Le Bourne shell (sh)

Bernard Cassagne

Laboratoire de Génie Informatique

IMAG

Grenoble

novembre 1991

Chapter 1

Les bases

1.1 Présentation

Un interpréteur de commandes (un shell dans le jargon UNIX) est un programme qui sert d'intermédiaire entre l'utilisateur (une personne physique) et le système (un ensemble de programmes).

Le service essentiel rendu par un shell est l'exécution de programmes. En simplifiant, un shell exécute une boucle infinie sur les actions suivantes:

- lecture d'une ligne
- interprétation de cette ligne comme une demande d'exécution d'un programme avec des paramètres.
- lancement de ce programme avec passage des paramètres.

Par exemple, si l'on tape:

```
grep resu main.c
```

le shell interprète cela comme étant la demande d'exécution du programme `grep` avec les paramètres `resu` et `main.c`.

Tous les autres services offerts par le shell n'ont d'autre but que de faciliter cette tâche fondamentale qu'est l'exécution de programme.

Parmi les autres services de base du shell, on rencontre essentiellement:

- la génération de noms de fichiers
- la redirection des entrées-sorties
- la possibilité d'exécution en avant plan ou en arrière plan.
- la possibilité d'exécuter plusieurs programmes en parallèle en les faisant coopérer selon la technique du travail à la chaîne.
- la gestion de variables
- la substitution de commandes
- les fichiers de commandes

1.2 L'exécution de programme

1.2.1 Programmes et paramètres

Quand le shell analyse une ligne de commande, il découpe cette ligne en mots séparés par des blancs. Une fois ce découpage réalisé, le premier mot de la ligne est interprété comme étant le nom d'un fichier à exécuter, les autres mots sont considérés comme des paramètres à passer au programme. Le shell ne plaque aucune sémantique sur les paramètres, pour lui se sont de simples chaines de caractères. C'est la commande appelée qui affectera une sémantique aux paramètres. Par exemple:

```
grep -i alloc names.c
```

le shell passe les chaines de caractères `-i`, `alloc`, `names` que `grep` interprètera respectivement comme étant:

1. une option (`-i`) signifiant ne pas faire de différence entre majuscules et minuscules
2. une chaîne de caractères (`alloc`) : la chaîne a rechercher
3. un nom de fichier (`names.c`) : le fichier dans lequel procéder à la recherche.

1.2.2 Paramètres formés de plusieurs mots

Il arrive parfois que la manière dont le shell découpe la ligne en paramètres ne satisfasse pas l'utilisateur. Supposons que l'on ait crée un ficher qui soit un annuaire de téléphone. Il est formé d'un ensemble de lignes de la forme suivante:

```
Jacques Eudes 9056
Serge Rouveyrol 4567
```

Si ce fichier a pour nom `annuaire`, quand on désire obtenir un numéro de téléphone, il suffit de faire:

```
$ grep -i rouveyrol annuaire
Serge Rouveyrol 4567
$
```

Mais pourquoi ne peut on pas faire:

```
$ grep -i serge rouveyrol annuaire
rouveyrol: No such file or directory
annuaire:Serge Rouveyrol 4567
$
```

on voulait rechercher la chaîne `serge rouveyrol` dans le fichier `annuaire`. Mais le shell n'a pas compris que pour nous `serge rouveyrol` forme un tout, il a passé à `grep` 4 paramètres qu'il a interprété de la manière suivante: `serge` : chaîne à rechercher, `rouveyrol` et `annuaire` : noms de fichiers dans lesquels procéder à la recherche. Le message d'erreur indique qu'il n'a (et pour cause) pas trouvé de fichier de nom `rouveyrol`.

Pour résoudre ce problème, il faut indiquer au shell que les mots `serge` et `rouveyrol`, bien que séparés par un blanc, forment un seul paramètre. Cela se réalise en entourant le paramètre de deux caractères `'` (simple quote), comme ceci:

```
$ grep -i 'serge rouveyrol' annuaire
Serge Rouveyrol 4567
$
```

A la place du caractère ', on peut aussi utiliser le caractère " (double quote).

```
$ grep -i "serge rouveyrol" annuaire
Serge Rouveyrol 4567
$
```

1.2.3 Interprétation des blancs

La phase consistant pour le shell à découper une ligne de commande en nom de programme et paramètres porte le nom d'*interprétation des blancs*. En effet, chaque blanc est interprété en fonction du contexte dans lequel il se trouve: un blanc est un séparateur de mot à l'extérieur des quotes, mais pas à l'intérieur.

Le caractère *espace* n'est pas le seul caractère “blanc”. Les caractères blancs sont contenus dans la variable `IFS` (Internal Field Separator) et sont par défaut l'espace, la tabulation et le caractère (*line-feed*).

1.3 Génération de noms de fichiers

1.3.1 Présentation

Travailler avec le shell nécessite de manipuler à tout instant des noms de fichier : l'activation d'une commande nécessite le plus souvent de lui passer en paramètre un ou plusieurs noms de fichiers. Il est donc agréable que le shell offre des moyens puissants de désignation de fichiers. Supposons que l'on ait un répertoire contenant quelques dizaines de fichiers que l'on désire détruire. Taper un par un les noms de fichiers pour les donner en paramètre à la commande `rm` (remove) serait très peu ergonomique. On sent donc bien le besoin de pouvoir exprimer : `rm` *tous-les-fichiers*.

Le shell `sh` a adressé ce problème dont la solution porte le nom technique de *génération de noms de fichiers*. Elle passe par l'utilisation d'un certains nombre de caractères servant à construire des modèles. Un modèle permet de désigner un ensemble de noms de fichiers.

Il existe trois constructeurs de modèles qui sont `*`, `?` et `[]`.

1.3.2 Le constructeur *

Un modèle de la forme X*Y où X et Y sont des chaînes quelconques de caractères, éventuellement nulles, désigne l'ensemble des noms de fichiers de la forme XUY où U est une chaîne de caractères quelconques eventuellement nulle.

Exemple:

`*`	désigne tous les fichiers
`toto*`	désigne tous les fichiers commencant par `toto`
`*.c`	désigne tous les fichiers se terminant par `.c`
`*/*.c`	désigne tous les fichiers se trouvant dans un répertoire quelconque et se terminant par `.c`

1.3.3 Le constructeur ?

Un modèle de la forme X?Y où X et Y sont des chaînes quelconques de caractères, éventuellement nulles, désigne l'ensemble des noms de fichiers de la forme XuY où u est un caractère quelconque.

`?`	désigne tous les fichiers dont le nom comporte un seul caractère
`fic.?`	désigne tous les fichiers de type `fic.x`

1.3.4 Le constructeur []

Un modèle de la forme X [abc...z]Y où X et Y sont des chaînes quelconques de caractères, éventuellement nulles, et abc...z une chaîne littérale de caractères, désigne l'ensemble des noms de fichiers ayant l'une des formes suivantes: XaY ou XbY ou ... ou XzY.

Dans le cas où l'ensemble des caractères abc ... z forment une suite continue lexicographiquement, on peut utiliser la raccourci d'écriture suivant : a-z. On peut de surcroit mélanger les deux techniques, par exemple utiliser `[a-z.;,]`

Exemples:

`[a-z].[0-9]`	désigne tous les fichiers de type a.0 a.1 ... b.0 b.1 etc ...

Particularité du sh System V

Le `sh` de UNIX Systeme V, par opposition à celui de UNIX BSD a la possibilité de mettre le signe `!` juste après le `[` pour désigner non pas l'ensemble des caractères entre crochets, mais le complément de cet ensemble

Exemples:

`*[!a-z]`	désigne les fichiers se terminant par autre chose qu'une lettre

1.3.5 Mise en œuvre de la génération de noms de fichiers

Après avoir découpé une commande en mots, le shell scrute chaque mot à la recherche des métacaractères `* ? [ ]`. Si un mot comporte un métacaractère, il est considéré comme un modèle, la génération de noms de fichiers est déclenchée, et le modèle est remplacé par l'ensemble des noms de fichiers qu'il désigne. Ces noms sont classés par ordre alphabétique. Exemple:

```
$ ls -l texinfo*
-rw-r--r--  1 bernard      4035 Oct 10 11:10 texinfo
-rw-r--r--  1 bernard     50848 Oct 10 11:10 texinfo-1
-rw-r--r--  1 bernard     51697 Oct 10 11:10 texinfo-2
-rw-r--r--  1 bernard     52123 Oct 10 11:10 texinfo-3
-rw-r--r--  1 bernard     26458 Oct 10 11:10 texinfo-4
-rw-r--r--  1 bernard    188731 Oct 10 11:09 texinfo.texinfo
$
```

1.4 Redirection des entrées-sorties

Un programme s'exécutant sous UNIX dispose, pour réaliser ses entrées-sorties, d'un certain nombre de *périphériques logiques* repérés par un numéro de 0 à N. Parmi ces périphériques logiques, trois sont particularisés:

- Le périphérique logique 0 a pour vocation de réaliser des entrées, on le nomme *entrée standard*, il est affecté par défaut au clavier du terminal.
- Le périphérique logique 1 a pour vocation de réaliser des sorties, on le nomme *sortie standard*, il est affecté par défaut à l'écran du terminal.
- Le périphérique logique 1 a pour vocation d'être le support des mesages d'erreurs, on le nomme *erreur standard*, il est affecté par défaut à l'ecran du terminal.

L'expression "a pour vocation de" signifie qu'il s'agit d'une norme d'utilisation des périphériques logiques, mais rien dans le noyau UNIX oblige à les utiliser de cette façon.

Un grand nombre de programmes se contentent de lire un flot de donnés, de faire un traitement sur ces données et d'écrire un flot de données résultat. Ces programmes prennent donc leurs données sur l'entrée standard et écrivent leurs résultats sur la sortie standard. Un programme qui respecte cette convention porte (dans le jargon UNIX) le terme de filtre. On a donc le schéma suivant:

```
                           +-------+
                           |       |
      entree standard  ---> | prog  | --->   sortie standard
         (clavier)         |       |            (ecran)
                           +-------+
```

Prenons comme exemple `bc` (binary calculator), il lit une ligne au terminal, interprète cette ligne comme étant une expression à calculer, calcule l'expression et imprime le résultat sur le terminal. Voici un exemple d'utilisation interactive de `bc`:

```
$ bc -l
456 + 1267
1723
84 * 35
2940
quit
$
```

(L'argument -l permet d'avoir une plus grande précision dans les divisions).

On peut cependant imaginer beaucoup de situation où l'on aimerait activer `bc` en lui faisant lire les expressions non pas à partir du terminal, mais à partir d'un fichier. Imaginons par exemple que l'on désire faire des statistiques de vitesse sur des communications. Supposons que les communications sont gérées par un logiciel qui écrit des messages dans un fichier de log, messages qui ont la forme suivante:

```
from pad: 1254 characters transmitted in 34 seconds
from pad:  687 characters received in 23 seconds
```

A l'aide d'un éditeur il est facile de transformer ce fichier de façon a remplacer chaque ligne par l'expression : nombre de caractères divisé par nombre de secondes. Sous `vi` cela peut se faire en 3 commandes:

```
:%s/^[^0-9]*//
:%s+car.*in+/+
:s/ seconds//
```

Ces trois commandes agissent sur l'ensemble des lignes du fichier (%). La première substitue l'ensemble des caractères n'étant pas des chiffres (`[^0-9]` et commençant au début de la ligne (`^`), par le vide. La seconde remplace la chaîne débutant par `car` et se terminant en `in` par le signe de la division (`/`). La troisième remplace la chaîne *espace* `seconds` par le vide.

Notre fichier se trouve alors transformé en:

```
1254 / 34
687 / 23
```

Pour faire calculer ces expressions par `bc`, il suffit de l'activer en lui faisant prendre le flot de données d'entrée non pas à partir du terminal, mais à partir du fichier qui les contient, par exemple `data`. Cela se fait de la manière suivante:

```
$ bc -l < data
36.88235294117647058823
29.86956521739130434782
$
```

Si le fichier a une taille de l'ordre du millier de lignes, on voit le temps gagné.

Ce que nous avons réalisé avec `bc` est général: le fait d'exécuter une commande suivie du signe `<` suivi d'un nom de fichier, a pour effet de rediriger l'entrée standard de cette commande à partir du fichier indiqué.

On peut de la même manière rediriger la sortie standard d'un programme vers un fichier en faisant suivre la commande du signe `>` suivi d'un nom de fichier. Par exemple:

```
ls > fics
```

mettra le résultat de la commande `ls` dans le fichier `fics`.

On peut combiner les deux mécanismes et rediriger à la fois l'entrée et la sortie d'un programme. Par exemple:

```
$ bc -l < data > resu
$
```

`bc` est activé en lisant les expressions dans le fichier `data` et en écrivant les résultats dans le fichier `resu`.

1.5 Exécution en séquence

Il est possible de demander l'exécution en séquence de plusieurs commandes. Cela s'obtient à l'aide de l'opérateur `;` (point virgule). Exemple:

```
cd src; ls -l
```

1.6 Exécution en premier plan ou en arrière plan

Lorsqu'on demande l'exécution d'une commande, le shell lance la commande et se met en attente de sa terminaison. Ceci est mis en évidence aux yeux de l'utilisateur par le fait que le shell n'imprime un prompt que lorsque la commande en cours est terminée. Quand on travaille de cette manière on dit que l'on exécute les programmes en premier plan (foreground). Dans la grande majorité des cas, les commandes ont une durée d'exécution courte et cette technique est bien adaptée. Il existe par contre certaines commandes qui demandent un long temps d'exécution comme des compilations de gros fichiers par exemple. Si ces exécutions ne sont pas interactives, attendre la fin de leur exécution est une perte de temps. Il existe donc une possibilité pour l'utilisateur de demander au shell de lancer l'exécution d'un programme et de ne pas attendre la fin de son exécution. L'utilisateur pourra alors continuer à interagir avec le shell pendant que le programme s'exécute. Quand on exécute un programme de cette façon, on dit qu'on l'exécute en arrière plan (dans le background). Pour demander ce service, il suffit de taper le caractère `&` à la fin de la commande. Exemple:

```
$ cc -o essai essai.c &
```

On peut combiner redirection des entrées-sorties et mise en tâche d'arrière plan. C'est même nécessaire si le programme réalise des sorties, car sinon elles se mélangeraient avec les sorties du programme en cours. Dans l'exemple suivant, on lance un tri en arrière plan en redirigeant la sortie standard sur le fichier `donnees_triees`

```
$ sort donnees_brutes > donnees_triees &
```

1.7 Travail à la chaîne

Nous avons vu qu'un programme lisait de l'information sur son entrée standard, effectuait un certain traitement et écrivait le résultat sur sa sortie standard. Il arrive très souvent que l'on désire faire coopérer deux programmes de manière à ce que le second effectue son travail sur l'information produite par le premier selon le schéma suivant:

```
                  +-------+             +-------+
                  |       |             |       |
        entree ---> |  Pg1  | ----------> |  Pg2  | --->   sortie
                  |       |             |       |
                  +-------+             +-------+
```

Une telle organisation est ce que l'on appelle communément du travail à la chaîne.

Pour réaliser cela, le système établi un tampon entre les deux programmes, et connecte la sortie du premier programme à l'entrée du tampon, et l'entrée du second programme à la sortie du tampon. Dans le jargon UNIX, un tel tampon porte le nom de *pipe*.

On a alors le schéma suivant:

```
             +-------+          pipe            +-------+
             |       |      +-----------+       |       |
entree ---> |  Pg1  |  ---> |           |  ---> |  Pg2  |  ---> sortie
             |       |      +-----------+       |       |
             +-------+                          +-------+
```

Le shell permet de créer deux programmes s'exécutant selon une telle méthode. Il suffit de taper les textes d'activation des deux commandes séparés par le signe conventionnel | qui demande l'établissement d'un pipe. On aura donc: *commande1* | *commande2*

Voyons un exemple pratique d'utilisation. Supposons que l'on travaille sur une grosse machine sur laquelle plusieurs dizaines d'utilisateurs sont connectés. On désire savoir si une certaine personne est actuellement connectée. Il existe une commande `who` qui donne la liste des utilisateurs connectés:

```
$ who
bernard  ttyp0   Oct 16 13:00
eudes    ttyp3   Oct 18 12:06
langue   ttyp8   Oct 19 16:03
mounier  ttyp9   Oct 19 11:41
henzinge ttypa   Oct 19 14:05
$
```

Si la liste est très longue, il est difficile de repérer à l'œil la personne cherchée. Faisons faire le travail par la machine:

```
$ who | grep eudes
eudes    ttyp3   Oct 18 12:06
$
```

1.7.1 Notion de pipe-line

La technique que nous venons de voir peut se généraliser à un nombre quelconque de programmes. Exemple:

```
prog1  |  prog2  |  prog3  |  prog4
```

Un ensemble de programmes connectées par pipes porte le nom de *pipe-line*.

Du point de vue de la redirection des entrées-sorties, il est possible de rediriger l'entrée standard du premier programme, et la sortie standard du dernier: pipe-line:

```
prog1  <  data_in   |  prog2   |  prog3   |  prog4  >  data_out
```

Un pipe-line peut également être éxécuté en arrière plan:

```
prog1  <  data_in   |  prog2   |  prog3   |  prog4  >  data_out   &
```

1.8 Les variables

Le shell permet de manipuler des variables. Les variables n'ont pas besoin d'être déclarées, et elles n'ont pas de type (au sens des langages de programmation). En effet, elles ne peuvent avoir comme valeur que des objets d'un seul type: des chaînes de caractères. Pour affecter une valeur à une variable il suffit de taper le nom de la variable, suivi du signe =, suivi de la valeur que l'on désire affecter à la variable. Par exemple:

```
MBOX=/users/bernard/mbox
```

Attention à ne pas laisser de blanc autour du caractère =.

Pour référencer la valeur d'une variable, on utilise la notation consistant à écrire le signe `$` suivi du nom de la variable. Par exemple:

```
vi $MBOX
```

résultera en un `vi /users/bernard/mbox`. Le travail du shell consistant à remplacer `$`*nom-de-variable* par la valeur de la variable, porte le nom de *substitution de variable.*

En travail interactif, les variables sont très utiles pour faire des abréviations. Supposons que l'on ait besoin d'agir de manière répétée sur les fichiers `donnees_brutes` et `donnees_triees`, on peut se définir les variables:

```
F1=donnees_brutes
F2=donnees_triees
```

il sera ensuite agréable de pouvoir référencer ces noms par `$F1` et `$F2`.

On peut également s'en servir pour faire des abréviations de commandes complètes:

```
CC='cc -o essai essai.c'
$ $CC
"essai.c", line 3: i undefined
$
```

Dans tous les exemples donnés les noms de variables sont en majuscules, mais cela n'est nullement une obligation.

1.9 La substitution de commande

La substitution de commande consiste à écrire une commande entourée du signe ` (back quote). On prendra garde à ne pas confondre ce caractère avec le caractère ' (simple quote). Sur rencontre d'une commande entourée du signe `, le shell exécute la commande et remplace le texte de la commande par sa sortie (le texte produit par son exécution). Exemple:

```
$ pwd
/users/lgi/systemeV/bernard/enseignement
$ D=`pwd`
$ echo $D
/users/lgi/systemeV/bernard/enseignement
$ ls texinfo*
texinfo texinfo-2 texinfo-4 texinfo-1 texinfo-3 texinfo.texinfo
$ FICS=`ls texinfo*`
$ echo $FICS
texinfo texinfo-1 texinfo-2 texinfo-3 texinfo-4 texinfo.texinfo
```

1.10 Les fichiers de commandes

1.10.1 Présentation

Il arrive souvent que l'on fasse du travail répétitif nécessitant de taper souvent les mêmes commandes enchainées. Il est alors agréable de pouvoir mettre une fois pour toutes ces commandes dans un fichier, et d'invoquer ensuite le fichier provoquant ainsi l'exécution des commandes comme si on les avaient tapées au terminal. Un tel fichier est dit "fichier de commandes" et s'appelle un *shell script* dans le jargon UNIX.

Supposons que l'on soit en train de mettre au point un programme de nom `gestion_notes.c`, on tape souvent les commandes de compilation et d'exécution. Il peut être interessant de mettre dans un fichier de nom `RUN` (par exemple) les lignes suivantes:

```
cc -o gestion_notes gestion_notes.c
gestion_notes
```

Pour exécuter ce fichier de commandes, on peut activer `sh` en le lui passant en paramètre, comme ceci:

```
$ sh gestion_notes
```

Ceci aura pour effet d'enchaîner la compilation et l'exécution du programme `gestion_notes`.

Il existe une autre méthode qui permet d'exécuter un fichier de commandes en l'invoquant de la même manière que l'on invoque un programme binaire. Il faut d'abord donner le droit d'exécution au fichier de commandes:

```
$ chmod +x RUN
$
```

Pour l'exécuter, il suffit ensuite de taper:

```
$ RUN
```

1.10.2 Passage de paramètres

Il est possible de passer des paramètres à un fichier de commandes. Dans un fichier de commandes les neuf premiers paramètres sont des variables portant les noms `1`, `2`, ... `9`. A l'intérieur du fichier de commandes on les référence donc par `$1`, `$2`, ... `$9`. Si nous voulions paramétrer le fichier de commandes `RUN`, nous l'écririons de la manière suivante:

```
cc -o $1 $1.c
$1
```

et nous pourrions l'invoquer par:

```
RUN gestion_notes
```

Il n'y a pas de moyen de référencer directement les paramètres au-delà du neuvième. On verra plus loin (cf commande interne `shift`, chapitre 4.10) comment il faut procéder. Par contre, il y a un moyen de référencer tous les paramètres: c'est la variable `*`.

```
$ cat VARETOILE
echo $*
$ VARETOILE f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12
f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12
$
```

Enfin, il faut savoir que le variable `0` est positionnée avec le nom du fichier de commandes qui est exécuté:

```
$ cat VARZERO
echo $0
$ VARZERO f1
VARZERO
$
```

1.10.3 Le shell d'exécution

Il existe dans le monde UNIX plusieurs shells. Les plus connus sont `sh`, `csh`, `ksh` et `tcsh`. Les deux shells de base sont `sh` et `csh`. Les shells `ksh` et `tcsh` sont respectivement des extensions de `sh` et `csh`. Quand on exécute un fichier de commandes simplement en invoquant son nom, le shell interactif que l'on utilise va lancer l'exécution d'un shell identique à lui-même pour exécuter le fichier de commandes. Il y a un moyen de forcer le shell d'exécution du fichier de commandes en mettant en tête de celui-ci la ligne :

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

ou:

```
#!/bin/csh
```

selon le shell désiré.

Comme on est jamais sûr, quand on écrit un fichier de commandes, du shell qui sera utilisé lors de son exécution, c'est une bonne habitude de toujours mettre une telle ligne en tête du fichier.

1.11 Le problème des métacaractères

En faisant le tour de l'ensemble des mécanismes qu'offre le shell, nous avons vu que chaque mécanisme repose sur un ensemble de métacaractères. La génération des noms de fichiers fonctionne avec `*`, `?` et `[]`, la redirection des entrées-sorties avec `>` et `<` etc ... Ceci offre le grand avantage que l'on peut exprimer `rm` *tous-les-fichiers* par `rm *`, mais comment faire quand je veux faire `rm` sur un fichier dont le nom est `*`? En d'autres termes, comment dénoter de manière littérale les métacaractères?

Il y a plusieurs solutions à ce problème.

Solution 1: le caractère `\` est un métacaractère signifiant que le caractère qui suit doit être pris de manière littérale. Par exemple, `\*` signifie `*` et non pas *tous-les-fichiers*. Autre exemple, `\a` signifie `a`. Attention, `\` n'a ce sens qu'a l'extérieur des chaînes entourées du caractère `'` ou du caractère `"`.

Solution 2: tous les caractères entourés par `'` (simple quote) doivent être pris de manière littérale. Par exemple, `'*?[]'` sigifie `*?[]` et `'\a'` signifie `\a`.

Solution 3: dans une chaîne de caractères entourée du signe `"` (double quote) tous les caractères doivent être pris de manière littérale sauf les caractères `$` `` ` `` `\` (dollar, back quote et back slash). Cela signifie que le shell continue à faire la substitution de variables, la substitution de commandes à l'intérieur d'une telle chaîne. D'autre part, le caractère `\` a le sens de rendre littéral le caractère suivant sur les SEULS caractères `$` `` ` `` `\` `"`. Un `\` suivi d'un *line-feed* sert de continuateur de ligne: les deux caractères sont ignorés.

```
$ ls
texinfo texinfo-1 texinfo-2 texinfo-3 texinfo-4 texinfo.texinfo
$ echo *
```

```
texinfo texinfo-1 texinfo-2 texinfo-3 texinfo-4 texinfo.texinfo
$ V=bonjour
$ echo '* $V  \* \$'
* $V \*  \$
$ echo "* $V  \* \$"
* bonjour  \* $
$
```

On fera attention à la subtile différence de fonctionnement du `\` selon le contexte dans lequel on le trouve: à l'exterieur ou à l'intérieur de chaînes entourées de `'` ou de `"`.

```
$ echo \a \$
a $
$ echo '\a \$'
\a \$
$ echo "\a \$"
\a $
$
```

Chapter 2

L'exécution de programme

2.1 Rappel sur le système de fichier UNIX

Le système de fichiers UNIX est bâti sur le modèle de l'arborescence. Les objets terminaux sont essentiellement les fichiers, et les nœuds de l'arborescence sont des répertoires. Un objet dans le système de fichiers est désigné par un *nom de chemin.* Un *nom de chemin* est un chemin dans l'arborescence, allant jusqu'à l'objet à désigner. Les noms de chemin peuvent être de deux types selon le répertoire qui est pris comme point de départ:

relatifs: le point de départ est le répertoire courant.

absolus: le point de départ est la racine de l'arborescence.

Par exemple, si le répertoire courant est `/usr/local`, l'objet `/usr/local/src/tex` peut être désigné par le nom de chemin relatif `src/tex` ou par le nom de chemin absolu `/usr/local/src/tex`.

2.2 Recherche du fichier à exécuter

Nous avons vu qu'une commande est formée du nom du fichier à exécuter, suivi des paramètres. En ce qui concerne le nom du fichier à exécuter, le shell distinge deux cas:

1. ce nom est formé de plusieurs composants (il comporte au moins une fois le séparateur `/`). Dans ce cas, le shell le considère comme un nom de chemin et l'interprète de la manière habituelle.

2. ce nom est formé d'un seul composant (il ne comporte pas le séparateur `/`). Dans ce cas, le shell interprète ce nom comme étant un nom de fichier à rechercher dans un ensemble de répertoires. La liste des noms de ces répertoires est contenue dans la variable PATH.

Exemple de variable PATH:

```
$ echo $PATH
/usr/local/bin:/usr/ucb:/usr/bin:/bin:.
$
```

On voit que par convention, dans la variable PATH, les répertoires sont séparés par le signe `:` (deux points). De plus, le signe `.` (point) désigne le répertoire courant.

Pour un utilisateur ayant une telle variable PATH, et demandant à exécuter le programme **awk**, le shell recherchera ce fichier dans les répertoires suivants, dans l'ordre indiqué:

1. `/usr/local/bin`
2. `/usr/ucb`
3. `/usr/bin`
4. `/bin`
5. le répertoire courant.

Le shell lancera l'exécution du premier fichier trouvé répondant aux deux conditions suivantes: s'appeler **awk** et avoir le droit d'exécution (droit `x`).

Il est classique qu'une installation redéfinisse de manière locale certaines commandes qui sont mises dans `/usr/local/bin`. Il suffit de mettre ce répertoire au début de son `PATH` pour que cela cache les commandes standards de même nom. Ces dernières restent cependant accessibles en utilisant une désignation absolue. Supposons que l'administrateur système ait généré une nouvelle version de **awk** qu'il ait mise dans `/usr/local/bin`, la version standard sera utilisable en l'invoquant par `/bin/awk`.

De la même manière, il est classique que les utilisateurs développent des outils personnels, ou fassent des versions personnelles de commandes standard, et les mettent dans un répertoire `bin` dans leur répertoire d'origine. Il leur suffit de mettre ce répertoire en tête de leur `PATH` pour pouvoir accéder à ces nouvelles commandes.

Supposons un utilisateur ayant `/users/bernard` comme répertoire d'origine, il lui suffit de faire:

```
$ PATH=/users/bernard/bin:$PATH
$
```

pour pouvoir accéder aisément à ses commandes personnelles. En effet:

```
$ echo $PATH
/users/local/bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin:.
$
```

2.3 Variable PATH et sécurité

Il y a une raison qui invite à mettre le répertoire courant en tête dans sa variable `PATH`, c'est la mésaventure suivante. La première chose que fait un débutant avec UNIX est de créer un petit programme et d'essayer de l'exécuter. Et comment appeller ce programme? `test`, bien sûr. Ce que cette personne ne sais pas, c'est qu'il existe dans le système une commande qui s'appelle `test`, elle réside sous `/bin`. Si `/bin` se trouve avant `.` dans la variable `PATH`, sur invocation de `test`, c'est la commande système qui sera exécutée et non pas le programme de l'utilisateur. On peut passer beaucoup de temps à comprendre pourquoi ce si petit programme de test refuse absolument de marcher ...

Ce serait cependant une énorme bréche dans la sécurité que de mettre le répertoire courant en tête et non pas en queue de sa variable PATH. En effet, on n'exécute pas toujours des programmes en restant dans son arborescence personnelle, parfois on fait `cd` pour aller ailleurs. Or certains répertoires offrent le droit `w` (écriture) à tout le monde, c'est le cas par exemple de `/tmp`. Imaginons que vous vous positionniez sur `/tmp` et que vous fassiez `ls` après qu'une personne mal intentionnée ait mis dans ce répertoire un fichier de nom `ls`, que va-t-il se passer? Si le répertoire courant est en tête de votre variable PATH vous allez exécuter le `ls` de `/tmp`. Et maintenant que se passe-t-il si le fichier `ls` est un fichier de commandes qui contient:

```
cd $HOME
rm -r *
```

Il faut donc résister à l'envie de mettre le répertoire courant en tête de son PATH et bien garder à l'esprit que quand on invoque une commande simplement par son nom, elle est recherchée dans un ensemble de répertoires et que le répertoire courant est le dernier. Pour exécuter un programme `test` du répertoire courant, il suffit de l'invoquer par `./test`.

2.4 Variable PATH et environnement BSD ou System V

Il existe deux grandes familles de systèmes UNIX: ceux de type BSD (de l'université de Californie à Berkeley), et ceux de type System V (de ATT). Ces deux familles ont un ancêtre commun, mais elles ont divergé [1] . Entre les deux familles il y a beaucoup de choses en commun, mais aussi beaucoup de petites différences horripilantes. Certains contructeurs qui veulent offrir à la fois un environnement BSD et System V, mettent dans l'arborescence de leur machine les commandes des deux versions. Il est traditionnel d'avoir sous `/usr/ucb` des commandes Berkeley, et de mettre dans `/usr/5bin` les commandes System V.

En jouant sur l'ordre des répertoires dans la variable PATH, on peut donner à l'utilisateur l'impression qu'il est dans un environnement Berkeley ou System V.

2.5 Passage de paramètre à un programme

Un programme qui désire récupérer les paramètres passés par le shell doit être programmé de la manière suivante:

```
void main(argc,argv)
int argc;
char *argv[];
{
...
}
```

`argc` a pour valeur 1 + le nombre de paramètres et `argv` est un tableau de pointeurs vers les paramètres contenus dans des chaînes de caractères. `argv[1]` pointe vers le premier paramètre, `argv[2]` pointe vers le deuxième paramètre, etc... et `argv[0]` pointe vers une chaîne de caractères contenant le nom du programme exécuté. Le paramètre `argc` a donc pour valeur le nombre d'éléments du tableau `argv`.

A titre d'exemple, voici un programme qui imprime les valeurs de `argc` et `argv`:

[1] L'une est devenue un singe, l'autre un homme

```
$ cat printargs.c
main(argc,argv)
int argc;
char *argv[];
{
int i;

printf("argc vaut %d\n",argc);
for (i = 0; i < argc; i++)
   printf("arg %d = |%s|\n",i,argv[i]);
}
$
```

Et maintenant une exécution de ce programme:

```
$ printargs hello + 34 %
argc vaut 5
arg 0 = |printargs|
arg 1 = |hello|
arg 2 = |+|
arg 3 = |34|
arg 4 = |%|
$
```

Montrons que les blancs ne sont pas significatifs (ceci vient de la phase dite d'*interprétation des blancs*:

```
$ printargs          hello          +            34                %
argc vaut 5
arg 0 = |printargs|
arg 1 = |hello|
arg 2 = |+|
arg 3 = |34|
arg 4 = |%|
$
```

Une autre exécution mettant en évidence l'effet des divers quotes:

```
$ printargs 'hello world'  \+  "bonjour a tous"
argc vaut 4
arg 0 = |printargs|
arg 1 = |hello world|
arg 2 = |+|
arg 3 = |bonjour a tous|
$
```

Exemple de paramètres nuls:

```
$ printargs '' hello ""
argc vaut 4
```

```
arg 0 = |printargs|
arg 1 = ||
arg 2 = |hello|
arg 3 = ||
$
```

2.6 Programmes et processus

Les utilisateurs ne raisonnent généralement qu'en terme de programme, mais il est nécessaire de bien comprendre la notion de processus, sinon un certain nombre de comportements du shell resteront incompréhensibles.

Un programme peut se définir comme étant un algorithme. Cet algorithme peut être sous la forme d'un programme source ou d'un programme compilé, prêt à être exécuté par la machine.

Un processus peut se définir comme étant un programme en cours d'exécution. Un programme donné n'existe généralement qu'en un exemplaire dans une machine, mais il peut donner naisance à un nombre indéterminé de processus qui l'exécutent en même temps.

Il y a une commande qui permet à un utilisateur de voir quels sont les processus que la machine exécute pour lui, c'est la commande `ps`.

```
$ ps
   PID TTY      TIME COMMAND
 28061 p9       0:19 sh
$
```

Ce processus est le shell `/bin/sh` que l'on utilise de manière interactive.

La commande `ps` admet un paramètre `-a` qui lui demande de lister tous les processus même ceux des autres utilisateurs. Si on l'utilise, on voit qu'il y a beaucoup de processus qui s'exécutent sur la machine, et qu'en particulier il y a beaucoup de processus qui exécutent un shell (`sh`, `csh`, `ksh` ou `tcsh`). Le système lance en effet un processus shell pour chaque utiisateur connecté à la machine.

2.6.1 Programmes binaires

Maintenant, lançons un programme en arrière plan, et essayons de le voir avec `ps`. Pour cela, il faut que son exécution soit un peu longue de manière à ce qu'on ait le temps de taper `ps` avant qu'il ne soit terminé. En voici un très long:

```
$ ls -lR / > fichiers &
28104
$ ps
  PID TT STAT TIME COMMAND
28061 p9      0:19 sh
28104 p9 D    0:02 ls -lR /
28105 p9 1    0:00 ps
$ kill 28104
$
```

On voit que la commande dont nous avons demandé l'exécution s'exécute dans un processus. Il y a toujours le processus `sh` et également `ps`. Sitôt le `ps` réalisé, nous avons tué le `ls` qui sinon aurait encombré inutilement la machine.

Exécutons maintenant un pipe-line:

```
$ ls -lR / | wc -l &
28134
$ ps
  PID TT STAT TIME COMMAND
28061 p9      0:19 sh
28134 p9 S    0:00 wc -l
28135 p9 R    0:00 ls -lR /
28136 p9 1    0:00 ps
$ kill 28134
```

On voit que les deux commandes qui participent au pipe-line s'exécutent chacune dans un processus. Ceci est une règle générale: le shell crée un processus pour toute commande que l'utilisateur demande d'exécuter.

2.6.2 Fichier de commandes

Essayons de comprendre ce qui se passe quand on exécute un fichier de commandes. Créons par exemple un fichier de commandes de nom `LS` et contenant simplement `ls -lR / > fichier` et exécutons-le en programme d'arrière plan:

```
$ chmod +x LS
$ LS &
28192
$ ps
  PID TT STAT TIME COMMAND
28061 p9      0:19 sh
28192 p9 S    0:00 sh
28193 p9 0    0:00 ls -lR /
28194 p9 1    0:00 ps
$ kill 28193
$
```

On voit que l'exécution des commandes du fichier `LS` se fait grace à un autre processus qui exécute `sh`. C'est le shell courant qui a lancé une autre invocation de lui-même pour exécuter le fichier de commandes. Un tel shell est appelé un sous-shell. Ce sous-shell est en attente de la terminaison de la commande `ls` qu'il a lancé. Par le fait qu'un fichier de commandes est interprété par un autre shell, un fichier de commandes peut être exécuté en arrière plan tout comme un programme binaire.

2.6.3 Fichier de commandes exécuté par le shell courant

Il est possible de faire exécuter un fichier de commandes par le shell courant et non pas par un sous-shell. Il faut pour cela utiliser la commande interne `.` (point). Exécuter un fichier de commandes de cette manière est nécessaire quand on désire que des affectations de variables réalisées dans le fichier de commandes affectent les variables du shell courant. Exemple:

```
$ cat modvars
#!/bin/sh
V1=1
V2=2
$ echo $V1 $V2

$ modvars
$ echo $V1 $V2

$ . modvars
$ echo $V1 $V2
1 2
$
```

Quand on réalise une modification au fichier `.profile` ayant pour but de changer les valeurs affectées aux variables gérées par `.profile`, si on veut que ces modifications affectent le shell courant, il faut exécuter le fichier `.profile` par la commande interne `.`.

```
$ echo $PATH
/usr/ucb:/usr/bin:/bin:.
$ vi $HOME/.profile   #   pour changer la valeur de PATH
$ . .profile
$ echo $PATH
/usr/local/bin:/usr/ucb:/usr/bin:/bin:.
```

2.6.4 En résumé

La manière dont le shell lance des processus peut être résumé dans les règles suivantes:

1. Tout programme est exécuté par un processus. En particulier, les différents composants d'un pipe-line sont exécutés par des processus différents, c'est ce qui assure leur exécution en parallèle.

2. Quand le shell lance l'exécution d'un programme en avant plan, il attend la fin de son exécution ; quand il lance un programme en arrière plan, il n'attend pas la fin de son exécution.

3. Quand un fichier de commandes est exécuté par une invocation de son nom, le shell crée un autre processus shell. C'est ce sous-shell qui interprètera les commandes du fichier.

4. Quand un fichier de commandes est exécuté par la commande interne `.` (point), il n'y a pas création d'un autre processus shell. C'est le shell courant qui interprète les commandes du fichier.

Chapter 3

La redirection des entrées-sorties

Pour comprendre l'ensemble des mécanismes de redirection offerts par le shell il est nécessaire de connaitre un minimum de choses sur la philosophie des services d' entrées-sorties offerts par le noyau UNIX.

3.1 Rappels sur le noyau UNIX

3.1.1 Les primitives d'entrées-sorties

Le noyau UNIX offre essentiellement deux primitives `read` et `write` qui ont toutes les deux le même interface:

```
int read(fd,buffer,nboct)
char *buffer;
int fd,nboct;

int write(fd,buffer,nboct)
char *buffer;
int fd,nboct;
```

fd est un périphérique logique.

buffer est un tableau de caractères

nboct est le nombre de caractères du tableau

Quand on exécute un opération de `read` ou `write`, le périphérique logique sur laquelle elle porte doit être affecté à un périphérique ou a un fichier. La primitive du noyau UNIX qui permet de réaliser une telle affectation est `open` (sur un périphérique ou un fichier), qui rend un numéro de périphérique logique que l'on référencera lors de toute entrée sortie ultérieure.

Le squelette d'un programme faisant des entrées-sorties est donc:

```
fd = open("data",O_RDONLY,0);
...
read(fd,buffer,nboct);
...
close(fd);
```

3.1.2 La notion de file descriptor

Dans le jargon UNIX ce que nous avons appelé un périphérique logique porte le nom de *file descriptor*. C'est un terme assez malheureux dans la mesure où ce n'est pas ce que l'on appelle communément un descripteur de fichier. Ce que l'on entend habituellement par descripteur de fichier correspond à la notion UNIX de *i-node*. Dans tout ce manuel nous nous tiendrons cependant à ce terme de file descriptor de manière à ce que le lecteur ne soit pas désorienté en lisant la documentation anglaise du système.

3.1.3 Périphériques physiques et fichiers

Le système UNIX unifie (du point de vue des primitives du système) la notion de périphérique physique et la notion de fichier. En effet, tout périphérique physique a un descripteur dans l'arborescence de fichiers. Ce descripteur peut être désigné au moyen d'un nom de chemin, comme un fichier. Traditionnelement, les descripteurs de périphériques résident dans le répertoire `/dev`. Un dérouleur de bandes pourra avoir un descripteur désigné par `/dev/mt0`, et un programme pourra affecter ce périphérique à un file descriptor en faisant:

```
fd = open("/dev/mt0",... );
```

les entrées-sorties ultérieures utilisant le file descriptor `fd` auront lieu sur ce dérouleur de bandes. Dans tout ce qui suit, pour éviter la lourdeur, nous ne diront pas "périphérique physique ou fichier", mais simplement "fichier".

3.1.4 L'héritage des file descriptors

Le mécanisme de création de processus UNIX possède la propriété de faire hériter au processus fils les file descriptors du père. Un programme n'est donc pas obligé d'affecter lui-même des fichiers aux file descriptors sur lesquels il réalise des entrées-sorties. Cette affectation peut être réalisée par son père. On peut donc concevoir deux programmes P1 et P2 qui collaborent de la manière suivante:

```
                P1                     |                  P2
                --                     |                  --
                                       |
fd = open("data",O_RDONLY,0)           |    read(fd,buffer,nboct)
                                       |    ...
creation d'un processus pour           |    close(fd)
executer P2                            |
```

3.2 Le shell et les filtres

Nous avons défini un filtre au chapitre 1.4 comme étant un programme qui lit ses données sur l'entrée standard (file descriptor 0), écrit ses résultats sur la sortie standard (file descriptor 1) et produit d'éventuels messages d'erreur sur l'erreur standard (file descriptor 2), selon le schéma suivant:

```
                              +-------+
                              |       | --->   sortie standard  (fd 1)
```

```
entree standard (fd 0)   ---> | prog  |
                              |       | --->   erreur standard  (fd 2)
                              +-------+
```

Nous pouvons maintenant préciser qu'un filtre n'affecte pas de fichier à ces trois file descriptors, il délègue cette responsabilité au programme qui l'appelle.

Le shell a donc sa propre statégie d'affectation des file descriptors quand il lance un programme. Si l'utilisateur ne précise rien, le shell transmet au programme appelé les file descriptors 0,1 et 2 affectés au terminal. Pour les cas où ces affectations par défaut ne conviennent pas à l'utilisateur, le shell dispose d'un mécanisme permettant de spécifier les affectations désirées. Ce mécanisme s'appelle la redirection des entrées-sorties.

3.3 Notations de redirections

Les notations de redirections peuvent apparaitre avant ou après la commande et dans n'importe quel ordre. Il est traditionnel de les mettre après la commande, et si on redirige entrée et sortie, de mettre la redirection de l'entrée, puis celle de la sortie, mais ce n'est nullement obligatoire. A la place de la façon traditionnelle:

```
cmd < data_in > data_out
```

on peut écrire indifférement:

```
< data_in cmd > data_out
< data_in  > data_out cmd
cmd > data_out < data_in
```

Il est possible de rediriger autant de file descriptors d'un programme qu'on le désire:

```
prog > resu 2> errors 3> fic 4> data
```

3.4 Redirection et substitutions

Le nom de fichier sur lequel porte une redirection subit la substitution de variables et de commande mais pas la génération de noms de fichiers. On peut donc écrire:

```
FICHIER=...
cmd > $FICHIER
```

ou:

```
cmd > `...`
```

Mais si on écrit:

```
cmd > fic*
```

quand bien même le modèle `fic*` correspondrait à un seul fichier, (par exemple `fichier_resultat`), il n'y aurait pas substitution de nom de fichier. Cette commande aura pour effet de mettre le résultat de `cmd` dans un fichier de nom `fic*`.

3.5 Redirection d'un flot d'entrée

Deux formes:

- < *nom-de-fichier* permet de rediriger l'entrée standard d'une commande (file descriptor 0) à partir de *nom-de-fichier*.
- *chiffre* < *nom-de-fichier* permet de rediriger le file descriptor *chiffre* d'une commande à partir de *nom-de-fichier*.

Dans la seconde forme, attention à ne pas laisser de blanc entre *chiffre* et le signe <. Exemple:

```
mail serge < lettre
cmd 3< data
```

Dans le premier exemple, l'entrée standard de `mail` est redirigée vers le fichier `lettre`, dans le second exemple, le programme `cmd` (qui est supposé lire sur le file descriptor 3) voit le flot de données associé à ce file descriptor redirigé vers le fichier `data`.

3.6 Redirection d'un flot de sortie

Deux formes de base, similiaires aux précédentes:

- > *nom-de-fichier* permet de rediriger la sortie standard d'une commande (file descriptor 1) à partir de *nom-de-fichier*.
- *chiffre* > *nom-de-fichier* permet de rediriger le file descriptor *chiffre* d'une commande à partir de *nom-de-fichier*.

si *nom-de-fichier* existe, son ancien contenu est perdu.

Il existe deux autres formes consistant à remplacer le signe > par le signe >>. Dans ce cas, si *nom-de-fichier* existe, son contenu n'est pas perdu : le flot redirigé va s'écrire à la fin du fichier.

exemples:

```
cmd > resu
cmd 2>errors
cmd > resu 2>errors
cmd 3>fic
```

Dans le premier cas, la sortie standard est redirigée sur le fichier `resu` et les erreurs sortent sur le terminal. Dans le second cas, la sortie standard sort sur le terminal, et les erreurs sont redirigées dans le fichier `errors`. Dans le troisième cas, la sortie standard ets redirigée vers le fichier `resu` et les erreurs sont redirigées vers le fichier `errors`. Dans le quatrième cas, le programme `cmd` (qui est supposé ecrire sur le file descriptor 3) voit le flot de données associé à ce file descriptor redirigé vers le fichier `fic`.

3.7 Redirection vers un file descriptor

Au lieu de rediriger un flot de données vers un fichier il est possible de le rediriger vers un autre file descriptor déjà ouvert. Ceci se fait par:

- `>&` *chiffre*
- *chiffre1* `>&` *chiffre2*
- `<&` *chiffre*
- *chiffre1* `<&` *chiffre2*

Attention à ne pas laisser de blanc entre le signe `&` et le *chiffre*. L'utilisation la plus courante de cette possibilité est de rediriger la sortie standard et la sortie erreur vers le même fichier. Exemple:

```
cc -o essai essai.c > log 2>&1
```

3.8 fermeture d'un flot de données

- `<&-` fermeture de l'entrée standard
- `>&-` fermeture de la sortie standard
- *chiffre* `>&-` fermeture du file descriptor *chiffre*

3.9 Redirection d'un flot d'entrée sur l'entrée du shell

Cette possibilité n'est interessante que dans les fichiers de commandes. Supposons que l'on désire écrire un fichier de commandes qui utilise un éditeur pour faire le type de modifications vu au chapitre 1.4. On peut mettre les commandes à l'éditeur:

```
%s/^[^0-9]*//
%s+car.*in+ / +
%s/ seconds//
w
```

dans un fichier que l'on appellera `ex.com` et ensuite faire `ex data < ex.com`. Ceci à l'inconvénient de nous obliger à gérer deux fichiers pour un seul programme ; il serait plus agréable de pouvoir mettre les commandes de l'éditeur dans le fichier de commandes lui-même. Cela est possible en redirigeant l'entrée standard à l'aide de la convention `<<` *marque-de-fin* Exemple:

```
$ cat cree_expr
#!/bin/sh
# shell
ex <<FIN
r data
%s/^[^0-9]*//
%s+car.*in+ / +
```

```
%s/ seconds//
w
q
FIN
$
```

La chaîne choisie comme marque de fin (ici `FIN`) est complètement arbitraire.

Dans les lignes qui sont ainsi passées au programme, le shell continue à réaliser la substitution de variables ainsi que la substitution de commandes. D'autre part, le caractère `\` rend littéral le caractère qui le suit dans le cas où ce caractère est l'un quelconque de ``\$` ``.

On peut profiter de cela pour paramétrer, dans l'exemple précédent, le nom du fichier dans lequel se font les substitutions.

```
$ cat CREE_EXPR
#!/bin/sh
# shell
ex <<FIN
r $1
%s/^[^0-9]*//
%s+car.*in+ / +
%s/ seconds//
w
q
FIN
$ chmod +x CREE_EXPR
$ CREE_EXPR data
```

Il peut arriver que les substitutions de variables et de commandes que le shell réalise entre le `<<` *marque-de-fin* et *marque-de-fin* soit génantes. Il y a un moyen d'inhiber ces substitutions: il suffit que l'un quelconque des caractères composant le mot *marque-de-fin* soit quoté. Si la *marque-de-fin* est le mot `FIN` on peut donc utiliser indifféremment `\FIN F\IN FI\N` ou `'FIN'`. Plutôt qu'utiliser anarchiquement n'importe quelle notation, on utilisera de préférence la dernière qui est la plus lisible.

Voici un fichier de commandes qui met en évidence l'inhibition des mécanismes de substitution:

```
$ cat ESSAI
#!/bin/sh
V=hello!
echo premier cas
cat > resu <<FIN
$V
`date`
FIN
cat resu
echo deuxieme cas
cat > resu <<'FIN'
$V
```

```
`date`
FIN
cat resu
```

Et voici le résultat:

```
$ ESSAI
premier cas
hello!
Mon Oct 21 12:44:07 MET 1991
deuxieme cas
$V
`date`
$
```

3.9.1 Applications classiques de ce mécanisme

Il y a deux applications classiques à ce mécanisme: la création de fichier d'archive, et la création de fichier avec substitution de mots.

Création de fichier archive

Il est parfois intéressant de remplacer un ensemble de petits fichiers par un seul fichier d'archive: soit pour la conservation, et le but est le gain de place disque, soit pour la distribution de logiciel, et le but est la simplicité de la distribution. Il existe pour résoudre ce problème deux commandes `shar` et `unshar`. Leur but est de créer ou d'exploiter des fichiers qui ont la structure suivante:

```
#!/bin/sh
cat > fichier1 <<'FIN-DE-FICHIER'
ligne 1 de fichier 1
ligne 2 de fichier 1
FIN-DE-FICHIER
cat > fichier2 <<'FIN-DE-FICHIER'
ligne 1 de fichier 2
ligne 2 de fichier 2
FIN-DE-FICHIER
```

Quand on exécute ce fichier de commande, il crée `fichier1` et `fichier2`.

Création de fichier paramétré

Il arrive parfois qu'à partir d'un fichier modèle on ait à créer un fichier paramétré selon les valeurs de certaines variables. Cela peut se faire en exploitant le mécanisme de substitution de variable réalisé lors d'une redirection d'entrée. Voici un exemple de modèle de lettre:

```
#!/bin/sh
# PERSONNE peut valoir 'Monsieur' ou 'Madame'
PERSONNE=Monsieur
# SENTIMENT peut valoir 'le regret' ou 'la joie'
```

```
SENTIMENT='le regret'
# AVOIR peut valoir 'avez' ou "n'avez pas"
AVOIR="n'avez pas"
cat > lettre <<FIN-DE-FICHIER
 $PERSONNE,
J'ai $SENTIMENT de vous annoncer que vous $AVOIR reussi votre examen.
Veuillez agreer, ...
FIN-DE-FICHIER
```

3.10 Le pseudo-périphérique /dev/null

Le peudo-périphérique `/dev/null` agit comme une source vide pour des lectures, et comme un puits sans fond pour des écritures. En d'autres termes, un programme lisant `/dev/null` recevra immédiatement une indication de *fin-de-fichier*, et toutes les écritures faites dans `/dev/null` seront perdues.

Exemple d'utilisation de `/dev/null` en sortie: on veut juste tester l'existence d'une chaîne de caractère dans un fichier avec `grep`, sans s'interesser à la ligne elle-même si elle existe:

```
if  grep chaine fichier > /dev/null
then
   echo la chaine existe
else
   echo "la chaine n'existe pas"
fi
```

Exemple d'utilisation de `/dev/null` en entrée: `cp /dev/null fichier` est une des manières classiques de créer un fichier de taille nulle.

3.11 Redirection sans commande!

Bien qu'il paraisse aberrant de vouloir rediriger les entrées-sorties d'une commande qui n'exite pas, le shell permet que l'on fasse:

```
> fichier
```

ceci a pour effet de créer un fichier de taille nulle si il n'existait pas, ou de le tronquer à une taille nulle si il existait déja.

On peut également écrire:

```
< fichier
```

ceci a pour effet de tester l'existence et le droit de lecture sur le fichier `fichier`.

Voici un exemple d'interaction dans un répertoire comportant un fichier `sh.info` et pas de fichier `qqq`:

```
$ < sh.info
$ echo $?
0
$ < qqq
```

```
qqq: cannot open
$ echo $?
1
$
```

Cette possibilité fait double emploi avec `test -r fichier` (Voir chapitre 7.4).

3.12 Entrées-sorties et fichier de commandes

Nous avons vu qu'un fichier de commandes est exécuté par une nouvelle instance du shell. Ce shell lit les commandes et les exécute. L'ensemble des entrées standard des commandes ainsi exécutées forme l'entrée standard du fichier de commandes, et l'ensemble des sorties standard des commandes forme la sortie standard du fichier de commandes. Ceci a pour conséquence que l'on peut rediriger les entrées-sorties d'un fichier de commandes exactement comme celles d'une commande.

Exemple: créons un fichier de commandes qui substitue dans son entrée standard les occurrences de son premier paramètre par son second paramètre:

```
$ cat SUBS
#!/bin/sh
sed "s/$1/$2/"
$
```

Maintenant, exécutons-le:

```
cat data
bernard
jean
serge
$ SUBS serge pierre < data
bernard
jean
pierre
$ SUBS serge pierre < data > resu
$
```

De la même manière, on peut créer un pipe-line dont l'un des composants est un fichier de commandes. On voit donc que du point de vue des entrées-sorties, les fichiers de commandes se comportent comme les programmes exécutables.

Chapter 4

Les variables

4.1 Les noms des variables

Les noms des variables peuvent être composés:

- soit d'une suite de lettres, de chiffres et du caractère `_`.
- soit d'un chiffre
- soit de l'un quelconque des caractères `* @ # ? - $ !`.

Le premier cas correspond au variables créées par l'utilisateur, le deuxième cas correspond aux paramètres des fichier de commandes, le troisisème cas correspond à un ensemble de variables gérées par le shell.

4.2 Déclaration et types des variables

Il n'est pas nécessaire de déclarer une variable avant de l'utiliser, comme on est obligé de le faire dans les langages de programmation classique. Les objets possédés par les variables sont d'un seul "type" : la chaîne de caractères.

4.3 Affectation d'une valeur à une variable

4.3.1 La syntaxe

La syntaxe d'une affectation est la suivante:
nom-de-variable = *chaîne-de-caractères* Exemples

```
$ V1=1
$ V2=2
$
```

Attention aucun blanc n'est admissible autour du signe =:

```
$ V1= a
a: not found
$ V1 =a
V1: not found
$
```

4.3.2 Affectation d'une chaîne vide

On peut affecter une chaîne vide à une variable de 3 manières différentes:

```
$ V1=
$ V2=''
$ V3=""
$
```

4.3.3 Affectation et interprétation des blancs

Si la *chaîne-de-caractères* à affecter à la variable comporte des blancs, il faut en faire un seul mot à l'aide des quotes.

```
$ MESSAGE='Bonjour a tous'
$
```

Les caractères "blancs" sont par défaut l'espace, la tabulation et le *line-feed*, on peut donc affecter une chaîne formée de plusieurs lignes.

```
$ MESSAGE='Bonjour a tous
--------------'
$echo $MESSAGE
Bonjour a tous
--------------
$
```

4.3.4 Affectation et substitutions

Dans la chaîne qui est affectée à la variable, le shell réalise la substitution de variable et la substitution de commandes, mais pas la substitution de noms de fichiers.

```
$ DATE=`date`
$ PERSONNES=$USER
$ FICHIERS=*
```

La variable `DATE` va mémoriser la date courante, la variable `PERSONNE` va recevoir la valeur de la variable `USER`, mais la variable `FICHIER` ne va pas recevoir comme valeur la liste des noms des fichiers, mais le caractère `*`.

4.4 Toutes les techniques de substitution

Il y a au total 6 variantes à la substitution de variables:

`$` *variable*	substitue la valeur de *variable*
`${`*variable*`}`	substitue la valeur de *variable*
`${`*variable* - *mot*`}`	si *variable* a une valeur substituer cette valeur sinon substituer *mot*
`${`*variable* = *mot*`}`	si *variable* n'a pas de valeur y affecter *mot* ensuite substituer *mot*
`${`*variable* ? *mot*`}`	si *variable* a une valeur substituer cette valeur sinon imprimer *mot* et sortir du shell
`${`*variable* + *mot*`}`	si *variable* a une valeur substituer cette valeur sinon substituer le vide

Il est licite de demander la substitution d'une variable à laquelle aucune valeur n'a été affectée: la substitution rendra une chaîne vide. Si ce comportement par défaut n'est pas satisfaisant, on peut positionner l'option `-u` à l'aide de la commande interne `set` (Voir chapitre 6.13).

```
$ echo :$toto:
::
$ set -u
$ echo :$toto:
toto: parameter not set
$
```

4.5 Substitution de variables et interprétation des blancs

Le shell procède d'abord à la sunstitution de variables et ensuite à l'interprétation des blancs (le découpage en mots). Ceci a pour conséquence qu'il n'est pas nécessaire que la valeur d'une variable constitue un paramètre entier de commande. Le texte résultant d'une substitution de commande peut constituer une fraction d'un paramètre de commande.

```
FROM=bernard
TO=jean
sed s/$FROM/$TO/ data
```

4.6 Substitution de variables et mécanisme de quote

4.6.1 Premier problème

Supposons que nous gérions un annuaire téléphoqnique sous la forme d'un fichier dont les lignes ont la structure suivante:

```
Jacques Eudes 9056
Serge Rouveyrol 4567
```

on a crée un fichier de commandes `notel` pour rechercher le numero de telephone de quelqu'un:

```
$ cat notel
#!/bin/sh
grep -i $1 $HOME/lib/annuaire
$ notel serge
Serge Rouveyrol 4567
$
```

Essayons manintenant de l'utiliser en lui passant en paramètre un nom complet. Comme on a bien compris ce qui est expliqué au chapitre 1.2.2, on entoure le nom par des `'`:

```
$ notel 'serge rouveyrol'
rouveyrol: No such file or directory
annuaire:Serge Rouveyrol 4567
```

On est tombé précisement sur le problème que l'on cherchait à eviter. Il faut savoir que le shell procede à la substitution de variables et ensuite à l'interprétation des blancs. La chaîne **serge rouveyrol** a bien été passée à **notel** comme un seul paramètre, mais lors du traitement de la commande **grep**, le shell à remplacé **$1** par sa valeur et a ensuite découpé la ligne en param'tres, passant ainsi **serge** et **rouveyrol** comme deux paramètres à **grep**.

Pour résoudre le problème, il faut que dans le fichier de commandes, **$1** soit pris comme un seul paramètre de **grep**. Il faut donc écrire:

```
grep -i "$1" $HOME/lib/annuaire
```

4.6.2 Second problème

Supposons que nous écrivions un fichier de commandes qui pose une question à l'utilisateur et teste la réponse. On a écrit:

```
echo "Voulez vous continuer ? [non]"
read $reponse
if [ $reponse = oui ]
then
...
fi
```

On pose une question à l'utilisateur qui peut répondre par oui, non ou retour chariot. Le non entre crochet signifie que si l'utilisateur ne répond rien, (il tape simplement retour chariot), la réponse sera considérée comme étant non. Ce fichier de commandes fonctionne correctement si on répond oui ou non, mais on a le message d'erreur:

```
test: argument expected
```

si on répond par retour chariot. En effet, dans ce cas la variable *réponse* a pour valeur la chaîne vide, et seulement deux paramètres sont passés à **test** au lieu de trois. Pour résoudre le problème, il faut obliger le shell à passer **$reponse** en paramètre même si il est vide. Se rappelant ce qui a été dit au chapitre **??** au sujet des paramètres nuls, on écrira:

```
if [ "$reponse" = oui ]
```

et le problème sera résolu.

4.7 La commande interne set et variables

La commande interne `set` peut être utilisée de plusieurs manières différentes. Elle peut servir à lister les variables connues du shell à un instant donné. Elle permet également d'affecter des valeurs aux variables `1`, `2`, ...

4.7.1 Lister les variables

La commande `set` sans paramètre permet d'obtenir une liste classée par ordre alphabétique de l'ensemble des variables avec leurs valeurs.

4.7.2 Affectation de valeur aux variable 1 2 ...

La commande interne `set` permet d'affecter ses paramètres aux variables `1`, `2` etc...

Exemple:

```
$ set bonjour a tous
$ echo $1
bonjour
$ echo $2
a
$ echo $3
tous
$
```

Cette possibité est intéressante pour découper en mots le contenu d'une variable. Voici un fichier de commandes qui extrait la deuxième colonne d'un fichier supposé contenir une information organisée en colonnes:

```
while read ligne
do
   set $ligne
   echo $2
done
```

Malheureusement, `set` est un fourre-tout incroyable, il y a une autre utilisation possible: positionner des options du shell. On peut écrire par exemple, `set -x` pour que toute commande soit imprimée avant d'être exécutée. Cette option est parfois utile pour mettre au point des fichiers de commandes.

Mais dans l'utilsation qu'on en fait ici, si par malheur dans le flot d'entrée il y a une ligne qui commence par le caractère `-`, l'exécution de `set $ligne` va résulter en un message d'erreur du shell.

Il y a un moyen d'éviter ce problème, mais ce n'est hélas pas le même selon qu'on a un shell BSD ou SystemV. En BSD, il faut écrire:

```
set - $ligne
```

et en System V:

```
set -- $ligne
```

ceci va prévenir `set` qu'il ne faut pas interpréter le reste de ses arguments comme des options, mais comme des mots à affecter à `1`, `2`, ...

4.8 La commande interne `readonly`

Si on désire protéger des variables de toute affectation ultérieure, on peut les déclarer en lecture seulement avec la commande interne `readonly`. Invoquer `readonly` sans paramètre permet d'obtenir la liste de toutes les variables en lecture seulement. Exemple:

```
$ V1=1 V2=2
$ readonly V1 V2
$ V1=3
V1: is read only
$ readonly
readonly V1
readonly V2
$
```

4.9 Les variables affectées par le shell

Les variables affectées automatiquement par le shell sont les suivantes:

`#`	nombre de paramètres d'un fichier de commandes
`-`	options courantes du shell
`?`	valeur retournée par la dernière commande exécutée
`$`	numéro de processus du shell
`!`	numéro de processus de la dernière commande exécutée en arrière plan
`0`	le nom du fichier de commandes
`0` 1...9	paramètres d'un fichier de commandes ou affectés par `set`
`*`	l'ensemble des paramètres d'un fichier de commandes
`@`	l'ensemble des paramètres d'un fichier de commandes chacun étant protégé par `"`

Quelques commentaires:

1. La variable `#` est pratique pour vérifier la validité de l'appel d'un fichier de commandes. On donne en exemple le début d'un fichier de commandes qui vérifie qu'on lui passe bien deux paramètres.

   ```
   #!/bin/sh
   if [ $# -ne 2 ]
   then
           echo "Il faut les parametres ... et ..."
           exit 1
   fi
   ```

   On fera attention au fait que `#` contient le nombre de paramètres, alors qu'en langage C, dans `main(arg\c{c}argv)`, `argc` contient le nombre de paramètres + 1.

2. La variable `$` est très pratique pour créer des noms de fichiers temporaires dont on soit sûr de l'unicité. On pourra écrire par exemple *commande* `> /tmp/gestion.$$`. Si plusieurs utilisateurs exécutent le fichier de commandes en même temps, on est sûr qu'il n'y aura pas de collision entre les noms des fichiers temporaires.

3. La variable `*` est très pratique pour référencer la liste des paramètres sans se préoccuper de leur nombre. Supposons que l'on crée une commande `LPRTEX` qui a pour but d'appeler `lpr` avec l'argument `-d` On pourra écrire:

   ```
   $ cat LPRTEX
   lpr -d $*
   ```

```
$ LPRTEX fic1 fic2 fic3
$
```

La différence entre `*` et `@` est subtile:

`$*` vaut `$1 $2 $3 ...`

`"$*"` vaut `" $1 $2 $3 ... "`

`$@` vaut `$1 $2 $3 ...`

`"$@"` vaut `"$1" "$2" "$3" ...`

Le fichier de commandes suivant et son exécution feront comprendre la différence:

```
$ cat printparams
#!/bin/sh
echo "*** phase 1 ***"
for i in $*
do
echo $i
done

echo "*** phase 2 ***"
for i in "$*"
do
echo $i
done

echo "*** phase 3 ***"
for i in $@
do
echo $i
done

echo "*** phase 4 ***"
for i in "$@"
do
echo $i
done

$ printparams a 'b c'
*** phase 1 ***
a
b
c
*** phase 2 ***
a b c
*** phase 3 ***
a
```

```
b
c
*** phase 4 ***
a
b c
$
```

4.10 La commande interne shift

La commande interne `shift` a pour effet d'affecter la valeur de `$2` à `$1`, la valeur de `$3` à `$2`, etc... et d'affecter la valeur du dixième paramètre (qui jusque là était inaccessible) à `$9`. De surcroît, `shift` met à jour la variable `$#` en diminuant sa valeur de 1. La commande `shift` permet donc d'accéder aux paramètres au delà du neuvième. Exemple:

```
#!/bin/sh
# recuperation de 11 parametres
P1=$1 P2=$2 P3=$3 P4=$4 P5=$5 P6=$6 P7=$7 P8=$8 P9=$9
shift
P10=$9
shift
P11=$9
```

La commande interne `shift` est également agréable pour traiter les paramètres optionnels d'un fichier de commandes. Soit le fichier de commandes `cmd` que l'on peut appeler de deux manières différentes: soit `cmd` *fichier*, soit `cmd -a` *ficher*. On pourra analyser les paramètres de la manière suivante:

```
#!/bin/sh
OPTION=
if [ $# -eq 0 ]
then
   echo "Usage: cmd [-a] fichier"
   exit 1
fi
if [ $1 = -a ]
then
   OPTION=a
   shift
fi
FICHIER=$1
```

4.11 Les variables formant l'environnement

4.11.1 Rappel sur l'exécution de programmes UNIX

L'appel système permettant de lancer l'exécution d'un programme binaire contenu dans un fichier est `execve`. Cet appel permet de passer au programme à exécuter non seulement des paramètres, mais également un ensemble de couples (noms-de-variable,valeur) que l'on appelle l'environnement du programme. Un couple (nom-de-variable,valeur) est réalisé sous la forme d'une chaine de caractères composée de *nom-de-variable* suivi du signe = suivi de la *valeur*.

L'interface de `execve` est le suivant:

```
void execve(name,argv,envp)
char *name;
char *argv[];
char *envp[];
```

`name` est le nom du fichier contenant le programme à exécuter. `argv` est un tableau de pointeurs vers les paramètres. `env` est un tableau de pointeurs vers les variables.

Un programme qui désire récupérer son environnement doit être programmé de la façon suivante:

```
main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
```

On remarque que l'interface prevoit d'indiquer le nombre d'élements du tablau `argv` : il s'agit de la variable `argc`. Par contre, il ne prévoit pas d'indiquer le nombre d'éléments du tableau `envp`. La fin de ce tableau est indiquée par un elément à `NULL`.

Voici un programme qui imprime son environnement d'exécution:

```
main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
int i = 0;
char **p = envp;

while (*p != NULL)
   printf("variable # %d: %s\n",i++,*p++);

}
```

Et voici un résultat de son exécution:

```
variable # 0: HOME=/users/bernard
variable # 1: PATH=/users/bernard/bin:/usr/ucb:/bin:/usr/bin:.
variable # 2: LOGNAME=bernard
variable # 3: SHELL=/bin/sh
variable # 4: MAIL=/usr/spool/mail/bernard
variable # 5: TERM=xterm
variable # 6: TZ=MET-1EET-2;85/02:00:00,267/02:00:00
variable # 7: HOST=ensisps1
variable # 8: VISUAL=/usr/ucb/vi
```

4.11.2 Les variables exportées

C'est le shell qui exécute l'appel système **execve**, c'est donc lui qui choisit les variables qui sont mises dans l'environnement du programme exécuté. Le shell **sh** gère deux types de variables: les variables dites exportées, et les variables dites non exportées. Quand il exécute un **execve**, le shell met dans l'environnement du programme l'ensemble de ses variables exportées. Pour exporter une variable, il faut exécuter la commande interne **export** en lui passant en paramètre la liste des variables à exporter. Exemple:

```
VISUAL=/usr/ucb/vi; export VISUAL
```

A partir de ce moment, tout programme exécuté aura `VISUAL` dans son environnement.

Pour lister l'ensemble des variables exportées à un instant donné, il faut exécuter **export** sans argument. ATTENTION: pb.

4.12 Une autre méthode pour passer des variables

On peut rajoutter des variables à l'environnement d'un programme en mettant en tête de son invocation des couples *nom* = *valeur*. Ces variables seront mises dans l'environnement du programme exécuté et ne seront pas connues du shell courant. Prenons un exemple. Le programme **epelle** est un vérificateur orthographique qui utilise un dictionnaire qui lui est propre, plus éventuellement un dictionnaire personnel de l'utilisateur. Si l'utilisateur a un tel dictionnaire, il doit positionner la variable `DICOPLUS` avec le nom du fichier dictionnaire. On peut donc appeler le vérificateur de la manière suivante:

```
DICOPLUS=$HOME/lib/dico     epelle document.tex
```

4.13 Gestion des variables

Le shell interactif de l'utilisateur a des variables. Quand un fichier de commandes est exécuté, il est exécuté par une autre instanciation du shell, qui a elle aussi des variables. Ce fichier de commandes peut éventuellement appeler un autre fichier de commandes , exécuté par une autre instance du shell qui aura elle aussi des variables etc... Au cours de l'exécution d'un fichier de commandes on a donc un ensemble de shells qui se sont appelés mutuellement, chacun ayant des variables. Dans ce chapitre nous allons nous interesser à la façon dont ces variables sont gérées.

Règle1

Toutes les variables sont locales à un shell: qu'elles soient exportées ou non, elles ne peuvent pas être modifiées par un sous-shell. Exemple:

```
$ cat setxy
#!/bin/sh
x=10
y=20
$ x=1
$ y=2 ; export y
```

```
$ setxy
$echo $x $y
1 2
$
```

Explication: dans le shell qui a exécuté le fichier de commandes il y a eu instanciation d'une variable `x` qui a recu la valeur 2. L'existence de cette variable a disparue quand le shell s'est terminé.

Règle2

Les variables exportées sont instanciées dans tous les sous-shells avec leur valeur. Exemple:

```
$ cat printxy
#!/bin/sh
echo "variables de printxy: x = $x y = $y"
printxy2
$ cat printxy2
#!/bin/sh
echo "variables de printxy2: x = $x y = $y"
$ x=1 ; export x
$ y=2 ; export y
$ printxy
variables de printxy: x = 1 y = 2
variables de printxy2: x = 1 y = 2
$
```

Règle3

Les variables exportées sont instanciées avec la valeur qu'elles ont dans le dernier shell qui les exporte.

Modifions `printxy` pour lui faire modifier les valeur de `x` et `y` et exporter seulement `y`.

```
$ cat printxy
#!/bin/sh
echo "variables de printxy: x = $x y = $y"
x=10
y=20 ; export y
printxy2

$ cat printxy2
#!/bin/sh
echo "variables de printxy2: x = $x y = $y"

$ x=1; export x
$ y=2 ; export y
$ printxy
variables de printxy: x = 1 y = 2
variables de printxy2: x = 1 y = 20
```

```
$
```

4.14 Les variables utilisées par le shell

Le shell utilise un certain nombre de variables exactement comme n'importe quel programme peut le faire, c'est à dire pour modifier son comportement en fonction de la valeur de ces variables.

4.14.1 Les variables `PS1` et `PS2`

Les variables `PS1` et `PS2` ont pour valeur les chaînes de caractères utilisées respectivement en temps que prompt primaire et prompt secondaire par le shell lorsqu'il est utilisé de manière interactive. Ces deux variables ont comme valeur par défaut `$`*espace* et `>`*espace*. Le prompt secondaire est affiché par le shell en début de chaque ligne lorsque l'utilisateur est en train de taper une commande tenant sur plusieurs lignes.

Exemple d'interaction où apparait `PS2`:

```
$ for fichier in texinfo*
> do
> echo $fichier
> done
texinfo
texinfo-1
texinfo-2
texinfo-3
texinfo-4
texinfo.texinfo
$
```

Les variables `PS1` et `PS2` peuvent être modifiées comme n'importe quelle variable. Par exemple:

```
$ echo hello
hello
$ PS1="j'ecoute: "
j'ecoute: echo hello
hello
j'ecoute:
```

4.14.2 La variable `HOME`

Cette variable est positionnée par le programme `login`. La valeur qui lui est donnée est celle du champ *répertoire d'origine* qui se trouve dans le fichier `/etc/passwd`. On rappelle que ce fichier contient des lignes ayant la structure suivante:

```
serge:Kk43cgXNNuYSo:40:226:Serge Rouveyrol,113B,4879,:/users/serge:/bin/sh
```

Chaque ligne est formées de 7 champs séparés par des `:` (deux points). L'avant dernier champ est le répertoire d'origine de l'utilisateur.

La commande `cd` (change directory) est une commande interne du shell. Quand on l'utilise sans lui donner de paramètre, le shell l'interpréte comme signifiant `cd` *répertoire-d'origine.* Il utilise alors la valeur de `HOME` pour satisfaire cette requête.

4.14.3 La variable PATH

La variable `PATH` est elle aussi affectée par le programme `login` avant d'appeler le shell. Nous avons vu (chapitre 2.2) à quoi elle servait.

4.14.4 La variable IFS

Le nom `IFS` siginifie "Internal Field Separator". Cette variable a pour valeur l'ensemble des caractères à considérer comme des blancs lors du découpage d'une ligne de commande en mots. Par défaut cette variable contient les 3 caractères blanc, tabulation et line-feed. Pour voir la valeur de cette variable, `echo` sur le terminal n'est pas suffisant (elle ne contient que des caractères "blancs").

```
$ echo $IFS

$ echo "$IFS" > ifs
$ od -x ifs
0000000  2009 0a0a
0000004
$
```

Blanc, tabulation et *line-feed* ont respectivement comme valeur héxadécimale 20, 09 et 0a. Le dernier 0a est le `line-feed` que le shell imprime pour terminer `echo`.

Dans les fichier de commandes , il est parfois utile de changer la valeur de `IFS` en conjonction avec la commande interne `set`. On donne ci-dessous un exemple où l'on donne à `IFS` la valeur : (deux points) pour que la commande `set` découpe une ligne de `/etc/passwd` en ses différents champs.

```
$ SERGE=`grep '^serge' /etc/passwd`
$ echo $SERGE
serge:Kk43cgXNNuYSo:40:226:Serge Rouveyrol,113B,4879,:/users/serge:/bin/sh
$ IFS=:
$ set $SERGE
$ echo $1
serge
$ echo $2
Kk43cgXNNuYSo
$ echo $3
40
$
```

4.15 Opérations sur les variables

Dans un fichier de commandes on a parfois besoin de réaliser certaines opérations sur les valeurs des variables. Le shell ne contient que trois mécanismes permettant de réaliser des opérations sur les variables:

1. la commande interne set permet de découper une chaîne de caractères en “mots”. Le caractère servant de séparateur de mots peut être paramétré par la variable `IFS`.

2. la structure `case` permet de tester si la valeur d'une variable est conforme à un modèle.

3. la substitution de variable permet de réaliser la concaténation :

   ```
   listefic="$listefic $fic"
   ```

Si on a à réaliser un traitement qui ne rentre pas dans l'un des cas précédents, il faut le faire avec les commandes.

Supposons qu'une variable `FICHIER` contienne un nom de fichier se terminant en `.c` et que nous voulions remplacer cette terminaison par `.o`. Cela se fera en soumettant la valeur de `FICHIER` à l'appel suivant de `sed` : `sed 's/\.c$/.o/'`.

```
SOURCE=toto.c
OBJET=`echo $FICHIER | sed 's/\.c$/.o/'`
```

On voit qu'on a résolu le problème par un combinaison des mécanismes de *substitution de commande*, *substitution de variable*, *pipe* et *quote* (pour que le `$` dans la commande `sed` soit pris de manière littérale).

La diversité des commandes UNIX est énorme, mais pour réaliser des traitements sur les variables, les plus interessantes sont sans doute:

- `sed` pour sa capacité de substitution.
- `awk` pour sa capacité à traiter les champs d'une chaîne de caractères.
- `cut` pour sa capacité à couper selon des numéros de colonne.
- `tr` pour sa capacité de transformation d'un caractère par un autre.
- `expr` pour sa capacité de calculs arithmétiques.
- `wc` pour sa capacité de comptage.

Chapter 5

Les structures de contrôle

5.1 Rappel sur les codes de retour de programmes UNIX

Le noyau UNIX a prévu qu'un programme puisse, après son exécution, transmettre une valeur à celui qui l'a lancé. En dernière instruction, un programme exécute `exit(n)`, où `n` est la valeur à transmettre, et l'appelant exécute `wait(&status)`, où `status` est une variable dans laquelle le noyau UNIX met la valeur transmise.

Ces valeurs ainsi transmises servent de *code de retour* des programmes: elles servent à donner une indication sur la façon dont s'est déroulée l'exécution du programme. Par convention, la valeur 0 signifie que l'exécution du programme s'est déroulée sans erreur. Une valeur différente de zéro signifie qu'il y a eu une erreur, la valeur étant un code choisi par le programmeur de l'application.

5.2 La gestion des codes de retour par le shell

5.2.1 Code de retour d'un programme

Quand le shell lance l'exécution d'un programme en avant plan, il attend la fin de l'exécution par un `wait(&status)`. Il est ainsi prévenu du code de retour du programme. Le shell affecte ce code de retour à la variable `?`, que l'on peut manipuler comme n'importe quelle variable. Faisons un essai avec la commande `grep`, qui rend une indication d'erreur quand elle ne trouve pas la chaîne qu'on lui demande de rechercher:

```
$ grep toto /etc/passwd
$ echo $?
1
$ grep serge /etc/passwd
serge:Kk43cgXNNuYSo:40:226:Serge Rouveyrol,113B,4879,:/users/serge:/bin/sh
$ echo $?
0
$
```

Quand le shell lance l'exécution d'un programme en arrière plan, il n'attend pas la fin de l'exécution et ne peut donc pas connaitre le code de retour. Dans ce cas, la valeur de la variable `?` reflète la manière dont le lancement en programme d'arrière plan s'est passé:

- Si le lancement a pu être réalisé (les fichiers de redirection des entrées-sorties existaient bien, etc...), la variable ? vaut 0.
- Si le lancement n'a pu être réalisé, la variable ? vaut 1.

```
$ grep toto /etc/passwd &
12275
$ echo $?
0
$ grep toto < /jj
/jj: cannot open
$ echo $?
1
$
```

5.2.2 Code de retour d'un pipe-line

On rappelle qu'un pipe-line est un ensemble de commandes connectées par des pipes. Par exemple:

```
$ grep -i serge  /etc/passwd | wc -l
       7
$
```

Le code de retour d'un pipe-line est le code de retour de la dernière commande du pipe-line.

Pour expérimenter, réalisons un programme (`erreur.c`) dont le but est de se terminer en rendant un code d'erreur passé en paramètre par l'utilisateur:

```
$ cat erreur.c
main(argc,argv)
int argc;
char *argv[];
{
exit(atoi(argv[1]));
}

$  erreur 34 | erreur 73
$ echo $?
73
$
```

5.2.3 Code de retour d'une séquence de commandes

Le code de retour d'une séquence de commandes est le code de retour de la dernière commande de la séquence.

```
$ erreur 34 ; erreur 72 ; erreur 89
$ echo $?
89
$
```

5.3 La structure &&

Deux pipe-lines peuvent être connectés par la structure **&&**. Sa sémantique est la suivante: si le premier pipe-line rend un code de retour de 0, le second est exécuté, sinon il n'est pas exécuté. Expérimentons:

```
$ erreur 34 && echo hello
$ erreur 0 && echo hello
hello
$
```

Cette structue est adaptée à l'exécution d'un programme conditionnée par la bonne exécution d'un autre programme.

Le code de retour d'une structure **&&** est le code de retour du dernier pipe-line exécuté.

```
$ erreur 34 && erreur 18
$ echo $?
34
$ erreur 0 && erreur 18
$ echo $?
18
$
```

5.4 La structure ||||

Deux pipe-lines peuvent être connectés par la structure `||`. Sa sémantique est la suivante: si le premier pipe-line rend un code de retour différent de 0, le second est exécuté, sinon il n'est pas exécuté. Cette structure est bien adaptée à l'émission conditionnelle de messages d'erreurs. Par exemple:

```
grep $USER /etc/passwd || echo "$USER" "n'existe pas"
```

remplace avantageusement:

```
if grep $USER /etc/passwd
then
   :
else
   echo "$USER" "n'existe pas"
fi
```

Le code de retour d'une structure `||` est le code de retour du dernier pipe-line exécuté.

5.5 Notion de liste de pipe-lines

La syntaxe des structures de contrôle qui suivent fait appel à la notion de *liste de pipe-line*. Sa définition est la suivante: une liste de pipe-lines est un ensemble de pipe-lines connectés par `;` **&&** `||` ou *line-feed*.

Le code de retour d'une liste de pipe-lines est le code de retour du dernier pipe-line exécuté.

Exemple de liste de pipe-lines:

```
cmd && echo "execution de cmd ok" || echo "erreur a l'execution de cmd"
```

ou:

```
sort rawdata > data
lpr data
```

5.6 La structure for

5.6.1 La syntaxe

Attention a la syntaxe qui est pénible: il faut aller a la ligne de la manière indiquée ci-dessous.

La syntaxe est la suivante:
`for` *nom* `in` (*liste-de-mots*)
`do` *liste-de-pipe-lines*
`done`

Dans *liste-de-pipe-lines* les commandes peuvent être séparées par des points virgules ou par des Retour Chariot.

La partie `in` (*liste-de-mots*) est optionnelle. Si elle est omise, le shell la remplace par `"$@"`. La sémantique de `"$@"` est expliquée au chapitre 4.9.

Ecriture d'un for sur une seule ligne:
`for` *nom* `in` *liste-de-mots* `; do` *liste-de-pipe-lines* `; done`

5.6.2 La sémantique

La structure for itère l'exécution de *liste-de-pipe-lines*. A chaque pas de l'itération, la variable *nom* prend comme valeur le mot suivant de *liste-de-mots*

5.6.3 Exemples

```
for fichier in essai1.c essai2.c essai3.c
do
        echo je compile $fichier
        cc -c $fichier
done

for fichier in *.c; do cc -c $fichier; done
```

5.7 La structure if

5.7.1 La syntaxe

Il y a plusieurs formes possibles: avec ou sans partie else, partie else combinée avec un if pour faire un elif (else if).

La syntaxe est la suivante:
`if` *liste-de-pipe-lines*
`then` *liste-de-pipe-lines*
`fi`

ou:
`if` *liste-de-pipe-lines*
`then` *liste-de-pipe-lines*
`else` *liste-de-pipe-lines*
`fi`
ou:
`if` *liste-de-pipe-lines*
`then` *liste-de-pipe-lines*
`elif` *liste-de-pipe-lines*
`else` *liste-de-pipe-lines*
`fi`
écriture d'un if sur une seule ligne:
`if` *expression* `; then` *liste-de-pipe-lines* `; fi`
`if` *expression* `; then` *liste-de-pipe-lines* `else` ; *liste-de-pipe-lines* `; fi`

5.7.2 La sémantique

La *liste-de-pipe-lines* après le if est exécutée, si elle rend un code de retour nul, la *liste-de-pipe-lines* de la partie then est exécutée, sinon, la *liste-de-pipe-lines* de la partie else est exécutée.

Dans la pratique, la *liste-de-pipe-lines* après le if est généralement réduite a une seule commande, et c'est souvent `test`.

On remarquera que le `if` su shell fonctionne à l'inverse de celui du langage C. Pour le shell, la partie *then* est exécutée si la partie *condition* a rendu une valeur nulle.

5.7.3 Exemples

```
if [ $langue = "francais" ]
then echo bonjour
else echo hello!
fi
```

5.8 La structure case

5.8.1 La syntaxe

La syntaxe est la suivante:
`case` *mot* `in`
liste-de-modèles `)` *liste-de-pipe-lines* `;;`
liste-de-modèles `)` *liste-de-pipe-lines* `;;`
`esac`

Dans les *listes-de-modèles* les *modèles* sont séparés par le caractère `|`

5.8.2 La sémantique

La sémantique des modèles

Les *modèles* sont les mêmes que ceux qui servent de modèles de noms de fichiers. Les métacaractères sont `* ? [ ]` avec la même sémantique. Un mot est dit conforme à une *liste-de-modèles* si il est conforme à l'un quelconque des modèles de la liste (le caractère `|` a donc le sens d'un "ou").

La sémantique du case

Le *mot* est comparé successivement à chaque *liste-de-modèles.* A la première *liste-de-modèles* pour laquelle le *mot* correspond, on exécute la *liste-de-pipe-lines* correspondante.

5.8.3 Exemples

```
case $langue in
   francais) echo Bonjour ;;
   anglais)  echo Hello ;;
   espagnol) echo Buenos Dias ;;
esac

case $param in
   0|1|2|3|4|5|6|7|8|9 ) echo $param est un chiffre ;;
   [0-9]*              ) echo $param est un nombre ;;
   [a-zA-Z]*           ) echo $param est un nom  ;;
   *                   ) echo $param de type non prevu ;;
esac
```

5.9 La structure while

5.9.1 La syntaxe

La syntaxe est la suivante:
`while` *liste-de-pipe-lines*
`do` *liste-de-pipe-lines*
`done`

5.9.2 La sémantique

La structure `while` itère:

- exécution de la *liste-de-pipe-lines* du `while`
- si celle ci rend un code de retour nul il y a exécution de la *liste-de-pipe-lines* du `do`, sinon la boucle se termine.

De la même manière que pour la structure `if`, la *liste-de-pipe-lines* de la partie `while` est généralement réduite à une seule commande.

5.9.3 Exemples

L'exemple ci-dessous réalise une boucle de 10 itérations:

```
i=1
while test $i -le 10
do
   ...
   i=`expr $i + 1`
done
```

5.10 La structure until

5.10.1 La syntaxe

La syntaxe est la suivante:
`until` *liste-de-pipe-lines*
`do` *liste-de-pipe-lines*
`done`

5.10.2 La sémantique

La structure `until` itère l'exécution de la *liste-de-pipe-lines* du `until`, si celle ci rend un code de retour nul il y a exécution de la *liste-de-pipe-lines* du `do`, sinon la boucle se termine.

De la même manière que pour la structure `if`, la *liste-de-pipe-lines* de la partie `until` est généralement réduite à une seule commande.

5.11 La structure sous-shell

5.11.1 La syntaxe

La syntaxe est la suivante:
`(` *liste-de-pipe-lines* `)`

5.11.2 La sémantique

La *liste-de-pipe-lines* est exécutée par une autre invocation de `sh`.

5.11.3 Exemples

L'utilité d'une telle structure est de permettre d'agir à l'aide des opérateurs pipe, mise dans l'arrière plan, redirection des entrées-sorties sur une ensemble de commandes, et non pas sur une seule commande.

Exemple 1: on veut exécuter en arrière plan deux commandes dont l'ordre d'exécution doit être respecté : la première produit un fichier nécessaire à la seconde.

```
(sort data -o data_trie; lpr data_trie) &
```

Exemple 2: on veut piper la sortie de deux commandes dans une autre commande:

```
(echo "je t'envoie ca pour information" ; cat lettre ) | mail serge
```

Exemple 3: on veut rediriger l'erreur standard de plusieurs commandes à la fois.

```
(cc -c fic1.c ; cc -c fic2.c ) 2>erreurs
```

5.12 Plus sur les structures de contrôle

5.12.1 Break et continue

Il est possible de sortir d'une boucle `for`, `while` ou `until` à l'aide de la commande interne `break`. Il est possible d'indiquer un paramètre numerique permettant de sortir de plusieurs boucles à la fois. Exemple:

```
while true
do
   read ligne
   if [ $ligne = stop ]
   then
      break
   else
   ...
   fi
done
```

5.12.2 Redirection des entrées-sorties

Il est possible de rediriger les entrées-sorties d'une structure conditionnelle (`if` ou `case`) ou de boucle (`for`, `while`, `until`) ou de sous-shell. ex:

```
for i in fic1 fic2
do
   cc -c $i.c
done 2> erreurs
```

Chaque commande peut avoir ses entrées-sorties redirigées indépendement:

```
for i in fic1 fic2
do
   echo "Je traite $i" > /dev/tty
   cc -c $i.c
done 2> erreurs
```

5.12.3 Pipe

Il est possible de piper les mêmes structures soit en entrée soit en sortie.
ex:

```
cat noms | while read nom
do
   sort $nom -o $nom
done
```

5.12.4 Exécution en arrière plan

Il est possible d'exécuter les structures de contrôle en arrière plan.

```
for i in fic1 fic2
do
   cc -c $i.c
done &
```

5.12.5 Attention

Quand on redirige ou que l'on pipe les entrées-sorties ou que l'on met dans l'arrière plan un telle structure, celle-ci est exécutée dans un sous-shell. Attention donc si on utilise des variables. ex:

```
nbligne=0
grep hello data  | while read ligne
do
nbligne=`expr $nbligne + 1`
done
```

A la fin, la variable nbligne vaut toujours 0! Il y a eu incrémentation de la variable nbligne du sous-shell crée pour exécuter le while puis disparition de cette variable lors de la fin du while qui a entrainé la mort du sous-shell.

Chapter 6

Les commandes internes

Une ligne soumise au shell est interprétée comme étant la demande d'exécution d'une commande. Ces commandes sont généralement des programmes indépendants du shell comme par exemple `ls` ou `echo`. Le shell reconnait cependant un certain nombre de commandes comme étant des ordres qui lui sont adressés: on les appelle les commandes internes.

6.1 Pourquoi des commandes internes?

Nous avons déja vu (chapitre 2.6) que le shell lance un nouveau processus pour exécuter un programme. Il est donc impossible de modifier de cette manière l'etat du processus qu'est le shell. En particulier, le répertoire courant fait partie de l'état d'un processus. Il est impossible que la commande `cd` (Change directory) soit une commande externe au shell. Il faut que le shell interprète lui-même cette commande en exécutant un appel au noyau UNIX `chdir(2)`.

6.2 La commande commentaire

La commande `#` sert à indiquer une ligne commentaire dans les fichiers de commandes. On rappelle que la commande `what` extrait d'un fichier toute lignes contenant la chaine magique `@(#)`. C'est une bonne méthode de travail de commencer un fichier de commande par quelques lignes ainsi commentées pour donner la fonction du fichier.

```
#!/bin/sh
#
# @(#) consultation du fond documentaire de la mediatheque
#
```

6.3 La commande : (deux points)

Cette commande ne fait rien, mais ça ne l'empèche pas d'être utile!

```
if grep $chaine $fichier > /dev/null
then
   :    ne rien faire
else
```

```
   echo "Attention $chaine n'exite pas dans $fichier"
   exit 1
fi
```

Ce qu'il y a après le signe `:` est ignoré, cela peut donc servir de commentaire, mais la commande `:` n'est pas un commentaire, c'est une commande nulle. En particulier, elle permet de satisfaire la syntaxe du `if`, ce que la commande `#` ne permet pas.

6.4 la commande . (point)

Cette commande a eté traitée au chapitre 2.6.3.

6.5 La commande `cd`

La syntaxe est: `cd` *nom-de-repertoire*

Cette commande demande au shell de prendre comme répertoire courant *nom-de-répertoire*. Le *nom-de-repertoire* est optionnel, quand il est omis on utilise la valeur de la variable `HOME`

6.6 La commande `eval`

Le shell procède aux opérations suivantes dans l'ordre indiqué:

1. découpage en commandes connectées par pipe, redirection des entrées-sorties.
2. substitution de variables
3. substitution de commandes
4. interprétation des blancs
5. génération de noms de fichiers.

Cet ordre est généralement conforme à ce que l'on désire. Il arrive cependant que l'on désirerait un ordre différent. Supposons que l'on veuille obtenir substitution de commande puis substitution de variables:

```
$ cat vars
$V1 $V2

$ V1=1
$ V2=2
$ echo `cat vars`
$V1 $V2
$
```

On voudrait que l'echo nous donne non pas le nom des variables mais leur valeur. La commande interne `eval` permet de résoudre ce problème. Sa syntaxe est la suivante: `eval` *commande*

Elle a pour effet de réaliser deux fois la suite des 5 opérations précédemment citées.

```
$ eval echo `cat vars`
1 2
$
```

Autre exemple où on a besoin de réaliser deux fois la substitution de variables: on veut faire écho d'un paramètre dont le numéro est contenu dans une variable.

```
$ set bonjour a tous
$ echo $3
tous
$ N=3
$ echo \$$N
$3
$ eval  echo \$$N
tous
$
```

6.7 La commande exec

Nous avons déja vu (chapitre 2.6) que le shell exécute les programmes en lançant à chaque fois un nouveau processus. Il arrive parfois que l'on désire que le processus shell exécute lui-même le programme. Cela revient à remplacer l'exécution du shell par l'exécution du programme demandé.

6.7.1 Utilisation normale de exec

En utilisation interactive, cela n'a pas de sens d'utiliser exec, car cela revient à tuer le shell que l'on utilise. Dans un fichier de commandes au contraire, on peut utilise exec pour exécuter la dernière commande du fichier de commandes . Cela économise les ressources de la machine: au lieu d'y avoir deux processus, il n'y en a qu'un.

6.7.2 exec et redirection des entrées-sorties

Il est possible de rediriger les entrées-sorties de la commande qui est soumise à `exec`. Mais on peut aussi invoquer `exec` sans commande, simplement en redirigeant les entrées-sorties. Dans ce cas, `exec` a pour effet de rediriger les entrées-sorties du shell. Par exemple la commande interne `read` lit sur l'entrée standard. On peut la faire lire à partir d'un fichier en redirigeant l'entrée standard du shell grâce à `exec`. Voici en exemple un fichier de commandes qui imprime la première colonne d'un fichier dont le nom est passé en paramètre.

```
#!/bin/sh
exec < $1
while read ligne
do
   set $ligne
   echo $1
done
```

Il est possible d'affecter dans un même fichier de commandes les entrées-sorties vers des fichiers différents à l'aide de plusieurs appels à **exec**. Si après redirection, on désire réaffecter l'entrée standard au terminal, il suffit de faire:

```
exec < /dev/tty
```

`/dev/tty` est en effet un pseudo-périphérique qui désigne le terminal de l'utilisateur.

6.8 La commande exit

La commande interne `exit` permet de terminer un shell en transmettant un code de retour. Si `exit` est invoqué avec un paramètre, c'est ce paramètre qui est pris comme code de retour. Sinon, le code de retour sera le code de retour de la dernière commande exécutée.

Clasiquement, les test de validité d'appel d'un fichier de commandes sont codés de la manière suivante:

```
#!/bin/sh
if [ $# -ne 2 ]
then
        echo "Il faut les parametres ... et ..."
        exit 1
fi
```

Ici, le programmeur a choisi la valeur 1 comme signifiant "nombre de paramètres incorrect".

6.9 La commande export

Cette commande a été traitée de manière exhaustive au chapitre 4.11.2.

6.10 La commande login

Syntaxe: `login` *paramètres*

Son effet est celui de `exec login` *paramètres*. Ceci signifie que l'on abandonne le shell courant. Si on désire appeler `login` sans abandonner son shell courant, il faut faire: `/bin/login` *paramètres*. Dans ce cas, lors du logout, on reviendra dans le shell appelant.

6.11 La commande read

Syntaxe: `read` *liste-de-noms*

La commande `read` lit une ligne sur l'entrée standard, la découpe en mots et affecte chaque mot aux variables de la *liste-de-noms*. Si la ligne lue comporte plus de mots que la *liste-de-noms* comporte de noms, il affecte au dernier *nom* l'ensemble des mots restant à affecter. `read` rend un code de retour nul sauf sur fin de fichier. Exemple d'utilisation:

```
while read c1 c2 c3
do
   echo c1 = $c1 c2 = $c2 c3 = $c3
done
```

6.12 La commande `readonly`

La commande `readonly` a ete traitée de manière exhaustive (chapitre 4.8).

6.13 La commande `set`

La commande `set` a trois utilisations possibles:

1. Utilisée sans paramètre, elle donne la liste alphabétique des variables avec leur valeur. Ceci a été vu au chapitre 4.7.1.
2. Elle permet d'affecter se paramètres aux variables `1`, `2`, ... Ceci a été vu au chapitre 4.7.2.
3. Elle permet egalement de positionner des options du shell.

Les options possibles sont les suivantes:

-e SI le shell n'est pas interactif, stopper à la première commande qui rend une erreur.

-k traiter tous les couples nom=valeur, pas seulement en sébut de ligne mais n'importe où dans la ligne.

-n lire les commandes mais ne pas les exécuter. Permet de vérifier la syntaxe d'un fichier de commandes.

-t terminer le shell après avoir exécuté une seule commande.

-u generer une erreur sur référence d'une variable non explicitement initialisée.

-v imprimer les lignes lues par le shell avant de les exécuter.

-x imprimer les commandes et leurs paramètres avant de les exécuter. Utile pour mettre au point un fichier de commandes.

6.14 La commande `shift`

La commande `shift` a été traitée de manière exhaustive (chapitre 4.10).

6.15 La commande `times`

Imprime les temps cumulés pour tous les processus lancés depuis le début de la session. Les temps imprimés sont les temps d'exécution en mode utilisateur et temps d'exécution en mode système.

```
$ times
0m0s 0m0s
$ sort data -o data_triees
$ times
0m12s 0m2s
```

```
$  sort data -o data_triees
$ times
0m25s 0m4s
$
```

6.16 La commande `trap`

La commande interne `trap` permet de spécifier une commande à exécuter sur réception de signaux. Sa syntaxe est la suivante:
`trap` *commande liste-de-signaux*
dans laquelle la *liste-de-signaux* est une liste de numéros correspondant aux signaux gérés par le noyau UNIX. Voici une liste des signaux qu'il peut être interessant de récupérer Dans un fichier de commandes. cette liste est compatible UNIX BSD et System V.

0	exit du shell
1	Hangup
2	Interrupt
3	Quit
15	Software terminaison

6.16.1 Exemple

Voici un fichier de commandes dans lequel on récupère le signal "Interrupt".

```
$ cat recup_intr
#!/bin/sh
trap "echo interruption" 2
while true
do
   sleep 10000
   echo "je suis reveille"
done
$
```

Et une activation:

```
$ recup_intr

interruption
je suis reveille

interruption
je suis reveille

interruption
je suis reveille
```

A 3 reprises, l'utilisateur a envoyé le caractère "intr" (dans ce cas, la touche DEL du terminal):

```
$ stty -a
speed 9600 baud; line = 0;
intr = DEL; quit = ^|; erase = ^h; kill = ^u; eof = ^d; eol = ^‘
swtch = ^‘
-parenb -parodd cs8 -cstopb hupcl cread -clocal
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl -iuclc
ixon -ixany -ixoff -loblk
isig icanon -xcase echo echoe echok -echonl -noflsh
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel
```

6.16.2 Application typique

L'application typique de la récupération des signaux est le nettoyage a réaliser sur les fichiers temporaires. Dans le cas où un fichier de commandes a besoin de fichiers temporaires, il ne faut pas oublier d'effacer ces fichiers en cas d'avortement de l'exécution. Comme ce nettoyage doit aussi se faire sur terminaison normale d'exécution, on programme classiquement de la manière suivante:

```
#!/bin/sh
#   TEMPO : fichier temporaire
TEMPO=toto.$$
trap "rm -f $TEMPO" 0 1 2 3 15
....
```

Petite subtilité: dans le cas de la terminaison sur réception de l'un des signaux 1 2 3 ou 15, la commande `rm` sera exécutée deux fois, car en plus il y aura le signal 0 (fin du shell). Ceci explique dans l'exemple, l'utilisation de l'option `-f` pour éviter le message d'erreur de `rm` lorsqu'à la seconde exécution il essaye d'effacer un fichier qui n'existe pas.

6.17 La commande `umask`

6.17.1 umask dans le noyau UNIX

Dans le contexte d'un processus UNIX il existe une variable qui porte le nom de *umask*. Cette variable est utilisée lorsque le processus crée un fichier. Lors d'une création de fichier, l'utilisateur spécifie les permissions qu'il désire affecter a ce fichier. Mais les permissions effectivement allouées au fichier sont les permissions demandées auquelles on enlève celles de *umask*. Chaque bit à 1 dans *umask* indique en effet une position à mettre à 0 dans les permissions du fichier. Classiquement, la valeur par défaut de *umask* est 022 (en octal) de manière à ne pas mettre les permissions write au groupe et aux autres.

6.17.2 umask dans le shell

La commande interne `umask` permet d'affecter une valeur au `umask` du processus qu'est le shell. Sa synatxe est la suivante: `umask` *nombre*
où *nombre* est un nombre exprimé en octal. Il faut savoir que lorsque le shell crée un fichier pour redirection d'une sortie, il le crée en utilisant les permissions `-rw-rw-rw-` Exemple:

```
$ umask 0
$ > t1
$ ls -l t1
-rw-rw-rw-  1 bernard         0 Nov 13 15:03 t1
$ umask 22
$ > t2
$ ls -l t2
-rw-r--r--  1 bernard         0 Nov 13 15:04 t2
$
```

6.18 La commande `wait`

La commande interne **`wait`** a pour but s'attendre la fin d'un processus qui s'exécute en arrière plan. Sa syntaxe est la suivante:
`wait` *numéro*
où *numéro* est le numéro de processus que l'on désire attendre.

Chapter 7

La commande test

7.1 Présentation

La commande test permet d'exprimer des prédicats sur les chaînes de caractères, sur les entiers et sur les fichiers. La valeur de ces prédicats peut ensuite être testée dans une structure `if`, `while` ou `until`. La commande `test` ne fait pas partie du shell, mais la programmation en shell en fait un tel usage qu'il est logique de lui consacrer un chapitre. La commande `test` réside traditionnelement sous `/bin`. Le shell en fait un tel usage, qu'il y a un alias qui porte le nom `[` (oui, c'est un fichier qui se nomme `[`)! Ceci peut se voir grâce à l'option `-i` de `ls`:

```
$  ls -i /bin/test /bin/[
  4208 /bin/[             4208 /bin/test
$
```

Les deux noms ont même numéro de inode, ils repèrent donc le même fichier.

Ceci permet d'écrire:

```
if [ $rep = "oui" ]
then
   ...
fi
```

à la place de :

```
if test $rep = "oui"
then
   ...
fi
```

Le shell System V a introduit la commande `test` en commande interne, mais en restant compatible avec `/bin/test`.

7.2 Prédicats sur les chaînes de caractères

Dans ce qui suit, on notera s1 et s2 deux chaînes de caractères. Les opérateurs dont on dispose sont les suivants:

s1 = s2	vrai si s1 est égal à s2
s1 != s2	vrai si s1 est différent de s2
-n s1	vrai si s1 est non vide
-z s1	vrai si s1 est vide

7.3 Prédicats sur les entiers

Dans ce qui suit, n1 et n2 sont des nombres décimaux.

n1 **-eq** n2	vrai si n1 est égal à n2
n1 **-ne** n2	vrai si n1 est différent de n2
n1 **-gt** n2	vrai si n1 est strictement supérieur à n2
n1 **-ge** n2	vrai si n1 est supérieur ou égal à n2
n1 **-lt** n2	vrai si n1 est strictement inférieur à n2
n1 **-le** n2	vrai si n1 est inférieur ou égal à n2

7.4 Prédicats sur les fichiers

-r fichier	vrai si fichier existe et on a le droit read
-w fichier	vrai si fichier existe et on a le droit write
-f fichier	vrai si fichier existe et n'est pas un répertoire
-d fichier	vrai si fichier existe et est un répertoire
-s fichier	vrai si fichier existe et a une taille non nulle

7.5 Prédicats sur les descripteurs de fichiers

-t fd	vrai si fd est un descripteur associé à un terminal

7.6 Combinaison de prédicats

!	opérateur not booléen
-a	opérateur and booléen
-o	opérateur or booléen
( )	pour parenthéser les sous-expressions

7.7 Les problèmes

Attention à bien passer chaque opérateur et opérande comme un argument à test. Ne pas écrire

```
if [$cmd=end]
```

mais:

```
if [ $cmd = end ]
```

Attention en ce qui concerne les parenthèses: elles ont une signification pour le shell, il faut donc les quoter pour les tranmettre telles quelles à test. ex:

```
if [ $ordre = w -a \( $nb -qt 20 -o $nb -lt 10 \) ]
then
...
fi
```

Chapter 8

La commande expr

La commande `expr` permet de réaliser des calculs arithmétiques, des comparaisons de nombres entiers et de la reconnaissance de modèle. Dans sa capacité de comparaison d'entiers, elle fait double emploi avec la commande `test`. Cette commande interprète ses arguments comme étant une expression, et imprime le résultat sur sa sortie standard. Chaque opérande ou opérateur de la liste d'argument doit être passé comme un seul paramètre.

Exemple d'utilisation:

```
$ expr 2 + 3
5
$ expr 2 +3
syntax error
$
```

Dans ce qui suit, on notera *expr* une expression formée d'une combinaison d'entiers, d'opérateurs autorisés par `expr` et de parenthèses.

8.1 Calculs entiers

On dispose des opérateurs suivants:

expr + *expr*	addition
expr - *expr*	soustraction
expr * *expr*	multiplication
expr / *expr*	division
expr % *expr*	reste de la division

Attention

L'opérateur * est un métacaractère pour le shell, il faut le passer à `expr` de manière littérale.

```
$ expr 3 * 4
syntax error

$ expr 3 \* 4
12
$
```

8.2 Comparaisons sur les entiers

On dispose des opérateurs suivants:

expr = *expr*	égal
expr != *expr*	différent
expr > *expr*	strictement supérieur
expr >= *expr*	supérieur ou égal
expr < *expr*	strictement inférieur
expr <= *expr*	inférieur ou égal

Attention

Tous ces opérateurs sauf = et != sont des métacaractères du shell.

8.3 Les parenthèses

On peut parenthéser les expressions, mais comme les parenthéses sont des métacaractères pour le shell, il faut les passer à `expr` de manière littérale.

Exemple:

```
$ expr 3 \* \( 4 + 5 \)
27
$
```

8.4 Calculs booléens

On dispose des deux opérateurs booléens:

expr1 | *expr2* rend *expr1* si elle n'est pas nulle rend *expr2* sinon

expr1 & *expr2* rend *expr1* si aucune *expr* n'est nulle rend 0 sinon

Dans ce qui précède, l'expression "être nul" peut signifier, soit être nul au sens arithmétique, soit être une chaîne nulle (vide).

Attention

Ces deux opérateurs sont des métacaractères du shell.

Exemple:

```
$ expr 1 \& 0
0
$ expr 2 \& 1
2
$ expr 3 \& ''
0
$
```

8.5 Reconnaissance d'expressions régulières

La syntaxe est la suivante: *expr1* : *expr2*.
expr2 doit être une expression régulière du type de celles reconnues par `ed`.

- Si *expr2* ne comporte pas de sous expression régulière (parenthésée par `\ (` et `\ )`), l'opérateur `:` rend comme résultat le nombre de caractères de *expr1* si celle-ci correspond à l'expression régulière, et 0 sinon.
- Si *expr2* comporte une sous expression régulière, l'opérateur `:` rend la partie de *expr1* qui correspond à la sous-expression regulière, ou le vide si *expr1* ne correspond pas à *expr2*.

Exemples:

```
$ expr aaaa : 'a*'
4
$ expr aaa : 'b*'
0
$ expr aaaabbbaa : 'a*\(b*\)a*'
bbb
$ expr acccaa :  'a*\(b*\)a*'

$
```

8.6 Utilisation avancée

En combinant plusiurs possibilités de **expr**, on peut écrire des constructions très puissantes. L'exemple qui est donné dans le manuel de **expr** est le suivant. Soit une variable `FICHIER` dont la valeur est un nom de fichier. Ce nom peut être simple (**main.c**) ou être composé (**../sh/main.c**). Supposons que nous voulions extraire le dernier composant de ce nom.

```
$ FICHIER=main.c
$ expr $FICHIER : '.*/\(.*\)' \| $FICHIER
main.c
$ FICHIER=../sh/main.c
$  expr $FICHIER : '.*/\(.*\)' \| $FICHIER
main.c
$
```

Glossaire

`#!/bin/sh`	A mettre en tête d'un fichier de commandes pour avoir la certitude qu'il sera exécuté par `/bin/sh` quelque soit le shell interactif utilisé.
`$#`	Nombre de paramètres d'un fichier de commandes.
`$$`	Numéro de processus du shell.
`$-`	Options courantes su shell.
`$0`	Dans un fichier de commandes, nom du fichier.
`$1$2...$9`	Paramètres d'un fichier de commandes repérés par leur position.
`$?`	Valeur retournée par la dernière commande exécutée.
`&`	Métacaractère provoquant l'exécution en arrière plan.
`*`	Métacaractère de la génération de noms de fichiers.
`/dev/null`	Source vide pour les entrées et puits pour les sorties.
`;`	Métacaractère provoquant l'exécution en séquence.
`?`	Métacaractère de la génération de noms de fichiers.
`[`	Métacaractère de la génération de noms de fichiers.
`anti-slash`	(il s'agit du caractère \ ). Métacaractère permettant de rendre littéral le caractère qui suit (dans certains contextes seulement).
`‘`	Métacaractère provoquant la substitution de commandes.
`|`	Métacaractère provoquant la création d'un pipe-line.
HOME	Variable du shell contenant le nom du répertoire d'origine de l'utilisateur.
PATH	Variable du shell contenant la liste des répertoires dans lesquels rechercher les commandes à exécuter.
PS1	Variable du shell contenant le prompt principal.
PS2	Variable du shell contenant le prompt secondaire.

argument	Terme anglais pour paramètre.
arrière plan	technique d'exécution de programme qui consiste pour le shell, a ne pas attendre la terminaison du programme et à imprimer le prompt immédiatement. Voir aussi avant plan.
avant plan	technique d'exécution de programme qui consiste pour le shell, a se bloquer jusqu'à la fin du programme. Ensuite, le shell imprime le prompt. Voir aussi arrière plan.
background	Terme anglais pour arrière plan.
built-in	Terme anglais pour commande interne.
BSD	Berkeley System Distribution. Nom de la version d'UNIX qui est distribuée par l'Université de Californie à Berkeley.
code de retour	Le code de retour d'un programme est une valeur servant à transmettre à l'appelant une indication sur la manière dont le programme s'est exécuté.
commande	Une commande est un programme système, par opposition à un programme developpé par un utilisateur.
commande interne	Un ordre directement interprété par le shell, par opposition aux commandes qui sont des programmes externes au shell.
`csh`	Le C shell.
directory	Terme anglais pour répertoire
entrée standard	file descriptor 0 d'un programme. Par défaut, l'entrée standard est affectée au terminal (le clavier).
erreur standard	File desriptor 2 d'un programme. Par défaut, 'erreur standard est affectée au terminal (l'écran).
fichier de commandes	Programme écrit dans le langage reconnu par l'interpréteur de commandes.
file descriptor	Périphérique logique d'un programme. Le shell lance l'exécution d'un programme après avoir affecté ses file descriptor à des périphérique physiques ou à des fichiers. Le terme file descriptor est du jargon spécifique à UNIX.
flot de données	Suite de données lues ou écrites par un programme.
flot d'entrée	Suite de données lues par un programme.
flot de sortie	Suite de données écrites par un programme.
foreground	Terme anglais pour avant plan.

génération de noms de fichiers	capacité du shell à remplacer un modèle par l'ensemble des noms des fichiers qui corrspondent à ce modèle
home directory	Terme anglais pour répertoire d'origine.
interprétation des blancs	La phase d'interprétation des blancs est la phase où le shell découpe une commande en mots. Le premier mot sera le nom du programme appelé, les autres mots seront les paramètres. Les caractères qui sont considérés comme des "blancs" sont contenus dans la variable IFS.
interpréteur de commandes	Programme dont la vocation principale est le lancement et l'exécution de programmes.
`ksh`	Le Korn shell. Développé par David Korn de ATT Bell Laboratories.
line-feed	Nom du caractère ASCII de code héxadécimal 0a. C'est un caractère non graphique dont l'effet est de passer à la ligne suivante. Il est utilisé par UNIX en temps que séparateur de ligne dans les fichiers texte.
nom de chemin	Un *nom de chemin* est la désignation d'un fichier dans l'arborescence UNIX.
	Terme anglais pour nom de chemin.
pipe	Un concept du noyau UNIX. Un pipe est une file d'attente de caractères. Sert à la commnunication entre processus.
pipe-line	Ensemble de programmes connectés les uns aux autres, et collaborant selon la technique du travail à la chaîne: la sortie de l'un va dans l'entrée du suivant via un pipe. Le terme pipe-line est du jargon spécifique aux shells UNIX.
processus	Programme en exécution
programme binaire	Programme sous forme de binaire exécutable, par opposition à un programme sous forme de source ou à un fichier de commandes.
prompt	petit message d'invitation à taper. Un programme imprime un promt pour prévenir l'utilisateur qu'il s'est mis en attente de lecture.
quote	A strictement parler, il s'agit d'un caractère typographique qui peut être `'` (simple quote), `` ` `` (back quote) ou `"` (double quote). En LISP le "simple quote" est utilisé pour prendre de manière littérale ce qui suit. Cet usage a été repris dans d'autres langages, ce qui a amené une extension du sens du mot quote. Dans un sens large, "quote" désigne tout caractère qui rend littéral ce qui suit. Dans le shell, les quotes sont `'` `"` `\` .

répertoire	Un nœud de l'arborescence de fichiers UNIX. Un répertoire a pour but de contenir des fichiers et des répertoires.
répertoire courant	L'état d'un processus UNIX contient une référence à un répertoire dit *répertoire courant*. Les noms de chemin relatifs sont interprétés relativement au répertoire courant.
répertoire d'origine	Le répertoire qui est pris comme répertoire courant au moment du login.
`set`	commande interne
`sh`	Le Bourne shell.
script	Abbréviation pour shell script.
shell	Terme anglais pour interpréteur de commandes.
shell script	Terme anglais pour fichier de commandes.
sortie standard	file descriptor 1 d'un programme. Par défaut, la sortie standard est affectée au terminal (l'écran).
substitution de variable	Capacité du shell à remplacer le nom d'une variable par sa valeur.
substitution de commande	Capacité du shell à remplacer une commande par la sortie qu'elle produit.
System V	Nom de la version d'UNIX qui est distribuée par ATT.
`tcsh`	Le ???
variable	Un couple (nom,valeur) géré par le shell. Il existe deux types de variables les variables exportées et celles qui ne le sont pas. Pour une introduction voir chapitre 1.8, pour une étude exhaustive voir chapitre 4.

Index

Contents