la commande `jobs` représentent respectivement le job courant et le job précédent.

7.2.3 Modifier la priorité d'exécution d'un processus

Un processus donné ne peut monopoliser qu'une partie du temps processeur sur une machine multi-tâches.

En général, chacun des processus occupe de manière égale la puissance de calcul disponible. Cependant, il est possible de donner une priorité plus forte ou plus faible à certains processus. Ceci peut se faire de deux manières différentes.

nice

La priorité d'exécution d'un processus peut se déterminer au lancement de celui-ci avec la commande `nice(1)` dont la syntaxe est la suivante :

```
nice [-n incrément] <commande>
```

où `incrément` est un nombre entier compris entre -20 et 20 définissant le niveau de priorité de l'application. Plus le nombre est élevé, moins l'application occupera de temps processeur.

En tant que simple utilisateur, nous ne pouvons que "ralentir" l'exécution d'un application. Seul le super-utilisateur (root) a le droit de proposer des valeurs négatives.

renice

La priorité d'exécution d'un processus peut également se déterminer pendant son exécution à l'aide de la commande `renice(8)`. La syntaxe ressemble à celle de `nice` :

```
renice priorité [[-p] pid ...] [[-g] gid ...] [[-u] utilisateur ...]
renice -n incrément [[-p] pid ...] [[-g] gid ...] [[-u] utilisateur ...]
```

On ne passe plus le nom du processus, mais son identifiant, son pid. Il est également possible d'agir sur un groupe de processus en précisant un nom d'utilisateur ou un `gid`.

Il est important de noter que cette commande, tout comme `nice` ne permet à un simple utilisateur que de "ralentir" le fonctionnement d'un processus. De plus, une fois le processus ralenti, il est impossible de lui faire retrouver sa priorité initiale. Seul root peut faire ça !

Chapitre 8

Scripts

Un script peut être vu comme un ensemble de commandes réunies dans un fichier dans le but de faire quelque chose de cohérent ensemble.

Ces commandes peuvent être des commandes du shell, des commandes du système et d'autre scripts shell organisées par des expressions et des structures de contrôle (voir les sections correspondantes plus bas).

Un script, comme toute autre commande, est exécuté dans un sous shell, de fait, il n'hérite que des variables exportées.

De plus, les variables éventuellement définies dans les scripts ont une visibilité limité qui est celle du script.

Pour qu'un shell soit "auto"-exécutable, il doit remplir deux conditions :

1. commencer par la séquence de caractères suivante :
   `#!/bin/bash`.
   l'interprète de commande, `/bin/bash` peut être changé pour utiliser celui que vous voulez, mais ici, nous nous intéressons à `bash(1)` ;
2. être exécutable au sens du système (`chmod +x nom_du_script`).

Vous pouvez entendre parler de la séquence de caractères **`#!`** sous le nom de *she-bang*.

Il est possible de faire passer des informations, des données à un script pour qu'il les exploite. Ceci est décrit à la section 8.2.

8.1 Structures de contrôle

Comme dans tout langage de programmation, nous trouvons dans les Shells des mécanismes permettant de modifier le cours séquentiel de l'exécution d'un script ; ce sont les structures de contrôle. Elles permettent principalement de faire des tests conditionnels et des boucles.

Chacune de ces structures de contrôle utilise le résultat d'une expression pour déterminer son futur comportement. Nous allons donc commencer par aborder les expressions.

8.1.1 Expressions

Une expression peut être vue comme un test logique. C'est quelque chose (un ensemble de commandes) qui, une fois évalué, va retourner une valeur signifiant vrai ou faux (en général, 0 représente vrai et les reste représente faux).

Dans le cadre d'une structure de contrôle, une expression peut être introduite de différentes manières :

1. Une commande dont la valeur de retour sera considérée être la valeur à prendre en compte pour le test ;
2. Un test (soit introduit avec la commande interne `test` soit avec le caractère `[` (dans ce cas, il faut[1] terminer le test par le caractère `]` ;
3. Une expression commençant par le mot clé `[[` et se terminant par le mot clé `]]` ;
4. L'évaluation d'un ensemble de commandes dans un sous-shell (introduit par `(` et terminé par `)`) ;
5. L'évaluation d'un ensemble de commandes dans le shell courant (soit un bloc, commençant par { et se terminant par }).

Dans les cas 2 et 3, **il faut** laisser un espace entre le(s) crochet(s) et le reste de l'expression. En effet, il sont considérés comme une commande dans le premier cas et comme un mot réservé du shell dans le second cas.

Ci-dessous, quelques exemples d'expressions valides (la variable spéciale `$?` représente le code de retour de la dernière commande exécutée) :

```
[P]:~> test ls
[P]:~> echo $?
0
[P]:~> echo $a

[P]:~> test $a
[P]:~> echo $?
1
```

[1] même si, normalement, ça ne devrait pas être le cas

```
[P]:~> a="val"
[P]:~> [ "$a" == "valide" ]
[P]:~> echo $?
1
[P]:~> a="valide"
[P]:~> [ "$a" == "valide" ]
[P]:~> echo $?
0
[P]:~>
```

```
[P]:~> i=1
[P]:~> [[ $i -eq 5 ]]
[P]:~> echo $?
1
[P]:~> i=5
[P]:~> [[ $i -eq 5 ]]
[P]:~> echo $?
0
[P]:~>
```

Une expression peut être constituée d'un nombre quelconque de tests comme ceux présentés ci-dessus et, de plus, chacun de ces tests peut contenir des commandes complexes (tests logiques, *piping*, ...)

À l'aide de ces expressions, il est possible de construire des structures de contrôle plus compliquées telles que celles présentées ci-dessous, les conditions et les boucles.

Comme les exemples ci-dessus peuvent le laisser penser, il y a une différence d'écriture lorsque l'on compare des chaînes de caractères ou des valeurs numériques dans les tests.

Le premier type de comparaison se fait avec les opérateurs **=** (ou **==**), **!=**, **<** et **>** pour tester respectivement l'égalité, la différence, le fait qu'une chaîne soit plus petite ou plus grande qu'une autre.

Les comparaisons numériques s'effectuent avec les opérateurs **-eq**, **-ne**, **-lt** (ou **-le**) et **-gt** (ou **-ge**) pour les mêmes tests.

Il existe d'autres opérateurs de test sur les chaînes de caractères :
– **-n** vrai si la chaîne est de longueur non nulle (*non zero*) ;
– **-z** vrai si la chaîne est de longueur nulle (*zero*) ;
Enfin, il existe des opérateurs permettant d'effectuer des tests sur des fichiers dont les plus utilisés sont donnés ici :
– **-a** vrai si le fichier existe ;
– **-d** vrai si le fichier est un répertoire ;
– **-f** vrai si le fichier est un fichier ordinaire ;

– **-r** vrai si le fichier peut être lu ;
– **-s** vrai si le fichier est de taille non nulle ;
– **-w** vrai si le fichier peut être écrit ;
– **-x** vrai si le fichier peut être exécuté.

8.1.2 Conditions

Une condition est en général introduite par un des deux mots clés **if** ou **case**.

if

Syntaxe :

```
if <condition>; then <expressions>... ;
[elif <condition>; then <expressions>... ;]
[else <expressions>... ;]
fi
```

```
#!/bin/bash

if [ -z "$1" ]
then
        echo "Usage: `basename $0` <arg>"
fi
```

Cet test peut être écrit sur une unique ligne comme suit :

```
#!/bin/bash

if [ -z "$1" ]; then echo "Usage: `basename $0` <arg>" ; fi
```

Dans ce cas, les **;** sont obligatoires.

case

Une construction **case** remplace avantageusement une construction **if** quand il y a trop de **elif** cascadés.

Syntaxe :

```
case <expression> in
    <valeur_1> ) <expression_1> ;;
   [<valeur_2> ) <expression_2> ;;
    ...]
   [* ) <expression_3> ;;]
esac
```

Ceci peut-être exprimé en langage courant par :
Si `<expression>` a la valeur `<valeur_1>`, alors exécuter `<expression_1>`.
Sinon, si `<expression>` a la valeur `<valeur_2>`, alors exécuter `<expression_2>`.
Sinon, exécuter `<expression_3>`.

L'exemple suivant permet de savoir avec quel nom le script a été invoqué et permet de réagir en conséquence :

```
#!/bin/bash

case `basename $0` in
    "compress"   )
        gzip "$1" ;;
    "uncompress" )
        gunzip "$1" ;;
    *            )
        echo "Erreur : `basename $0` : commande inconnue"  ;;
esac
```

8.1.3 Boucles

Comme dans tout langage de programmation, nous retrouvons dans le shell des structures de contrôle permettant de répéter un ensemble d'opérations tant qu'un ensemble de conditions n'est pas rempli ; ce sont les boucles.

`for`

La boucle **`for`** permet de répéter l'exécution d'un ensemble de commandes sur un ensemble fini de données.

Syntaxe :

```
for <var> in <liste> ; do <commandes> ; done
```

`<var>` est simplement une variable qui va prendre tour à tour les valeurs sucessives présentes dans `<liste>`. Cette liste peut être un ensemble de données ou le résultat d'une commande renvoyant une liste (par exemple `ls`).

Ci dessous, deux exemples d'utilisation d'une boucle **`for`**.

Le premier compte le nombre de fichiers et de répertoires présent dans le répertoire où l'on exécute le script. Le second déplace un ensemble non contigu de fichiers dans un répertoire donné.

```
#!/bin/bash

rep=0
fic=0
for i in `ls` ; do
        if [ -d "$i" ]; then rep=`expr $rep \+ 1` ;
        else fic=`expr $fic \+ 1` ;
        fi
done

echo "Il y a $fic fichiers et $rep repertoires."
```

Nous pouvons remarquer que l'incrémentation des variables `fic` et `rep` fait intervenir une commande externe (`expr(1)`). En fait, ce n'est pas nécessaire avec `bash(1)`, mais le mécanisme d'arithmétique des variables n'est abordé qu'à la section 12.

```
#!/bin/bash

DEST_REP=/var/tmp

for i in 01 03 05 07 11 13 17 19 23; do
        mv -v IMG_99${i}.jpg $DEST_REP
done
```

Notons que, dans ce second exemple, les valeurs prises par la variable `i` ne sont pas numériques : c'est bien `01` et non pas 1.

while

La boucle **while** permet d'exécuter un ensemble de commandes tant qu'une condition est évaluée à **vrai**.

Syntaxe :

```
while <condition> ; do <commandes> ; done
```

Dans l'exemple suivant, nous prenons ce qu'entre l'utilisateur jusqu'à ce qu'il tape le mot `fin`.

```
#!/bin/bash

var=
n=1
while [ "$var" != "fin" ]
do
    echo "Entrez une valeur (\"fin\" pour terminer) > "
    read var
    echo "Entree $n : $var"
    n=‘expr $n \+ 1‘
done

echo "Vous avez entre $(expr $n \- 1) valeurs"
```

Dans cet exemple, il n'y a pas de **;** après la condition du `while`. C'est voulu et ce n'est pas une erreur. En effet, le **;** permet de marque la fin d'une commande quand plusieurs apparaissent sur la même ligne. Hors, dans le cas présent, le mot clé `do` est sur la ligne suivante. Nous pouvons donc nous dispenser de **;**.

until

La boucle **until** permet d'exécuter un ensemble de commandes tant qu'une condition est évaluée à **faux**.

Syntaxe :

```
until <condition> ; do <commandes> ; done
```

Vous trouverez un exemple d'utilisation de `until` à la section 8.2.

8.1.4 Altération des boucles

Il est possible d'interrompre le déroulement normal d'une boucle `for`, `while` ou `until` en utilisant les mots clés **`break`** ou **`continue`**.

Le premier permet de sortir immédiatement de la boucle, sans même exécuter les éventuellement commandes entre le `break` et la fin de la boucle, alors que le second permet d'arrêter l'itération courante pour passer immédiatement à la suivante.

Ci-dessous un script exemple illustrant la différence entre ces deux modes d'altération :

```
#!/bin/bash

echo "Break :"
i=0
while [ $i -lt 10 ]
do
    ((i++))
    [[ $i -eq 5 ]] && break
    echo $i
done

echo
echo "Continue :"
i=0
while [ $i -lt 10 ]
do
    ((i++))
    [[ $i -eq 5 ]] && continue
    echo $i
done
```

Essayez le pour vous rendre compte de la différence.

8.2 Arguments

Les arguments passés au script sur la ligne de commande lors de son invocation peuvent être retrouvés à l'intérieur de celui-ci dans les variables spéciales `$1`, ..., `$n`.

La variable `$0` représente, comme vous avez pu le voir dans certains exemples précédents, le nom du script.

L'ensemble des arguments (`$1` ...`$n`) peut être retrouvé dans deux variables :
- **`$@`**
  - **`$@`** est équivalent à **`"$1 $2 ... $n"`** ;
  - **`"$@"`** est équivalent à **`"$1" "$2" ... "$n"`**.
- **`$*`** qui, utilisé sous la forme
  - **`$*`** est équivalent à **`"$1" "$2" ... "$n"`** ;
  - **`"$*"`** est équivalent à **`"$1c$2c$n"`** ou **`c`** est le premier caractère de la chaîne trouvée dans la variable spéciale `IFS`.

Voici un petit script exemple, qui affiche l'ensemble des arguments qui lui sont fournis :

```
#!/bin/bash

until [ -z "$1" ] ; do
    echo "$1"
    shift
done
```

La commande interne *`shift`* permet de décaler la liste des arguments autant de fois que le numéro qui est passé en argument (de shift) l'ancien `$1` étant perdu. Si aucun numéro n'est fourni, le shell assume que c'est 1 et `$2` devient donc `$1`, `$3` devient `$2`, ...

Ce mécanisme permet un traitement automatique des arguments sans vraiment avoir à se soucier de sa position dans la liste.

8.2.1 Options

Quand on commence à élaborer des scripts un peu compliqués, qui de plus proposent une interaction avec l'utilisateur, il convient de définir précisemment l'ensemble des arguments qu'ils peuvent prendre. Nous parlons alors d'options, qui seront passées au programme sous la forme **`-x`** où **`x`** est une lettre ou un chiffre quelconque. Il existe un mécanisme permettant de traiter efficacement ces options : **`getopts`**.

Illustrons son fonctionnement sur un exemple :

```
#!/bin/bash

while getopts "ab:c" val ; do
    case $val in
        a) echo "Je traite l'option -a" ;;
        b) echo "Je traite l'option -b et l'argument $OPTARG" ;;
        c) echo "Je traite l'option -c" ;;
        *) echo "Je ne connais pas cette option..." ; break ;;
    esac
done

shift $((OPTIND - 1))
if [[ -n $@ ]]; then
    echo "Il reste a traiter : $@"
fi
```

La chaîne **"ab:c"** signifie que notre programmes sait traiter trois options : **-a**, **-b** et **-c**. De plus, la présence du caractère **:** à la suite du **b** signifie que l'option **-b** attend un argument.

La boucle **while** va donc parcourir l'ensemble des arguments de la ligne de commande et voir s'il rencontre une des options déclarées. Si c'est le cas, cette options est stockée dans la variable `val`. À nous d'analyser `val`.

Quand, comme pour **-b**, l'option requier un argument, ce dernier est stocké dans la variable spéciale `OPTARG` que nous devons également traiter (enfin, normalement, c'est le cas).

L'analyse prend fin dès que quelque chose ne ressemblant pas à une option (donc qui ne commence pas par le caractère **-**) est rencontré.

Une fois l'analyse terminée, l'indice du premier argument n'étant pas considéré comme une option est stocké dans la variable spéciale `OPTIND`.

Dans l'exemple, la constructions avec les doubles parenthèses **$((OPTIND - 1))** permet de faire une opération arithmétique sur la valeur de `OPTIND`. Ici, on soustrait la valeur 1. Ce fonctionnement est expliqué au chapitre 12.

Chapitre 9

Expressions régulières

Les expression régulières (parfois aussi appelées expression rationnelles) peuvent être considérées comme des filtres avancés sur des chaînes de caractères.

Nous n'allons aborder ici qu'une infime partie du vaste domaine que sont les expressions régulières et, si vous désirez en savoir plus, je vous conseille de lire les ouvrages cités en références à la fin de ce document.

9.1 Principe

Les expressions régulières répondent au besoin que l'on a parfois de pouvoir désigner un groupe hétérogène de "données" par une expression unique. Dans notre cas, nous pouvons imaginer que ces "données" sont des fichiers, des lignes de texte dans un fichier, ou encore des mots dans un fichier.

Par exemple, nous voulons identifier, dans un fichier de *log*, toutes les adresses IP. Tout ce que nous savons c'est que c'est une suite de 4 groupe de maximum 3 chiffres séparés par un caractère particulier qui est le '.' (point). Il existe donc en tout $255^4 = 4228250625$ adresses IP différentes. Il semble donc impensable de les énumérer toutes. Les expressions régulières permettent, en une courte ligne, de les identifier toutes et nous allons voir comment dans la suite de ce chapitre.

Une expression régulière est une chaîne de caractères appelé motif (ou *pattern*) dont la syntaxe est dictée par un ensemble de règles.

9.2 Caractères d'identification

Pour identifier un groupe de lettres ou de chiffres, nous disposons d'un ensemble de méta-caractères ou expressions spéciales.

Les tableaux 9.1 et 9.2 donnent une description de ces méta-caractères avec des exemples simples d'utilisation.

Ces méta-caractères sont divisés en deux groupes. Les premiers (tableau 9.1) sont historiques et existent depuis le début des premiers moteurs d'expression régulières. Ensuite, ces dernières ont été normalisées POSIX[1] et des nouveaux méta-caractères (tableau 9.2) sont apparus, rendant l'écriture des expressions régulières plus facile ; ce sont les expressions régulières étendues (ou ERE : *Extended Regular Expressions*) dont vous trouverez le nom dans de nombreuses pages de manuel.

Les méta-caractères du tableau 9.2 utilisant des crochets sont considérés comme des caractères simples. À ce titre, ils devront souvent être utilisés entre crochets dans les moteurs d'expressions régulières que vous rencontrerez pour que leurs crochets ne soient pas interprétés comme une classe de caractères. En fait, le but est de créer une classe de caractères ne contenant qu'un seul caractère : `[[:alpha:]]` par exemple.

Des exemples d'utilisation d'expressions régulières sont présentés dans la descriptions des commandes importantes au chapitre 10.

[1] *Portable Operating System Interface* : famille de standards émis par l'IEEE pour définir des API permettant des rendre les logiciels des différentes plateformes Unix, compatibles.

TAB. 9.1 – Méta-caractères pour expressions régulières

Meta-car.	Description
`.`	Identifie n'importe quel caractère qui n'est pas un saut de ligne. A l'intérieur de crochets, le point identifie le caractère point lui même. Par exemple, `a.c` identifie "`abc`", etc. mais `[a.c]` n'identifie que "`a`", "`.`" ou "`c`".
`[ ]`	Identifie n'importe quel caractère unique que est contenu dans les crochets. Par exemple, `[abc]` identifie "`a`", "`b`" ou "`c`". `[a-z]` désigne un intervalle qui identifie toutes les lettes minuscules de "`a`" à "`z`". Ces formes peuvent être mélangées : `[abcx-z]` et `[a-cx-z]` identifient toutes les deux "`a`", "`b`", "`c`", "`x`", "`y`" et "`z`". Le caractère `-` est traité comme étant effectivement le caractère `-` (et non pas quelque chose désignant un intervalle) quand il apparaît en première ou dernière position de la notation entre crochets ou s'il est échappé avec un *backslash* : `[abc-]`, `[-abc]`, ou `[a\-bc]`. Le caractère `[` peut être présent n'importe où entre les crochets, par contre, la manière la plus simple d'identifier le caractère `]` est le l'échapper avec un *backslash* (par exemple : `[\]]`).
`[^ ]`	Identifie un caractère unique qui n'est pas contenu entre les crochets. Par exemple, `[^abc]` identifie n'importe que caractère qui n'est ni "`a`", ni "`b`", ni "`c`". `[^a-z]` identifie n'importe quel caractère unique qui n'est pas dans l'intervalle des lettres minuscules allant de "`a`" à "`z`". De le même manière que ci-dessus, les intervalles et les caractères uniques peuvent être mélangés.
`^`	Identifie le début de la chaîne de caractères. Dans un contexte multilignes, ça identifie tous les débuts de lignes.
`$`	Identifie la fin de la chaîne de caractères ou la position immédiatement avant le caractère "saut de ligne". Dans un contexte multilignes, ça identifie toutes les fins de lignes.
`( )`	Définissent une sous-expression référencée. La chaîne de caractères qui a été identifiée par l'expression contenue dans ces parenthèses peut être re-référencée plus tard (voir l'expression suivante : `\n`). Une sous-expression référencée est aussi appelée un bloc ou un groupe de capture. Cette fonctionnalité n'est pas présente dans tous les moteurs d'expressions régulières et, dans de nombreux utilitaires Unix comme par exemple sed ou vi, un *backslash* doit précéder la parenthèse ouvrante et la parenthèse ouvrante pour qu'elles soient interprétées dans le sens décrit ici.
`\n`	Identifie la `n`-ieme sous-expression référencée qui a été identifiée. `n` est un entier compris entre 1 et 9. Cette construction est théoriquement irrégulière et n'a pas été retenue dans les expressions régulières étendues POSIX (ERE).
`*`	Identifie l'élément précédent zéro ou plusieurs fois. Par exemple, `ab*c` identifie "`ac`", "`abc`", "`abbbc`", etc. `[xyz]*` identifie "", "`x`", "`y`", "`z`", "`zx`", "`zxy`", "`xyzzy`", etc. `(ab)*` identifie "", "`ab`", "`abab`", "`ababab`", etc.
`{m,n}`	Identifie l'élément précédent au moins `m` fois mais pas plus de `n` fois. Par exemple, `a{3,5}` identifie seulement "`aaa`", "`aaaa`" et "`aaaaa`".

TAB. 9.2 – Méta-caractères pour expressions régulières étendues

Meta-car.	Description
`?`	Identifie l'élément précédent zéro ou une fois. Par exemple, `ba?` identifie "`b`" ou "`ba`".
`+`	Identifie l'élément précédent une ou plusieurs fois. Par exemple, `ba+` identifie "`ba`", "`baa`", "`baaa`", etc.
`|`	L'opérateur de choix (ou d'altération, ou d'union) identifie soit l'expression à sa gauche, soit l'expression à sa droite. Par exemple, `abc|def` identifie "`abc`" ou "`def`".
`[:alnum:]`	Équivalent à `[A-Za-z0-9]`
`[:alpha:]`	Équivalent à `[A-Za-z]`
`[:blank:]`	Équivalent à `[ \t]`
`[:cntrl:]`	Équivalent à `[\x00-\x1F\x7F]`
`[:digit:]`	Équivalent à `[0-9]`
`[:graph:]`	Équivalent à `[\x21-\x7E]`
`[:lower:]`	Équivalent à `[a-z]`
`[:print:]`	Équivalent à `[\x20-\x7E]`
`[:punct:]`	Équivalent à `[!"#$%&'()*+,-./:;?@[\\\]_`{|}~]`
`[:space:]`	Équivalent à `[ \t\r\n\v\f]`
`[:upper:]`	Équivalent à `[A-Z]`
`[:xdigit:]`	Équivalent à `[A-Fa-f0-9]`

Chapitre 10

Commandes importantes

Toutes les commandes présentées ici sont destinées à traiter des chaînes de caractères.

Elles sont devenues tellement populaires que vous les trouverez sur tous les systèmes de la famille Unix.

Pour traiter les chaînes de caractères, elles font, pour la majorité, un usage intensif des expressions régulières.

La *GNU Software Fundation* a souvent repris ces utilitaires en en écrivant de nouvelles versions aux fonctionnalités augmentées. Sur un système (surtout si c'est un Unix propriéraite, comme Solaris par exemple) les deux versions peuvent cohabiter. Sachez laquelle vous utilisez pour ne pas faire d'erreur dans l'interprétation des diverses options.

10.1 `wc`

La commande `wc(1)` sert à compter les caractères, les mots et les lignes d'un fichier. Les deux lettres *w* et *c* signifient en fait *word count*.

Syntaxe :

```
wc [options] <fichiers>...
```

où les options sont en nombre réduit : `-c`, `-l`, `-m`, `-w`, pour compter respectivement les octets, les lignes, les caractères et les mots dans les fichiers `<fichiers>...`.

La commande peut être utilisée comme un filtre et donc être *pipée* avec d'autres commandes.

Exemples :

```
[P]:~> wc shell.tex
    1340    6743   48414 shell.tex
[P]:~> wc -l /etc/passwd
      37 /etc/passwd
[P]:~> ls -l /tmp | wc -l
      30
[P]:~>
```

10.2 sort

La commande `sort(1)`, comme son nom l'indique, permet de trier. Elle travaille soit sur un fichier, soit sur son entrée standard, soit sur un ensemble de fichiers.

Syntaxe :

```
sort [options] <fichiers>...
```

`sort` donne sa réponse, par défaut, sur la sortie standard.

Exemple :

```
[P]:~> sort /etc/services
...
[P]:~> wc -l /etc/* 2 > /dev/null | sort -r
...
[P]:~>
```

Le première ligne affiche une liste triée des services connus du système et la seconde donne une liste, triée par ordre décroissant de leur nombre de ligne, des fichiers présents dans `/etc/`.

Il faut prendre garde à ce que cette commande trie. Par défaut, elle trie des caractères suivant leur code ASCII, ce qui fait que 2 est "plus grand" que 10. Pour éviter cela, `sort` possède l'option `-n`.
Pour illustrer cela, comparez le résultat des deux commandes suivantes :

```
[P]:~>(i=0 ; while [ $i -lt 20 ]; do echo $i ; i=$((i+1)) ; done) | sort
...
[P]:~>(i=0 ; while [ $i -lt 20 ]; do echo $i ; i=$((i+1)) ; done) | sort -n
...
[P]:~>
```

10.3 cut

La commande `cut(1)` permet, comme son nom le laisse deviner, de découper une chaîne de caractères selon des critères simples. Ici, il n'est pas question d'expressions régulières ou autres schéma compliqué, le découpage se fait selon des séparateurs ou en fonction du nombre d'octets.

La syntaxe de `cut` est la suivante :

Syntaxe :

```
cut [options] <fichiers>...
```

Cette commande peut être utilisée comme un filtre.

Les deux principales utilisations de `cut` sont les suivantes :

```
[P]:~> cut -d':' f1,7 /etc/passwd
...
nobody:/usr/bin/false
root:/bin/sh
daemon:/usr/bin/false
uucp:/usr/sbin/uucico
lp:/usr/bin/false
postfix:/usr/bin/false
...
[P]:~> who | cut -c 1-16,26-38
serpaggi console09:25
serpaggi ttyp1  09:26
[P]:~>
```

Dans le premier cas, nous retenons les champs (***f****ields*) 1 et 7 du fichier `/etc/passwd`, chacun des champs étant séparé (***d****elimited*) par le caractère `:`.

Dans le second cas, nous retenons les suites de caractères allant de 1 à 16 et de 26 à 38.

Dans le premier cas, il est à noter que le séparateur est un unique caractère et que, si plusieurs séparateurs sont présents successivement (le champ est vide), alors `cut` peut retourner ce champ vide.
Illustrons cela sur une sortie de la commande `ls -l` que nous tentons de découper pour obtenir le nom du fichier avec sa taille :

```
[P]:~> ls -l | cut -d' ' -f 9,5
...
serpaggi
serpaggi 198648
serpaggi
serpaggi
serpaggi 560623
serpaggi
...
[P]:~>
```

Le résultat obtenu est loin de ce que l'on pouvait espérer, les champs vides représentés par plusieurs espaces consécutifs étant comptés à part entière.

10.4 grep

La commande `grep(1)` permet d'afficher les lignes des fichiers correspondant à une expression donnée, comme la syntaxe l'exprime.

Syntaxe :

```
grep [options] <expression> <fichiers>...
```

La commande `grep` va donc inspecter l'intérieur de tous les fichiers qui lui sont passés en paramètre pour nous fournir le résultat.

`expression` est une expression régulière telle que décrite au chapitre 9 dans la partie concernant les expressions non étendues. Certaines versions de `grep` permettent d'utiliser les expressions régulières étendues via l'option `-E` ou via la commande `egrep` qui est une version de `grep` prenant directement en compte les ERE. Pour savoir ce qu'accepte la version de `grep` que vous utilisez, reportez-vous à sa page de manuel.

Les options que l'on passe à `grep` dépendent elles aussi beaucoup de la version utilisée, cependant, certaines sont communes comme par exemple (liste non exhaustive) :
- **`-l`** permet de n'afficher que le nom du fichier qui contient des lignes correspondant à `expression` ;
- **`-n`** permet d'afficher le numéro des lignes correspondant à `expression` ;
- **`-v`** permet de sélectionner les lignes qui ne correspondent pas à `expression`.
- **`-i`** demande à `grep` de ne pas faire de différence entre les caractères en majuscule et en minuscule.

Une option de GNU-`grep` intéressante est **`-r`**, qui permet de rechercher dans tous les répertoires de manière récursive.

10.5 find

La commande **find(1)**, contrairement à **grep**, va rechercher un ensemble de fichiers dont les caractéristiques correspondent à certains critères. L'intérieur des fichiers n'est donc pas examiné.

Syntaxe :

```
find [options] <chemin> [filtres]... [action]
```

find recherche donc, dans les fichiers présents dans la sous-arborescence désignée par `chemin`, ceux qui correspondent aux critères spécifiés éventuellement par les `filtres` et leur applique une `action` si celle-ci est donnée.

10.5.1 Filtres

La syntaxe de **find** est souvent déroutante au début, surtout à cause du format de description des filtres. Ces filtres sont en fait des options de la commande qui prennent des arguments. Par exemple, si l'on désire rechercher tous les fichiers dont le nom se termine par `.txt`, il faut donner comme format **`-name "*.txt"`**.

Les différents filtres peuvent se grouper et s'enchaîner avec les opérations logiques ET et OU. Par défaut, c'est le ET qui est considéré et si l'on désire avoir un OU il faut le préciser par l'option `-o` comme dans le cas où l'on désire rechercher les fichiers dont le nom se termine par `.tex` ou par `.ltx` : **`-name "*.tex" -o -name "*.ltx"`**.

Le groupage de filtres se fait via l'utilisation de parenthèses comme dans cet exemple où l'on désire retrouver tous les fichiers dont le nom ne contient ni `.tex`, si `.ltx` : **`! \( -name "*.tex" -name "*.ltx" \)`**. Comme nous pouvons le remarquer, il faut souvent protéger les parenthèses (avec le *backslash*) pour que le shell qui interprète cette commande ne les considère pas comme l'invocation de commandes dans un sous-shell.

Enfin, cet exemple introduit la négation de condition avec le caractère **`!`**. La négation est effectuée pour la condition suivant immédiatement le caractère **`!`**, que ce soit une condition simple ou un groupage de conditions.

Les filtres de **find** ne se limitent pas à rechercher une chaîne dans le nom du fichier, il est également possible de s'intéresser à la date de création, de modification, d'accès du fichier, mais aussi à ses droits, son propriétaire, son groupe, etc.

Le cas de la date est intéressant à détailler. Voyons ce que donne l'utilisation du filtre sur le temps d'accès. Le format est le suivant : **`-atime n`** où `n` précise le nombre de périodes de 24 heures depuis que le fichier a été accédé. Ce nombre est à interpréter comme *exactement* $n \times 24h$. Si l'on désire obtenir *plus de* ou *moins de* $n \times 24h$, alors il faut faire précéder n des caractères `+` ou `-` respectivement : **`-atime +7`** désigne tous les fichiers qui n'ont pas

été lus ou écrits depuis plus de 7 jours.

10.5.2 Actions

Si une action est précisée sur la ligne de commande, elle est entreprise pour chacun des fichiers trouvé. Il existe plusieurs types d'actions :

1. **`-print`** qui affiche le résultat sur la sortie standard. En fait, si cette action n'est pas précisée et qu'il n'y en a aucune autre, elle est ajoutée par défaut.
2. **`-exec <cmd> [options]`** `{}` **`;`** qui exécute la commande `cmd` avec les options `options`. Le paramètre de la commande est obtenu par le couple d'accolades qui représente le nom complet du fichier trouvé par **`find`**. Le filtre `exec` se termine au premier **`;`** trouvé. Tout comme les parenthèses, celui-ci devra souvent être protégé pour que le shell ne l'interprète pas comme une fin de commande.
3. **`-ok <cmd> [options]`** `{}` **`;`** a le même comportement que `exec` mis à part le fait que chaque action doit être validée par l'utilisateur.

Pour illustrer cela, nous recherchons, pour les effacer, tous les fichiers de notre répertoire personnel qui se terminent par `.o` et qui ont plus de 3 jours :

```
[P]:~> find $HOME -type f -name "*.o" -ctime +3 -exec rm -f {} \;
[P]:~>
```

Chapitre 11

Redirections avancées

11.1 Duplication de flux

Il possible de dupliquer les différents flux que manipule le shell. Ceci est en général fait pour offrir plus de souplesse à un script, rediriger les messages d'erreurs dans un fichier particulier, ou toute autre application que vous pourrez imaginer.

Il est possible de dupliquer les flux en sortie (stdout et stderr) mais également le flux d'entrée (stdin).

Pour cela, il y a une syntaxe qui ressemble beaucoup à ce que nous avons vu pour l'instant dans les cas des redirections simples :
- `[n]<&word` duplique le flux d'entrée `n` (0 par défaut) et le remplace par le flux désigné par `word`.
- `[i]>&j` permet de rediriger le flux `i` (1 par défaut) vers le flux `j`.
- `&>j` est l'équivalent de `>j 2>&1`.

Les duplications de flux d'entrée sont surtout utiles dans le cas où l'on désire lire un fichier depuis un script (voir plus loin).

Les duplications de sorties sont plus souvent utilisées pour, par exemple, mélanger la sortie d'erreur et la sortie standard.

Dans l'exemple suivant, il y a fort à parier que, si l'on n'est pas identifié en tant que `root`, la commande `find` produise des erreurs. Nous avons vu dans le chapitre dédié aux redirections comment ignorer cette sortie d'erreur. Nous allons voir ici comment la rediriger vers le même fichier que la sortie standard :

```
[P]:~> find / -name "*.tex" > $HOME/LsTeX 2>&1
[P]:~> ls $HOME/LsTeX
/users/chic/serpaggi/LsTeX
[P]:~>
```

Dans ce cas, l'ordre de déclaration des redirection a une importance. En effet, si l'on demande de dupliquer le flux 2 sur le flux 1 avant que ce dernier n'ait-été redirigé, alors nous verrons les erreurs s'afficher sur la sortie standard.

11.2 Here document

Le *Here document* est une construction particulière qui permet d'éviter au programmeur de la frappe.

Le but est de fournir des données à une commande tant qu'une chaîne de caractère spéciale n'a pas été rencontrée.

Le nom de *Here document* vient du fait que, au début, cette technique était surtout employée pour passer des lignes de texte à une commande les traitant. La syntaxe de cette redirection spéciale est décrite ci-dessous.

Syntaxe :

```
cmd << string
donnee_1
donnee_2
...
donnee_n
string
```

Cette redirection est donc introduite par les doubles chevrons `<<`. Les lignes `donnee_i` sont passée successivement à la commande **`cmd`** jusqu'à ce que l'on rencontre la chaîne de caractères `string`. Cette dernière est définie par l'utilisateur qui prendra soin de la choisir suffisamment unique pour qu'elle n'apparaisse pas dans les lignes `donnee_i`.

Voici un exemple d'utilisation de ce mécanisme avec la commande `wall(1)`, tiré du *Advanced Bash Scripting Guide* :

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
E-mail your noontime orders for pizza to the sys admin.
    (Add an extra dollar for anchovy or mushroom topping.)
# Additional message text goes here.
# Note: 'wall' prints comment lines.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
#         wall <message-file
#  However, embedding the message template in a script
#+ is a quick-and-dirty one-off solution.
exit 0
```

Dans cet exemple, le choix de la chaîne de caractère servant à délimiter le message est particulier, mais efficace.

La redirection `<<` a des déclinaisons :

- `<<-` qui permet de supprimer toutes les tabulations présentes devant les lignes `donnee_i`. Ceci ne supprime pas les espaces.
- `<<<expression` qui est aussi appelé *Here string* passe le résultat de l'évaluation de `expression` à la commande. Il n'y a donc, dans ce cas, pas de bloc de paramètres définis.

Pour clore cette section sur les duplications de flux, je vous propose une dernier exemple, toujours tiré du *Advanced Bash Scripting Guide*, mais traduit cette fois ci :

```
#!/bin/bash
# upperconv.sh
# Converti un fichier passe sur l'entree standard en majuscules

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]    # Le fichier en parametre est-il lisible ?
then
    echo "Impossible de lire le fichier !"
    echo "Utilisation : $0 <fichier-entree> <fichier-sortie>"
    exit $E_FILE_ACCESS
fi

if [ -z "$2" ]
then
    echo "Il faut un fichier de sortie."
    echo "Utilisation : $0 <fichier-entree> <fichier-sortie>"
    exit $E_WRONG_ARGS
fi

exec 4<&0   # On sauvegarde l'entree standard dans le fichier
            #+ dont le descripteur est 4
exec < $1   # La lecture se fera depuis le fichier passe comme
            #+ premier parametre

exec 7>&1   # Dupliquer la sortie standard pour la sauvegarder
exec > $2   # L'ecriture se fera dans le fichier passe comme
            #+ second parametre. On considere qu'il est autorise
            #+ en ecriture.

# ------------------------------------------------------------
    cat - | tr a-z A-Z    # conversion en majuscules.
#   ^^^^^                 # lecture depuis stdin.
#           ^^^^^^^^^^    # ecriture sur stdout.
# Cependant, les deux ont ete rediriges.
# ------------------------------------------------------------

exec 1>&7 7>&-    # On retabli stdout
exec 0<&4 4<&-    # On retabli stdin

# A present, la ligne suivante donne sa sortie sur stdout comme prevu
echo "Fichier \"$1\" a converti en majuscule dans \"$2\"."

exit 0
```

11.3 Redirections de blocs

Pour finir avec ce chapitre sur les redirections avancées, nous allons parler de la redirection de blocs. Ce terme de bloc s'applique en fait aux blocs conditionnels que nous avons vu au chapitre des scripts, c'est à dire aux constructions `while`, `for`, etc.

Il est possible de redéfinir le flux d'entrée de ces structures à l'aide d'une simple redirection, placée à la fin de la structure.

Prenons le cas où l'on doive lire un fichier jusqu'à rencontrer une chaîne donnée. Comment faire ? Une solution serait d'utiliser le mécanisme de duplication de flux vu ci-dessus, mais il est lourd à mettre en place et il existe une solution plus simple :

```
#!/bin/bash

if [ -z "$1" ]
then
    fichier="tok.txt"    # On va lire un fichier par defaut
else
    fichier="$1"         # On va lire le fichier de l'utilisateur
fi
echo "Lecture du fichier $fichier"

while [ "$tok" != "function" ]
do
    read tok
    echo $tok
done <"$fichier"

exit 0
```

La même chose peut se faire avec les boucles `for` et `until` tout aussi simplement.

Chapitre 12

Variables avancées

L'interprète bash permet un traitement intéressant des variables. En effet, il peut, en utilisant des notations simples, définir des valeurs par défaut, faire des substitutions automatiquement et même faire des calculs arithmétiques

12.1 Valeurs par défaut

Il est possible, pour les variables, de définir des valeurs par défaut de deux manières différentes.

12.1.1 Utilisation sans définition

`${var:-defaut}`

où `defaut` est une expression qui est évaluée et dont le résultat sera retourné si la variable `var` est nulle ou non définie.

À la sortie, la `var` n'aura pas été modifiée.

```
[P]:~> echo $UNDEFVAR

[P]:~> echo ${UNDEFVAR:-Vide}
Vide
[P]:~> echo $UNDEFVAR

[P]:~>
```

12.1.2 Utilisation et définition

`${var:=defaut}`

où `defaut` est une expression qui est évaluée et dont le résultat sera affecté à la variable `var` si cette dernière est nulle ou non définie. La valeur de `var` est alors retournée.

Contrairement au cas précédent, `var` est redéfini.

```
[P]:~> echo $UNDEFVAR

[P]:~> echo ${UNDEFVAR:-Rempli}
Rempli
[P]:~> echo $UNDEFVAR
Rempli
[P]:~>
```

12.2 Suppressions et substitutions

Il est possible de modifier la valeur retournée par une variable (et non pas la valeur de la variable elle même) avec des mécanismes de suppression et de substitution.

12.2.1 Suppression avant

Il est possible de supprimer un bout de chaîne de caractères au début de la valeur d'une variable (préfixe). La syntaxe est la suivante :

`${var#expr}`
`${var##expr}`

Les deux notations diffèrent par le fait que, la première supprime de la valeur de `var` la plus petite occurence de `expr` alors qu'en utilisant la seconde, c'est la plus grande occurence qui est supprimée.

Une utilisation serait, par exemple, de supprimer l'ensemble des répertoires du chemin absolu d'un fichier pour ne conserver que son nom, comme dans l'exemple suivant où l'on recherche tous les services susceptibles d'être lancé au démarrage du système, quel que soit le niveau de démarrage :

```
[P]:~> for i in $(find /etc/rc*.d -type f -name "S*")~; do
> echo ${i#rc*.d}
> done
...
[P]:~>
```

L'exemple suivant permet d'illustrer, sur un cas simple, a différence entre la première et la seconde forme :

```
[P]:~> MAVAR=abbb
[P]:~> echo "[${MAVAR#ab*}]"
[bb]
[P]:~> echo "[${MAVAR##ab*}]"
[]
[P]:~>
```

12.2.2 Suppression arrière

Il est possible de supprimer un bout de chaîne de caractères à la fin de la valeur d'une variable (suffixe). La syntaxe utilisée est la même que précédemment, à part le caractère spécial qui diffère : c'est le `%` qui est utilisé :

```
${var%expr}
${var%%expr}
```

Comme précédemment, le fait de doubler le caractère spécial permet de remplacer l'occurence la plus longue de `expr` dans `var`.

L'exemple suivant recherche l'ensemble des fichiers du système qui se terminent par `.jpg` ou par `.JPG` (des images au format jpeg[1]) pour écrire, dans le fichier `batch`, la ligne de commande qui pemettrait de les convertir au format png[2] à l'aide de l'utilitaire `convert`.

```
[P]:~> for i in $(find . -type f -name "*.jpg" -o -name "*.JPG") ; do
echo "convert ${i} ${i/.[jJ][pP][gG]}.png" >> batch ; done
```

12.2.3 Remplacement

Nous avons vu comment supprimer les préfixes et les suffixes de la valeur d'une variable, mais il est également possible de remplacer une partie de cette valeur.
La notation suivante le permet :

[1] *Joint Photographic Experts Group.*
[2] *Portable Network Graphic.*

TAB. 12.1 – Opérateurs arithmétiques et logiques sur les variables

Opérateur	Signification
`id++ id--`	post incrémentation et décrémentation d'une variable
`++id --id`	pré incrémentation et décrémentation d'une variable
`- +`	plus et moins unaires (-1 ou $+1$ par exemple)
`!`	négation au sens logique et au sens bit à bit
`**`	exponentiation
`* / %`	multiplication, division et reste de la division entière
`+ -`	addition et soustraction ($1 + 2$ par exemple)
`<< >>`	décalage bit à bit vers la gauche et vers la droite
`<= >= < >`	comparaison
`== !=`	égalité et ingalité
`&`	ET (logique) bit à bit
`^`	XOR (OU eXclusif logique) bit à bit
`|`	OU (logique) bit à bit
`&&`	ET logique
`||`	OU logique
`expr ?expr:expr`	opérateur ternaire d'évaluation conditionnelle
`= *= /= %= += -= <<= >>= &= ^= |=`	affectation avec, éventuellement, opération préalable
`expr1 , expr2`	opérateur vigule permettant de faire plusieurs opérations en une seule instruction logique. La valeur de la dernière opération est retournée.

```
${var/expr/remplacement}
${var//expr/remplacement}
```

Dans le premier cas, la première occurence de `expr` dans `var` est remplacée par `remplacement`. Dans le second cas, ce sont toutes les occurences qui sont remplacées.

L'exemple suivant est le même que pour la suppression du suffixe ; la façon de faire est simplement différente. Nous remplaçons le suffixe `.jpg` ou `.JPG` par `.png`.

```
[P]:~> for i in $(find . -type f -name "*.jpg" -o -name "*.JPG") ; do
echo "convert ${i} ${i/.[jJ][pP][gG]/.png}" >> batch ; done
```

12.3 Arithmétique

Même si les variables ne sont pas typées avec bash, nous avons vu qu'il est possible de les utiliser pour faire des calculs simples (calculs sur des entiers). Nous avons jusqu'à présent

utilisé la commande `expr` pour, par exemple, incrémenter la valeur d'un compteur. Il existe un mécanisme interne permettant de faire cela.

L'arithmétique sur les variables en bash s'introduit par la notation `((` et se termine par `))`. Ce qui est entre ces délimiteurs est considéré comme une expression arithmétique à évaluer.

Il est à noter que les évaluations ne se font qu'en entiers et que seule la division par 0 est détectée ; les dépassements de capacité ne le sont pas.

La priorité des opérateurs est la même qu'en C et, en fait, les opérateurs sont également les même que ceux que l'on trouve dans ce language de programmation.

La table 12.1 donne la liste de ces opérateurs ainsi que leur signification. Cette liste est présentée de telle manière à ce que les opérateurs ayant la même priorité soient sur la même ligne. Le tableau est globalement trié par ordre de priorité décroissante. Ce tableau a été repris de la page de manuel de `bash(1)`.

En pratique, il faudra préfixer la notation `((` d'un `$` pour l'utiliser dans les expressions du shell en tant que valeurs.

L'exemple suivant propose un script utilisant des expressions arithmétiques. En fait il regroupe l'utilisation de beaucoup de choses vues jusqu'à présent : les fonctions, les variables locales, les opération arthmétiques, le traitement des paramètres, les redirections, etc.

```
#!/bin/bash

function seq()
{
    local rnd=$RANDOM # $RANDOM est une fonction interne de bash
              # retournant un nombre pseudo aleatoire
              # entre 0 et 32767 (2**15-1)

    MAX=$1
    # On desire un nombre entre 0 et MAX
    rnd=$((rnd%$MAX+1))

    echo "Liste des nombres entre 0 et $rnd :"

    i=0
    while [[ $i -le $rnd ]]
    do
        echo $i
        ((++i))
    done
}

## Debut du script
###################

if [ -z "$1" ]
then
    cat << FIN
Usage : `basename $0` <entier>
  Genere la liste des entiers entre 0 et RND
  ou RND est un nombre aleatoire entre 0 et <entier>
FIN

    exit 1
fi

seq $1
```

Chapitre 13

Références

1. `man sh(1)`
2. `man bash(1)`
3. `man re_format(7)`
4. "Bash Guide for Beginners"
   (`http://tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html`)
5. "Advanced Bash-Scripting Guide"
   (`http://tldp.org/LDP/abs/html/`)
6. "Unix Text Processing"
   (`http://www.oreilly.com/openbook/utp/`)
7. "Maîtrise des expressions régulières, 3^e édition", Jeffrey E. F. Friedl, O'Reilly, 2006.