Module n°4

ADMINISTRATION SYSTEME

Auteur : Labo-Linux
Version 0.1 – 17 novembre 2003
Nombre de pages : 126

Paris Academy of Computer Science SUPINFO Paris Ecole Supérieure d'Informatique

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Introduction au shell Bash

1.1. Le shell

Un interpréteur de commandes est un programme qui sert d'intermédiaire entre l'utilisateur et le système d'exploitation.

BASH signifie Bourne Again Shell. C'est un interpréteur de commande puissant et le plus répendu sous GNU\Linux.

Sa tâche essentielle est l'exécution de programmes. Pour cela, il effectue (indéfiniment) :

– La lecture d'une ligne
– Interprétation de la ligne (en général une commande/programme suivi d'arguments)
– Le lancement de ce programme avec passage des paramètres et d'éventuelles redirections d'entrées-sorties

- Les exécutions de scripts (fichiers de commandes)

Le mode ligne de commande peut paraître complexe au premier abord, mais BASH possède de nombreuses fonctionnalités pour faciliter son utilisation.

1.2. Démarrage du shell

Lors de la création de son compte, un utilisateur est associé à un type de shell.

Cette information est contenu dans le fichier /etc/passwd : le dernier champ contient le nom du shell par défaut : /bin/bash.

Le shell associé est ainsi lancé automatiquement dès la saisie du login utilisateur. Le shell poursuit sa configuration en exécutant des scripts globaux à tous les utilisateurs et des scripts liés au compte et qui permettent une personnalisation.

Enfin, il affiche le prompt et se met en attente de la lecture d'une commande jusqu'à la commande exit, pour quitter le shell (ce qui équivaut à se déconnecter : logout).

1.3. Les scripts de connexion

Tout d'abord le shell exécute le script **/etc/profile** communs à tous les utilisateurs, y compris à root. On y trouve notamment la définition de **umask**.

Une fois ce script exécuté, le shell recherche le fichier **$HOME/.bash_profile** (la variable d'environnement **$HOME** contient le chemin vers le répertoire personnel).

Il s'agit ainsi d'un fichier de démarrage personnel et paramétrable.

A son tour, il exécute **$HOME/.bashrc** dans lequel il est recommandé de placer toutes les fonctions ou alias personnels (car .bashrc est exécuté dans tout shell bash).

Enfin le précédent script exécute /etc/bashrc, dans lequel on place les alias globaux et la définition symbolique du prompt **$PS1**.

La variable d'environnement **$PS1** représente le prompt du shell. Un prompt est un message d'attente pour permettre à l'utilisateur de savoir que le shell est prêt à recevoir des commandes. Il indique également le nom de l'utilisateur connecté au shell, ainsi que diverses informations. Ce prompt se termine généralement par un # lorsque vous êtes root (administrateur), et par un $ pour les autres utilisateurs. Bien entendu ce prompt est personnalisable en modifiant la déclaration de **PS1** dans le fichier /etc/profile.

1.4. Personnalisation des commandes bash

/etc/bashrc étant le dernier script d'initialisation du shell bash, root peut y définir des alias globaux pour tous les utilisateurs :

```
$ vi /etc/bashrc
alias lll='ll | less'
alias x='startx'
alias m='mc'
```

Pour que ces nouvelles commandes soient prises en compte par le nouveau shell il faudra vous re-loggué.

1.4.1. Personnalisation du login utilisateur

Chaque utilisateur peut ajouter des commandes shell au fichier de profil personnel, ~/.bash_profile. Par exemple :

```
clear
salut="Bonjour $USER !\nJe te souhaite bon courage .. \n\ Nous sommes le
$(date) "
# -e option indispensable pour interpréter les \n
echo -e $salut
```

1.5. Les variables d'environnement système

La liste en est accessible par la commande **env** à partir d'un prompt shell. La commande **echo** permet d'obtenir la valeur d'une telle variable. Par exemple :

```
echo $PATH
echo $USER
```

1.5.1. Ajout d'un nouveau chemin

Lorsque vous réalisez des opérations sur les variables systèmes qui existent déjà, prennez garde à ne pas effacer leur contenu ! Ainsi, pour ajouter un chemin à la liste des chemins existants (PATH), réalisez l'opération suivante :

```
-=[jms@localhost]=(~)> export PATH="$PATH:/home/jms/bin"
```

Ceci ajoutera le chemin vers les exécutables du répertoire personnel au chemin des exécutables systèmes.

1.5.2. Autres variables

La variable **$HOME** contient le chemin du répertoire personnel. La commande "cd $HOME" est abrégée en cd.

La variable **$USER** contient le nom de l'utilisateur.

La variable **$SHELL** donne le niveau du shell courant.

1.6. Facilités de saisie des commandes

Comme les commandes Unix sont souvent longues à saisir, diverses facilités sont offertes afin de nous simplifier la vie.

1.6.1. Historique des commandes

Cette liste numérotée est accessible en tapant :

```
$ history |less
```

Pour relancer la commande numéro *n*, saisir (sans espace) **!n** On peut aussi parcourir les précédentes lignes de commandes avec les flèches et les éditer. Ceci permet très facilement de reprendre une précédente commande pour l'éditer et la modifier sans avoir a tout retaper.

1.6.2. Le clic-droit

Dans un terminal console, sélectionner un texte quelconque. Un clic-droit recopie ce texte sur la ligne de commande, même dans une autre console. (Il faut pour cela avoir configurer la souris en mode console (gpm) ou utiliser un termnial en mode graphique X).

1.6.3. L'opérateur tilde

Le caractère tilde ~ (alt 126) seul renvoie au répertoire personnel de l'utilisateur qui l'a saisi. Si l'utilisateur actif est toto, chaque occurrence du caractère ~ est remplacé par le chemin */home/toto*. Le tilde ~ suivi d'un nom d'utilisateur, par exemple jms, renvoie au répertoire personnel de l'utilisateur jms, c'est à dire /home/jms.

Ainsi par cette commande :

```
cd ~stagiaire
```

Tente en vain d'aller dans le répertoire */home/stagiaire*.

1.7. Création et invocation d'un script

1.7.1. Création d'un script

Un script, ou fichier de commandes, est un ensemble de commandes placées dans un fichier texte. Chaque ligne de ce fichier correspond à une ligne qu'aurait pu entrer un opérateur. Un script est destiné à un interpréteur (shell) qui lit, analyse et exécute les commandes qu'il contient.

Bash n'est pas un simple interpreteur de commandes, c'est aussi un language de programmation qui permet de réaliser des scripts qui faciliteront l'administration.

Dans le cas d'un shell-script, l'interpréteur est un des shells d'Unix (sh, csh, ksh). Dans ce document, nous allons apprendre à écrire un script en Bourne-shell (sh ou bash).

La première ligne d'un script doit spécifier le programme à appeler pour l'interpréter.
Elle doit commencer par la séquence "#!" suivie d'au moins une espace puis du nom de l'interpréteur.

Pour un script Perl, la première ligne sera :

```
#! /usr/bin/perl
```

Pour un shell-script écrit en Bourne-shell, elle sera :

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

Le nom d'un shell-script peut être quelconque mais on choisit généralement :

- soit "mon-script" (sans extension), pour que son nom ait l'air d'une commande Unix comme une autre.

- soit "mon-script.sh" (ou "mon script.csh" s'il était écrit en C- shell), pour indiquer clairement qu'il s'agit d'un shell-script.

1.7.2. Invocation d'un script

On peut exécuter un script comme toute autre commande en l'appelant par son nom.

Pour cela, il faut d'une part que les droits d'exécution soient positionnés correctement. Faites par exemple :

```
chmod +x mon_script
```

D'autre part, il faut que le chemin du fichier appartienne à la variable d'environnement PATH (sinon, vous devriez systématiquement préciser le chemin de votre script pour l'exécuter.

Une autre façon d'exécuter un script, et qui aboutit au même résultat, est d'invoquer son interpréteur avec le nom du script en paramètre (exemple : "sh mon script.sh").

Il faut savoir que, lancé ainsi, un script s'exécute dans un shell secondaire (sous-shell), totalement indépendant du shell qui l'a appelé et, par conséquent, que toute modification de l'environnement du script (changement de chemin avec cd, création ou modification de variables, etc...) sera perdue dès que le script sera terminé.

Pour éviter cela, et donc permettre à un shell-script de modifier l'environnement de votre shell, il suffit d'appeler ce script grâce à la commande **source (ou '.').**

Toutes les commandes du script sont alors exécutées dans le shell actuel.

Exemple :

```
#! /bin/sh

# Exemple simple d'un shell-script
ps -f
echo "Attention, je change de répertoire"
cd /tmp
echo -n "Je suis maintenant dans "
pwd
```

Testons à présent :

```
$ pwd
/home/toto/src/sh/ex_simpl
```

On appelle le script exemple :

```
$ exemple
PID   TTY   STAT  TIME  COMMAND
206   1     S     0:00  /bin/login -- toto
1815  1     S     0:00  \_ -sh
1850  1     S     0:00  \_ sh ./exemple (un sous-shell a bien été lancé)
1851  1     R     0:00  \_ ps -f

Attention, je change de répertoire
Je suis maintenant dans /tmp (le script a changé de répertoire)
```

De retour à la ligne de commande :

```
$ pwd
/home/toto/src/sh/ex_simpl (le répertoire n'a pas changé)
```

1.7.3. Valeur retournée par un shell-script

C'est la valeur retournée par la dernière commande exécutée par le script. On peut fixer cette valeur en provoquant la fin du script par la commande :

```
exit n
```

Dans ce cas, c'est n qui est retourné.

Comme pour toute commande, la valeur de retour d'un script peut être récupérée dans la variable **$?**.

Par convention, une valeur de retour égale à 0 signifie que le script s'est bien déroulé. Inversement, toute autre évaleur indique une erreur.

1.7.4. Mise au point

Les shell-scripts s'exécutent sans afficher les commandes exécutées, ce qui est loin être pratique pour leur mise au point.

On peut avoir le détail de l'exécution en invoquant le Bourne-shell avec les options -v ou -vx. L'affichage avec l'option -vx est plus complet qu'avec -v (en particulier les variables sont remplacées par leur valeur) mais le résultat à l'écran peut paraître confus.

Pour la mise au point, on pourra donc lancer un script par la commande "sh -vx mon script" ou en l'appelant comme avant, par son nom, mais après avoir pris soin de changer sa première ligne en " **# ! /bin/sh -vx**".

1.8. Commandes d'un shell-script

1.8.1. Commandes simples

Dans un script, on peut employer toute commande exécutable par l'utilisateur :

– un mot-clé appartenant au Bourne-shell (exemple : if, case, ...)
– une commande interne (builtin) au Bourne-shell (exemple : echo, umask,...)
– une fonction définie par l'utilisateur avec le mot-clé function
– une commande externe (commande appartenant aux chemins de la variable PATH).

Pour savoir d'où provient une commande, on peut utiliser **type**.

Exemple : utilisation de type

```
$ type while
while is a shell keyword
```

```
$ type cd
cd is a shell builtin

$ type lsc
lsc is aliased to 'ls –color=auto'

$ type bonjour
# bonjour est une fonction défini dans l'un de vos script de démarrage
bonjour ()
{
      echo "Bonjour Tux"
}

$ type mkdir
mkdir is /bin/mkdir
```

Tout ce qui suit un caractère dièse '#' est considéré comme un commentaire. Toute commande suivie du caractère '&' s'exécutera en tâche de fond (arrière plan).

Une commande peut s'écrire sur plusieurs lignes mais dans ce cas, toutes les lignes sauf la dernière doivent être terminées par un anti- slash ('\') pour indiquer qu'il faut lier la ligne qui suit.

1.8.2. Commandes composées

Les exemples qui suivent sont donnés avec deux commandes mais peuvent être étendus à N commandes.

1.8.3. Commandes séquentielles

```
commande1 ; commande2
```

Exécute commande1 puis commande2.
Le point-virgule permet de réunir sur une seule ligne deux instructions qui devraient être sur deux lignes séparées. Lorsque c'est possible, son utilisation est à éviter puisqu'elle nuit à la lisibilité.

Si commande1 doit être exécutée en tâche de fond (c'est à dire est suivie d'un '&'), le point-virgule ne doit pas être précisé (voir paragraphe suivant).

1.8.4. Le pipe

```
commande1 | commande2
```

Exécute *commande1* et *commande2*.
Grâce au tube, *commande2* effectue son traitement sur le texte qu'aurait affiché *commande1*.
Attention, *commande1* et *commande2* s'exécutent dans deux shells secondaires (sous-shells) et, par conséquent, toute modification qu'ils apportent à l'environnement (variables, chemin actuel,...) est perdue dès la fin de la commande.

1.8.5. Commandes en parralèles

```
commande1 & commande2
```

Exécute *commande1* et *commande2* en parallèle. *commande1* est exécutée en tâche de fond.

1.8.6. Commande sur erreur

```
commande1 || commande2
```

Exécute *commande2* si et seulement si *commande1* a échoué.
Est généralement utilisé dans les conditions des structures if, while et until.

1.8.7. Commandes sur réussite

```
commande1 && commande2
```

Exécute *commande2* si et seulement si *commande1* s'est bien déroulée. Est généralement utilisé dans les conditions des structures if, while et until.

1.8.8. Commandes en paramètre

```
commande1 $(commande2) ou commande1 `commande2`
```

Cette combinaison exécute *commande1* avec en paramètre le texte qu'aurait affiché *commande2*.

Exemple :

```
#! /bin/sh
#
# Shell-script de mise en application de la récupération du résultat
# d une commande. Affiche l'heure en français.

heures=$(date +%H)
minutes=`date +%M`
echo "Il est $heures heures $minutes minutes et $(date +%S) secondes"
```

```
$ ./heure
Il est 12 heures 29 minutes et 42 secondes
```

Les deux variantes proposées sont strictement identiques. On utilise la notation $() parce que l'anti-quote (‘‘) n'est pas toujours (facilement) accessible au clavier ou parce qu'elle n'est pas très différentiable de l'apostrophe à l'écran.

1.8.9. Evaluation

eval *commande* :
eval exécute la commande qui lui est donnée en paramètre. Dans la construction qui est proposée ici, le résultat de "une commande" est une autre commande ("ls -l /src") qui est alors exécutée par *eval.*

Exemple :

```
$ ./une_commande
ls -l ~/src

$ eval $(une_commande)
total 4
drwxr-xr-x 6 toto users 1024 Sep 28 13:54 X/
drwxr-xr-x 8 toto users 1024 Aug 20 00:48 c/
drwxr-xr-x 5 toto users 1024 Mar 18 1996 java/
drwxr-xr-x 15 toto users 1024 Nov 9 17:44 sh/
```

Note : *eval* sert aussi à exécuter certaines commandes trop complexes pour que le shell les comprenne en une seule passe.

1.9.Utilisation des parenthèses et des accolades

```
{ commande1 ; commande2; }, ( commande1; commande2 )
```

Les parenthèses et accolades permettent d'encadrer une série de commandes séparées les unes des autres par "&", "&&", "|", "||" ou " ;".

L'usage des parenthèses ou des accolades permet à l'ensemble des commandes qu'elles contiennent de subir une redirection. En effet, avec la commande "**ls src ; ls bin** | **lpr**", seul le contenu du répertoire bin est imprimé car une redirection ne s'applique qu'à la commande qui précède. Par contre, la commande " {**ls src ; ls bin ;**} | **lpr**" provoque bien l'impression du contenu des répertoires src et bin.

On préférera généralement les accolades aux parenthèses car les instructions situées entre parenthèses s'exécutent dans un shell secondaire indépendant du shell actuel (voir l'utilisation du tube, plus haut).

Attention, la dernière commande de la liste entre accolades doit être terminée par un point-virgule.

1.9.1. Ecriture à l'écran

La commande "**echo texte**" permet d'écrire un texte à l'écran. Bien que ce ne soit pas nécessaire, pour éviter tout problème il est vivement conseillé d'encadrer le texte à afficher par des guillemets.

Si le texte à afficher est une chaîne vide, la commande provoque un saut de ligne.

- L'option **-n** évite le saut de ligne automatique à la fin du texte.

- L'option **-e** permet l'interprétation des caractères spéciaux utilisés avec printf(), tels l'alarme, le retour à la ligne, la tabulation, etc...

Exemples :

```
echo "Texte à
afficher"
echo ""
echo -n "Ce texte ne sera pas suivi d'un saut de ligne"
echo -e "\t\aCe texte commence par une tabulation et émet un signal
sonore."
```

1.9.2. Lecture au clavier

La commande **read** utilisée seule permet de lire une phrase complète. La phrase lue est stockée dans la variable **REPLY**.

La commande "**read mot1 mot2 reste**" permet aussi de lire une phrase au clavier mais son premier mot est affecté à la variable mot1, son deuxième mot est affecté à mot2 et le reste de la phrase est affecté à la variable reste.

Exemple :

```
toto> echo -n "Entrez votre nom de login et votre nom civil: "; read log
nom
Entrez votre nom de login et votre nom civil: moliere Jean-Baptiste
Poquelin
toto> echo "$log est le nom de login de $nom"
moliere est le nom de login de Jean-Baptiste Poquelin
```

Attention : une opération du style "**cat fichier | read ligne**" ne fonctionnera pas car l'instruction read (tout comme l'instruction cat) sera exécutée dans un shell indépendant et le contenu de la variable ligne sera perdu dès la fin de l'opération. Par contre "**read ligne < fichier**" fonctionnera correctement.

1.9.3. La commande select

La commande **select** permet d'offrir un menu à l'utilisateur. Sa syntaxe est :

```
select variable in "choix1" "choix2" ... "choixN"
do
<liste d'instructions>
done
```

Le menu présenté comporte l'ensemble des choix proposés précédés d'un numéro.

Une invite est affichée et le shell attend une entrée au clavier. La variable PS3 doit contenir la chaîne d'invite ("# ? " par défaut).

Lorsque l'utilisateur a entré son choix, le corps de la boucle est exécuté. La variable REPLY contient alors ce que l'utilisateur a rentré (un numéro normalement). Si le choix de

l'utilisateur correspond à l'une des propositions qui lui sont faites, la variable "variable" contient aussi la chaîne correspondante ("choix1", "choix2", ... ou "choixN").

Exemple :

```
#! /bin/sh
#
# menu
#
# Shell-script de mise en application de l'instruction select.
PS3="Entrez le numéro de votre commande -> "
echo "Que désirez-vous boire ?"
select boisson in "Rien, merci" "Café" "Thé
au lait" "Chocolat"; do
if [ -z "$boisson" ]; then
echo "Erreur: entrez un des chiffres proposés." 1>&2
elif [ "$REPLY" -eq 1 ]; then
echo "Au revoir!"
break
else
echo "Vous avez fait le choix numéro $REPLY..."
echo "Votre $boisson est servi."
fi
echo
done
```

Test:

```
toto> ./menu
Que désirez-vous boire ?
1) Rien, merci
2) Café
3) Théau lait
4) Chocolat
Entrez le numéro de votre commande -> 3

Vous avez fait le choix numéro 3...
Votre Thé au lait est servi.
1) Rien, merci
2) Café
3) Thé au lait
4) Chocolat
Entrez le numéro de votre commande -> 1
Au revoir!
toto>
```

1.9.4. Redirections des entrées-sorties standard

Avant tout, il faut savoir que toute redirection ne s'applique qu'à la dernière commande.

Dans le cas d'un série de commandes séparées par "&", "&&", "|", "||" ou " ;", si l'on veut que la redirection s'applique à l'ensemble des commandes, il faut employer des parenthèses (voir la partie "commandes composées").

- L'entrée standard (stdin) peut être redirigée afin de lire dans un fichier plutôt qu'au clavier. Pour cela, placez "< **fichier**" en fin de commande. Exemple : pour lire la première ligne d'un fichier

```
toto> read < mon_script; echo $REPLY
#! /bin/sh
toto>
```

- La sortie standard (stdout) peut être redirigée afin d'écrire dans un fichier plutôt qu'à l'écran. Pour cela, placez "> **fichier**" ou ">> fichier" en fin de commande. Si l'opérateur ">>" est utilisé, le contenu actuel du fichier est conservé et le texte est ajouté en fin de fichier.

Exemple :
Pour ajouter une ligne à la fin d'un fichier :

```
toto> echo "FIN DU FICHIER" >> fichier
toto>
```

- La sortie standard d'erreur (stderr) peut être redirigée afin d'écrire dans un fichier plutôt qu'à l'écran. Pour cela, placez "2> fichier" en fin de commande.

Exemple :
Pour supprimer les messages d'erreur de l'affichage :

```
./ma_commande 2> /dev/null
```

- La sortie standard et la sortie standard d'erreur peuvent être redirigées ensemble afin d'écrire dans un fichier plutôt qu'à l'écran. Pour cela, placez "**&**> **fichier**" en fin de commande.

Exemple :
Pour qu'une commande n'affiche rien, même pas les erreurs

```
ma_commande &> /dev/null
```

- La sortie standard peut être redirigée vers la sortie standard d'erreur en ajoutant "**1**>**&2**" à la fin de la commande.

Exemple :
Pour afficher un message sur stderr plutôt que sur stdout :

```
echo "Erreur fatale, le disque est plein!" 1>&2
```

- La sortie standard d'erreur peut être redirigée vers la sortie standard en ajoutant "**2**>**&1**" à la fin de la commande.

Exemple :
Pour afficher le résultat d'une commande page par page, y compris les erreurs :

```
ma_commande 2>&1 | more
```

- L'entrée standard peut être lue depuis le shell-script en ajoutant "<<**FIN DU TEXTE**" à la fin de la commande. Toutes les lignes comprises entre la commande et la ligne "FIN DU TEXTE" seront interprétées comme si elles avaient été lues au clavier. Le texte à afficher peut contenir des variables voire même l'expression "$(commande)".

Exemple :
Pour afficher plusieurs lignes de texte sans utiliser echo à chaque fois :

```
cat << FIN_DU_TEXTE
Vous êtes l'utilisateur $USER.
Vous êtes situé dans le répertoire $(pwd).
Merci de votre visite.
FIN_DU_TEXTE
```

- La redirection par tube et l'emploi de l'expression "`$(xxx)`" ont déjà été décrits dans la partie "Commandes composées".

1.10. Variables et paramètres

1.10.1. Variables

Contrairement au DOS qui ne connaît que les variables d'environnement, les shells d'Unix font la distinction entre variables simples et variables d'environnement.

La différence entre les deux est que seules les variables d'environnement sont transmises aux programmes lancés depuis le shell (exception : lorsqu'on utilise le tube ou les parenthèses dans une commande composée, toutes les variables sont communiquées aux sous-shells qui sont lancés implicitement - par contre, ce qui suit reste valable).

La modification d'une variable n'est perçue que par le shell qui la modifie et par ses descendants (c'est-à-dire les commandes qu'il exécute) lorsque c'est une variable d'environnement.

Il s'ensuit qu'un script ne pourra jamais modifier les variables du shell courant, sauf s'il est appelé avec la commande source (voir plus haut "Invocation d'un script").

En Bourne-shell, définir une variable se fait par la commande : **variable="valeur"**.
Il est important que valeur soit spécifiée entre guillemets ou apostrophes, et qu'il n'y ait d'espaces ni avant ni après le signe '='. Si valeur est absente, la variable est créée mais contient une chaîne vide.

L'utilisation d'une variable se fait grâce à l'expression $variable ou ${variable}.
La deuxième notation est nécessaire si la variable est suivie d'un texte qui pourrait être interprété comme faisant partie du nom de la variable.

Exemple :

```
toto> jour=10; mois=11; an=1997; heures=19; minutes=30
toto> echo "Nous sommes le $jour/$mois/$an"
Nous sommes le 10/11/1997
toto> echo "Il est ${heures}h ${minutes}mn"
Il est 19h 30mn

toto>
```

Définir une variable d'environnement se fait par la commande "**export *variable=valeur***". Transformer une variable en variable d'environnement se fait par la commande "**export variable**". Afficher la liste des variables d'environnement se fait par la commande "**export**". Afficher la liste complète des variables se fait par la commande "**set**". La commande "**unset var**" permet de détruire définitivement la variable var.

Au lancement du shell, beaucoup de variables sont prédéfinies, qu'elle soient d'environnement ou non. Par exemple, USER contient le nom de l'utilisateur, HOME contient le chemin de son répertoire personnel, etc... Pour en avoir la liste complète, faites "**set**" ou "**export**" dès le démarrage du shell, ou consultez le manuel en ligne de sh ("man sh" ou "man bash").

Sous Bourne-shell, certaines variables ont un sens spécial :

- `$$` donne le numéro de processus (pid) du shell.
- `$!` donne le numéro de processus (pid) de la dernière commande lancée en tâche de fond (c'est-à-dire avec l'opérateur '&').
- `$?` donne la valeur retournée par la dernière commande.
- `$-` donne la liste des options avec lesquelles le shell a été appelé.

1.10.2. Paramètres

La variable **`$\#`** donne le nombre de paramètres accompagnant l'appel du shell-script. Les variables **`$@`** et **`$*`** donnent l'ensemble des paramètres. La différence entre les deux termes ne s'observe que lorsqu'ils sont donnés entre guillemets. En effet, "**`$*`**" correspond à une seule valeur contenant tous les paramètres alors que "**`$@`**" correspond à autant de valeurs qu'il y a de paramètres.

On peut accéder à chaque paramètre en spécifiant son numéro après le signe **`$`**. Si le numéro tient sur deux chiffres ou plus, il doit être donné entre accolades.

Exemples :
`$1, ..., $9, ${10}, ${11}, ...\` Le paramètre `$0` correspond au nom complet (avec le chemin) du shell- script.

La commande "**shift N**" élimine les N premiers paramètres de la liste des paramètres

(N ne doit pas être supérieur au nombre de paramètres). La variable **$#** est diminuée d'autant.

- Après une commande shift, les paramètres sont donc décalés, c'est à dire que **$1** prend la valeur de **$(N1)+**, **$2** prend la valeur de **$(N2)+**, etc...

Exemple :

```
#! /bin/sh
#
# params
#
# Shell-script de mise en application des paramètres d'un script.
# Affiche l'ensemble des paramètres passés au script.
echo "Vous avez lancé
le shell-script $0"
echo "Vous lui avez fourni $# paramètres qui sont: $*"
echo ""
echo "Liste des paramètres :"
echo "----------------------"
numParam=1
for parametre in "$@"; do
echo "Paramètre $numParam = $parametre"
let $[numParam += 1]
done
echo ""
echo -n "Après la commande 'shift $#', "
shift $#
echo "il reste $# paramètres"
Test :
toto> ./params A BC DEF "GH IJ"
```

Vous avez lancé le shell-script en lui fournissant les 4 paramètres qui sont: A BC DEF GHIJ:

```
toto>./params A BC DEF GHIJ

Liste des paramètres:
---------------------
Paramètre 1 = A
Paramètre 2 = BC
Paramètre 3 = DEF
Paramètre 4 = GH IJ
Après la commande 'shift 4', il reste 0 paramètres
toto>
```

1.11. Calcul mathématique

Le shell permet d'effectuer des calculs mathématiques mais uniquement sur des entiers.

```
expr entier opérateur entier
```

L'expression mathématique peut faire intervenir n'importe quelle variable (pas forcément précédée du signe $) ainsi que n'importe quel nombre entier décimal (219), hexadécimal (0xDB ou 16#DB), octal (0333 ou 8#333) ou binaire (2#011011011). Tous les opérateurs du langage C sont autorisés (+, -, *, %, ||, >>, =, +=, ^=, ==, !=, etc...), y compris les parenthèses.

Exemple :

```
toto> expr 1 + 2
3

toto> expr 6 / 3
2
```

1.12. Les structures conditionnelles

1.12.1. Les conditions

La condition d'exécution des structures conditionnelles qui suivent est qu'une commande aboutisse, c'est à dire que sa valeur de retour soit 0.

La commande qui sert de condition peut être une commande composée, en particulier on utilise souvent une formule du type "**commande1 && commande2**" ou "**commande1** || commande2".

La condition peut être inversée si on fait précéder la commande d'un point d'exclamation. Donc " **! commande**" signifie que la condition est vraie si la commande échoue.
Attention, le point d'exclamation s'applique uniquement à la commande qui le suit ; pour qu'il s'applique à une commande composée, il faut l'encadrer avec des accolades (ou des parenthèses).

Exemples :

```
if commande; then echo "La commande a fonctionné"; fi
```

```
if ! commande; then echo "La commande a échoué"; fi
```

```
if commande1 && ! commande2; then \\
echo "commande1 a fonctionné mais pas commande2"; fi
```

```
if ! { commande1 || commande2; } ; then \
echo "Ni commande1 ni commande2 n'ont fonctionné"; fi
```

1.12.2. Les tests

La condition d'exécution devant être une commande, les tests s'effectuent par l'intermédiaire de la commande "**test expression**" qui peut être abrégée par la formule "**[**expression]".

Les tests possibles peuvent porter sur des entiers, des chaînes de caractères ou des fichiers/répertoires.

1.12.3. Tests sur les fichiers (et sur les répertoires)

- "-e fichier" : Vrai si le fichier/répertoire existe.

- ”-s fichier” : Vrai si le fichier à une taille supérieure à 0.
- ”-r fichier” : Vrai si le fichier/répertoire est lisible.
- ”-w fichier” : Vrai si le fichier/répertoire est modifiable.
- ”-x fichier” : Vrai si le fichier est exécutable ou si le répertoire est accessible.
- ”-O fichier” : Vrai si le fichier/répertoire appartient à l’utilisateur.
- ”-G fichier” : Vrai si le fichier/répertoire appartient au groupe de l’utilisateur.
- ”-b nom” : Vrai si nom représente un périphérique (pseudo-fichier) de type bloc (disques et partitions de disques généralement).
- ”-c nom” : Vrai si nom représente un périphérique (pseudo-fichier) de type caractère (terminaux, modems et port parallèles par exemple).
- ”-d nom” : Vrai si nom représente un répertoire.
- ”-f nom” : Vrai si nom représente un fichier.
- ”-L nom” : Vrai si nom représente un lien symbolique.
- ”-p nom” : Vrai si nom représente un tube nommé.
- ”fichier1 -nt fichier2” : Vrai si les deux fichiers existent et si fichier1 est plus récent que fichier2.
- ”fichier1 -ot fichier2” : Vrai si les deux fichiers existent et si fichier1 est plus ancien que fichier2.
- ”fichier1 -ef fichier2” : Vrai si les deux fichiers représentent un seul et même fichier.

1.12.4. Tests sur les entiers

Il y a erreur si ce ne sont pas des entiers autour de l’opérateur.

- ”entier1 -eq entier2” : Vrai si entier1 est égal à entier2.
- ”entier1 -ge entier2” : Vrai si entier1 est supérieur ou égal à entier2.
- ”entier1 -gt entier2” : Vrai si entier1 est strictement supérieur à entier2.
- ”entier1 -le entier2” : Vrai si entier1 est inférieur ou égal à entier2.
- ”entier1 -lt entier2” : Vrai si entier1 est strictement inférieur à entier2.
- ”entier1 -ne entier2” : Vrai si entier1 est différent de entier2.

1.12.5. Tests sur les chaînes

Encadrer la chaîne par des guillemets !

- ”-n ”chaîne”” : Vrai si la chaîne n’est pas vide.
- ”-z ”chaîne”” : Vrai si la chaîne est vide.
- ””chaine1” = ”chaine2”” : Vrai si les deux chaînes sont identiques.

- ””chaine1” != ”chaine2”” : Vrai si les deux chaînes sont différentes.

1.12.6. Combinaison de tests

Lorsqu’un test doit être inversé ou lorsque plusieurs tests doivent être combinés par des ET et des OU, on a le choix entre deux notations. La première a été présentée dans la partie précédente (”Les conditions”) ; elle est préférable car elle est plus lisible. La deuxième fait partie de la syntaxe de la commande test :

- ”NON expr” se traduit par ”test ! expr” ou ”[! expr]”.
- ”expr1 OU expr2” se traduit par ”test expr1 -o expr2” ou ”[expr1 -o expr2]”.
- ”expr1 ET expr2” se traduit par ”test expr1 -a expr2” ou ”[expr1 -a expr2]”.

1.12.7. Structure Si...Alors...Sinon

En Bourne-shell, la structure algorithmique se traduit par :

Si commande1 aboutit, alors if commande1; then
action1 action1
Sinon si commande2 aboutit, alors elif commande2; then
action2 action2
Sinon else
action3 action3
FinSi fi

Bien sûr, chacune des parties ”sinon si” et ”sinon” est optionnelle.
D’autre part les actions à effectuer peuvent être composées de plusieurs commandes sur une ou plusieurs lignes.

A l’extrême, on peut tout placer sur une seule ligne (bien respecter les points-virgules) mais c’est fortement déconseillé car cela nuit à la lisibilité :

```
if commande1; then action1; elif commande2; then action2; else action3; fi
```

Exemple :

```
#! /bin/sh
#
# signe
#
# Shell-script de mise en application de l’instruction if...then...else.
# Affiche le signe d’un nombre et de son opposé.
#
# Saisie de n
#
echo -n "Entrez un nombre: "
read n
#
# Test du signe de n
#
```

```
if test $n -lt 0; then # n < 0 ?
echo "Le nombre $n est négatif"
elif test $n -gt 0; then # n > 0 ?
echo "Le nombre $n est positif"
else # n = 0
echo "Le nombre $n est nul"
fi
#
# Changement du signe de n
#

let $[ n = -n ]
echo "Le nombre choisi devient $n"
#
# Test du signe de n (autre méthode)
#
if [ $n -le 0 ]; then # n <= 0 ?
if [ ! $n -eq 0 ]; then # n != 0 ? C'est à
dire n < 0 ?
echo "Le nombre $n est négatif"
else # n = 0
echo "Le nombre $n est nul"
fi
else # n > 0
echo "Le nombre $n est positif"
fi
```

Test :

```
toto> ./signe
Entrez un nombre: 3
Le nombre 3 est positif
Le nombre choisi devient -3
Le nombre -3 est négatif
toto> ./signe
Entrez un nombre: 0
Le nombre 0 est nul
Le nombre choisi devient 0
Le nombre 0 est nul
toto>
```

1.12.8. Structure Répéter...Jusqu'à

Cette structure n'existe pas en Bourne-shell. Attention aux confusions, le mot-clé until est lié à une structure TantQue, comme on le voit dans ce qui suit.

1.12.9. Structure TantQue

En Bourne-shell, la structure algorithmique se traduit par :

TantQue commande aboutit while commande; do
action action
FinTQ done
TantQue commande échoue until commande; do
action action

FinTQ done

L'action à effectuer peut être composée de plusieurs commandes sur une ou plusieurs lignes.

A l'extrême, on peut tout placer sur une seule ligne (bien respecter les points-virgules) mais c'est fortement déconseillé car cela nuit à la lisibilité :

```
while commande; do action; done
```

```
until commande; do action; done
```

Exemple :
Pour while :

```
#! /bin/sh
#
# factorielle
#
# Shell-script de mise en application de l'instruction while.
# Affiche la factorielle du nombre donné en paramètre.
if [ $# -ne 1 ] || [ $1 -lt 0 ]; then
echo "Usage: factorielle n (avec n >= 0)" 1>&2
else
resultat=1
n=$1
while [ $n -gt 1 ]; do
let $[ resultat *= n]
let $[ n -= 1]
done
echo "$resultat"
fi
```

Test:

```
toto> factorielle 3
6
toto> factorielle 6
720
toto> factorielle $(factorielle 3)
720
Exemple : pour until
#! /bin/sh
#
# testfact
#
# Shell-script de mise en application de l'instruction until.
# Teste le shell-script de calcul de factorielles.
n=0
until [ $n -eq 14 ]; do
resultat=$(factorielle $n)
echo "$n! = $resultat"
let $[ n += 1 ]
done
```

Test:

```
toto> ./testfact
0! = 1
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
toto>
```

1.12.10. Autres structures exécutives

1.12.11. Structure Pour

Pour chaque élément de l'ensemble (element1, element2, ..., elementN) action sur element
FinPour

En Bourne-shell, la structure algorithmique se traduit par :

for element in element1 element2 ... elementN; do
action $element
done

L'action à effectuer peut être composée de plusieurs commandes sur une ou plusieurs lignes.

A l'extrême, on peut tout placer sur une seule ligne (bien respecter les points-virgules) mais c'est fortement déconseillé car cela nuit à la lisibilité :

for element in element1 element2 ... elementN; do action $element; done

Exemple :

```
#! /bin/sh
#
# fic_ou_rep
#
# Shell-script de mise en application de l'instruction for.
# Affiche le type (fichier ou répertoire) de chaque paramètre.
for param in "$@"; do # Pour chaque paramètre...
if [ -f "$param" ]; then # C'est un fichier?
echo "$param est un fichier."
elif [ -d "$param" ]; then # C'est un répertoire?
echo "$param est un répertoire."
elif ! [ -e "$param" ]; then # N'existe pas
echo "$param n'existe pas."
else # C'est autre chose?
echo "$param n'est ni un fichier, ni un répertoire."

fi
```

```
done
```

Test :

```
toto> ./fic_ou_rep XXX /etc "./un rep/" /etc/hosts "./un rep/un fic"
/dev/lp0
XXX n'existe pas.
/etc est un répertoire.
./un rep/ est un répertoire.
/etc/hosts est un fichier.
./un rep/un fic est un fichier.
/dev/lp0 n'est ni un fichier, ni un répertoire.
toto>
```

Remarque : sous Unix, les espaces sont des caractères légaux (mais pas conseillés) dans les noms de fichiers et de répertoires. Sans les guillemets dans le script et sur la ligne de commande, "./un rep/" aurait été interprété comme deux mots distincts : "./un" et "rep/".

1.12.12. Structure Selon

En Bourne-shell, la structure algorithmique se traduit par :

Selon valeur case valeur in
cas X : X)
action1 action1;;
cas Y :
cas Z : Y $|$ Z)
action2 action2;;
defaut : *)
action3 action3;;
FinSelon esac

Les actions à effectuer peuvent être composées de plusieurs commandes sur une ou plusieurs lignes. Pour chaque cas, la série de commandes à exécuter doit être terminée par deux points-virgules que l'on placera de préférence sur une êseule ligne.
Les cas à tester peuvent employer les caractères génériques autorisés pour les fichiers.

Par exemple, les cas a ? ? ?b, A*B*C et [a-z]*[0-9] sont tout à fait valides. Si toutefois un caractère faire partie d'un cas à tester, il faudra l'encadrer par des guillemets pour qu'il ne soit pas interprété.

Exemple :

```
#! /bin/sh
#
# question
#
# Shell-script de mise en application de l'instruction case.
# Pose une question et attend une réponse par oui ou non.
# La valeur retournée est 0 pour oui et 1 pour non.
retour=X
```

```
while [ "$retour" = "X" ]; do
echo -n "On continue (O/N) ? "
read reponse
case "$reponse" in
o* | O* )
retour=0
;;
[nN]* )
retour=1
;;
"?"* )
echo "Il n'y a pas d'aide disponible"
;;
* )
echo "Erreur : entrez [O]ui ou [N]on."
;;
esac
echo ""
done
exit $retour
```

Test :

```
toto> question ; echo "Retour = $?"
On continue (O/N) ? bof
Erreur: entrez [O]ui ou [N]on.
On continue (O/N) ? oui, je veux bien
Retour = 0
toto> question ; echo "Retour = $?"
On continue (O/N) ? ???
Il n'y a pas d'aide disponible
On continue (O/N) ? Non, merci
Retour = 1
toto>
```

1.12.13. Structure Lorsque

Grâce à la commande **trap**, on peut déclencher une action sur un événement particulier. La contrainte est que cet événement soit un signal que l'on peut intercepter (faire "trap -l" pour en avoir la liste et "man 7 signal" pour obtenir des détails supplémentaires sur ces signaux).

En Bourne-shell, la structure algorithmique se traduit par :

Lorsque le signal SIGNAL survient trap "action" SIGNAL
action
FinLorsque

L'action peut être une commande composée. Dès qu'elle ne se limite pas à un seul mot, elle doit être précisée entre guillemets. Les guillemets et apostrophes situés à l'intérieur doivent être précédés par un anti-slash.Bien sûr, plusieurs signaux peuvent être précisés en une seule fois.

La commande trap utilisée seule donne la liste des actions exécutées pour chaque signal. La commande "**trap SIGNAL**" annule l'action à exécuter à l'arrivée du signal SIGNAL.

Exemple :

```
#! /bin/sh
#
# exTrap
#
# Shell-script de mise en application de l'instruction case.
trap "echo Le script s\'est terminé" EXIT
trap "echo Vous avez appuyé
sur Ctrl-C" SIGINT
trap "echo Vous avez fait: kill $$" SIGTERM
trap "echo J\'ai été
stoppé
et je continue" SIGCONT
trap "echo J\'ai reçu SIGUSR1 ou SIGUSR2" SIGUSR1 SIGUSR2
echo "Processus $$: J'attend un signal..."
#
# Compte à
rebours
#
i=10
while [ $i -gt 0 ]; do
echo -n "$i "
sleep 1
let $[ i -= 1 ]
done
echo "0"
```

Test :

```
toto> ./exTrap
Processus 1658: J'attend un signal...
10 9 Vous avez appuyé sur Ctrl-C
8 7
[1]+ Stopped exTrap
toto> kill -SIGUSR1 1658
toto> fg
exTrap
J'ai reçu SIGUSR1 ou SIGUSR2
J'ai été stoppé et je continue
6 5 4
[1]+ Stopped exTrap
toto> kill -SIGUSR2 1658
toto> kill 1658
toto> fg
exTrap
J'ai reçu SIGUSR1 ou SIGUSR2
Vous avez fait: kill 1658
J'ai été stoppé et je continue
3 2 1 0
Le script s'est terminé
toto>
```

1.13. Fonctions

Le Bourne-shell offre la possibilité de définir ses propres fonctions. Bien qu'elles soient internes à un shell, elles s'emploient comme un script externe. Mais leur appel ne provoque pas le lancement d'un sous-shell, donc une fonction a accès à toutes les variables, pas

seulement celles d'environnement, et leur modification reste prise en compte lorsque la fonction se termine.

1.13.1. Syntaxe

```
function maFonction()
{
local var1
local var2="valeur"
commande1
commande2
return val;
}
```

- Le mot-clé "**local**" permet de définir des variables locales à la fonction.
- La dernière commande doit être terminée par un point-virgule.
- On accède aux paramètres d'une fonction comme à ceux d'un script : grâce aux variables `$*, $@, $#, $1, $2, ...` qui sont temporairement modifiées pendant toute la durée de la fonction. En revanche, `$0` ne change pas.
- Le retour d'une valeur s'effectue grâce au mot-clé return. Si return n'est pas employé, la valeur de retour est celle de la dernière commande exécutée.

Attention, l'emploi de la commande exit termine non seulement la fonction mais aussi le script.

- Une fonction peut être récursive, c'est àdire qu'elle peut s'appeler elle-même.
- Une fois définie, une fonction apparaît dans la liste des variables. On peut l'exporter vers les autres shells grâce à la commande "export -f maFonction".

Exemple :

```
#! /bin/sh
#
# signe2
#
# Shell-script de mise en application des fonctions.
# Affiche le signe d'un nombre et de son carré.
#
#
# carré(nombre)
#
# Retourne le carré
du nombre donné
en paramètre.
#
# Paramètres: $1 -> Le nombre dont on veut le carré
#

function carré()
{
```

```
return $[ $1 * $1 ]
}
#
#
# affSigne(nombre)
#
# Affiche le signe du nombre donné
en paramètre.
#
# Paramètres: $1 -> Le nombre dont on teste le signe.
#
function affSigne()
{
local signe
if [ $n -lt 0 ]; then # n < 0 ?
signe=négatif
elif [ $n -gt 0 ]; then # n > 0 ?
signe=positif
else # n = 0
signe=nul
fi
echo "Le nombre $1 est $signe"
}
#
#
# Programme principal
#
echo -n "Entrez un nombre: "
read n # Saisie de n
affSigne $n # Affichage du signe de n
carré
$n # Calcul du carré
de n
n=$? # Récupération du résultat
echo "Le carré
du nombre choisi est $n"
affSigne $n # Affichage du signe de n
```

Test :

```
toto> ./signe2
Entrez un nombre: -5
Le nombre -5 est négatif
Le carré du nombre choisi est 25
Le nombre 25 est positif
```

```
toto> ./signe2
Entrez un nombre: 0
Le nombre 0 est nul
Le carré du nombre choisi est 0
Le nombre 0 est nul
```