2e Partie Cours LINUX

7.1.3 La ligne de commandes séquentielles

Il est possible de taper plusieurs commandes sur la même ligne en les séparant par des points-virgules ;. Les commandes sont exécutées séquentiellement, de façon totalement indépendante, la première n'influençant pas la seconde et ainsi de suite.

Exemple

xstra> **pwd ; who ; ls**
/home/xstra
xstra ttyp0 Mar 15 10:32
xstra1 co Mar 15 09:12
bin projet1 projet2 essai
xstra>

Cette commande affiche le répertoire courant de l'utilisateur, donne la liste des utilisateurs connectés, puis liste les fichiers du répertoire courant.

7.1.4 La commande sur plus d'une ligne

Il est possible de taper une commande sur plusieurs lignes. Pour cela les lignes de commandes, sauf la dernière, doivent se terminer par la suite de touches \<*return*>.

Exemple

xstra> **ls l /home/xstra/develop\<return>**
/essai.f <return>
rwx r 1 xstra staff 258 Jan 15 16:42 essai.f
xstra>

Cette commande est équivalente à :

xstra> **ls l /home/xstra/develop/essai.f <return>**

Elle liste les droits d'accès du fichier *essai.f* qui se trouve dans le répertoire */home/xstra/develop*.

7.1.5 Les séparateurs conditionnels de commandes

Il est possible de contrôler la séquence d'exécution de commandes en utilisant des séparateurs conditionnels. Le séparateur **&&** permet d'exécuter la commande qui le suit si et seulement si la commande qui le précède a été exécutée sans erreur (code retour du processus nul). Le séparateur || permet d'exécuter la commande qui le suit si et seulement si la commande qui le précède a été exécutée avec erreur (code retour du processus différent de 0).

Exemple

Suppression des fichiers si la commande *cd projet1* a été correctement exécutée.

xstra> **cd projet1 && rm ***

Exemple

Si le répertoire *projet1* n'existe pas, alors il sera créé par la commande *mkdir*.

xstra> **cd projet1 || mkdir projet1**
bash: cd: projet1: No such file or directory
xstra> **ls**
projet1
xstra>

7.2 LA REDIRECTION DES ENTRÉES-SORTIES

7.2.1 Le principe de redirection

On appelle processus, ou tâche, l'exécution d'un programme exécutable. Au lancement de chaque processus, l'interpréteur de commandes ouvre d'office une entrée standard (par défaut le clavier), une sortie standard (par défaut l'écran) et la sortie d'erreur standard (par défaut l'écran) (Fig. 7.1).

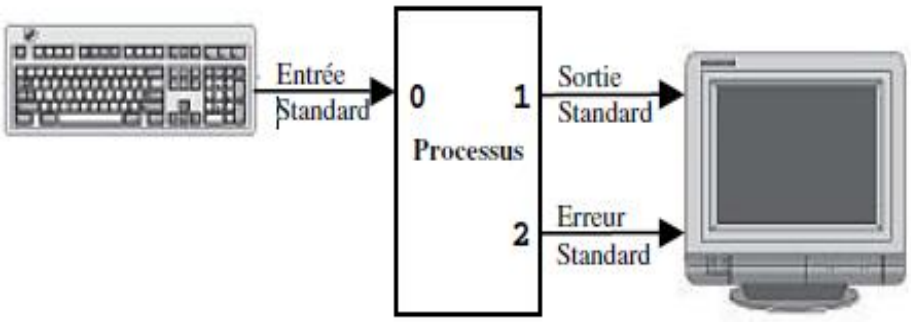

FIGURE 7.1. ENTRÉES/SORTIES STANDARD D'UN PROCESSUS.

Ces entrées-sorties standard peuvent être redirigées vers un fichier, un tube, un périphérique. La redirection de la sortie standard consiste à renvoyer le texte qui apparaît à l'écran vers un fichier (Fig. 7.2). Aucune information n'apparaîtra à l'écran, hormis celles qui transitent par la sortie d'erreur standard.

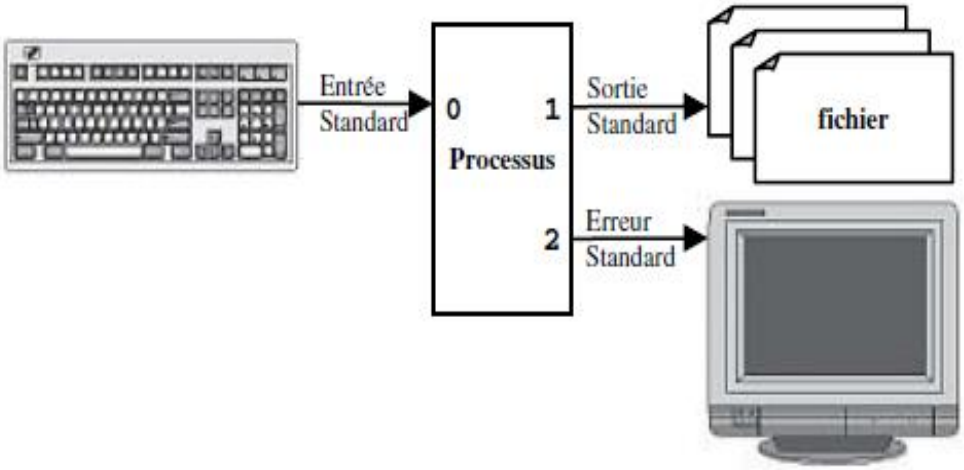

FIGURE 7.2. REDIRECTION VERS LE FICHIER *FS* DE LA SORTIE STANDARD D'UN PROCESSUS.

Il est naturellement possible de rediriger toutes les entrées-sorties standard d'un processus. Par conséquent, le processus recherchera les informations dont il a besoin dans un fichier et non plus au clavier. Il écrira dans des fichiers ce qui devait apparaître à l'écran (Fig. 7.3).

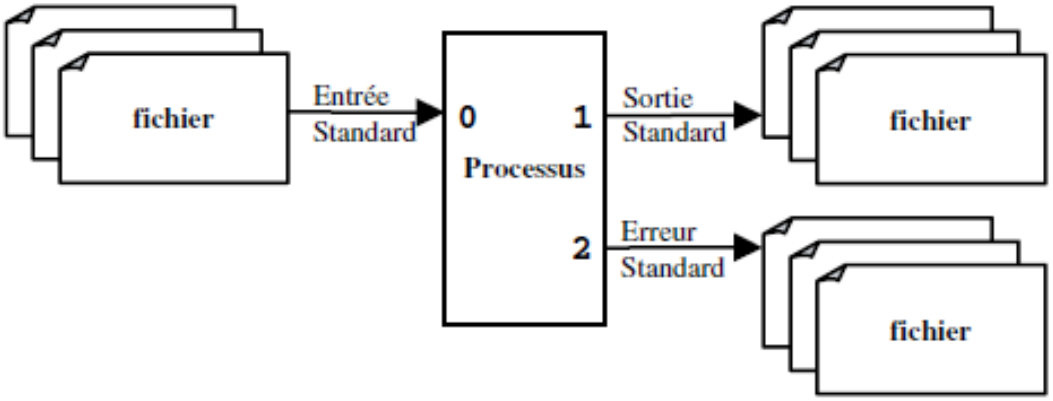

FIGURE 7.3. REDIRECTION VERS DES FICHIERS DE TOUTES LES ENTRÉES/SORTIES STANDARD D'UN PROCESSUS.

La redirection des sorties peut être réalisée par effacement et création du fichier ou par ajout à la fin du fichier si ce dernier existe. Dans le cas contraire, un nouveau fichier sera créé. Dans le cas de la redirection de l'entrée, il est évident que le fichier doit exister. Le tableau 7.1 résume les différentes redirections.

	Le fichier existe-t-il?	
	OUI	NON
Redirection de l'entrée standard	lit le fichier	erreur
Redirection de la sortie standard et de la sortie erreur standard	effacement et création du fichier	création du fichier
Concaténation de redirection de la sortie et de l'erreur standard	ajoute à la fin du fichier	création du fichier

TABLEAU 7.1. RÉSUMÉ DES REDIRECTIONS.

Le caractère **<** suivi du nom d'un fichier indique la redirection de l'entrée standard à partir de ce fichier :

`<fe` Définition de `fe` comme fichier d'entrée standard.

Le caractère **>** suivi du nom d'un fichier indique la redirection de la sortie standard vers ce fichier :

`>fs`	Redirection de la sortie standard vers le fichier `fs` ; le fichier `fs` est créé ou écrasé s'il existe déjà et si la variable booléenne `noclobber` n'est pas initialisée.

Si on double le caractère >, l'information ou la redirection sera ajoutée au fichier `fs`.

`>>fs`	La sortie standard rallonge le fichier `fs`.

Les caractères ***2>*** suivis du nom d'un fichier indiquent la redirection de la sortie d'erreur standard vers ce fichier :

`2>erfs`	Redirection de la sortie d'erreur standard vers le fichier `erfs`.

Les caractères ***2>>*** suivis du nom d'un fichier indiquent que la redirection de la sortie d'erreur standard sera ajoutée au fichier :

`2>>erfs`	La sortie d'erreur standard est ajoutée au fichier `erfs`.

Si l'on souhaite rediriger la sortie standard et la sortie d'erreur standard, la syntaxe sera :

`commande >fs 2>erfs`	Dirige la sortie standard vers `fs` et la sortie d'erreur vers `erfs`.

Les caractères ***&>*** suivis du nom d'un fichier indiquent la redirection de la sortie standard et de la sortie d'erreur standard vers ce fichier :

`&>erfs`	Redirection des sorties standard et de la sortie d'erreur standard vers le fichier `erfs`.

Exemples

```
xstra> ls >temp
    § Range, dans le fichier temp, la liste
    § des fichiers du répertoire courant.
    § Le fichier temp est créé s'il n'existe
    § pas ou écrasé s'il existe.
xstra> ls >>f.rep
    § Ajoute à f.rep la liste des fichiers
    § du répertoire courant.
xstra> ls /toto 2>temp
    § Range, dans le fichier temp,
    § les messages d'erreurs et affiche
    § à l'écran la liste des fichiers.
    § /toto  n'existe pas.
xstra> ls /toto * >temp 2>ertemp
```

7.3 LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

7.3.1 Les tubes

Un **tube** (**`pipe`** en anglais) est un flot de données qui permet de relier la sortie standard d'une commande à l'entrée standard d'une autre commande sans passer par un fichier temporaire (Fig. 7.4).

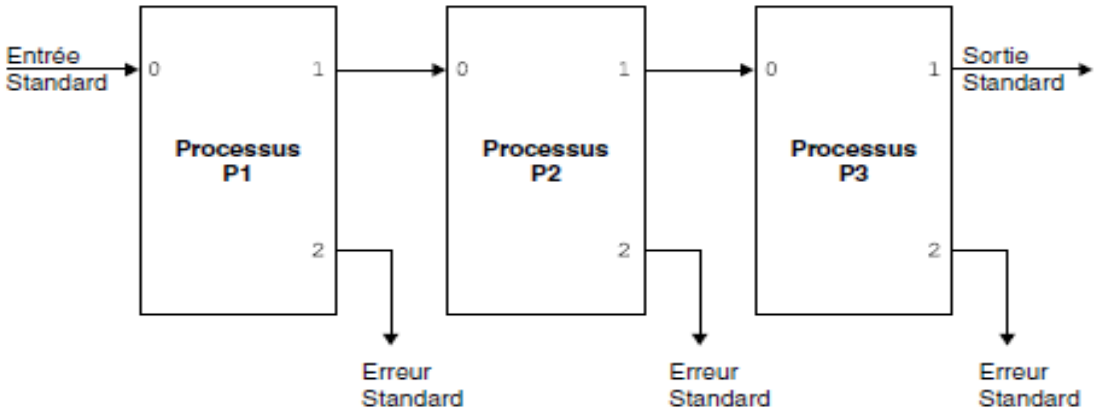

FIGURE 7.4. UN TUBE (PIPE).

Dans une ligne de commandes, le tube est formalisé par la barre verticale | , que l'on place entre deux commandes :

```
P1 | P2 | P3
```

Exemple 1

Affichage page par page du contenu du répertoire courant :

```
xstra> ls  l | less
```

Dans cet exemple, le résultat de la commande `ls  l` n'apparaît pas à l'écran : la sortie standard est redirigée vers l'entrée standard de la commande `less` qui, quant à elle, affichera son entrée standard page par page. La commande `less`, contrairement à la commande `more`, permet à l'aide des touches claviers ↓, ↑, <page-up> et <page-down> de monter et descendre dans le flot de données obtenu dans un tube.

Exemple 2

Affichage page par page du contenu du répertoire courant dont les protections sont `rwxr xr x :`

```
xstra> ls  l | grep "rwxr xr x"| less
```

Pour obtenir le même résultat en utilisant le mécanisme de redirection des entrées-sorties, il faut écrire :

```
xstra> ls  l >temp1; \<return>
grep "rwxr xr x" <temp1 >temp2 \<return>
; less temp2; rm temp1 temp2
```

Cette solution est non seulement plus lourde mais aussi plus lente. Les tubes ont deux qualités supplémentaires :

- Toutes les commandes liées par le tube s'exécutent en parallèle. C'est le système qui réalise la synchronisation entre les processus émetteurs et récepteurs,
- Il n'y a pas de limite de taille pour le flot de données qui transite dans le tube (il n'y a pas création de fichier temporaire).

Exemple 3

```
xstra> who | wc  l
```

Indique le nombre de personnes connectées au système.

```
xstra> ls | wc  w
```

Indique le nombre de fichiers dans le répertoire courant.

Un processus suspendu peut être réactivé à l'aide de la commande *bg %numéro job*. Un processus peut être supprimé à l'aide de la fonction *kill %numéro job*. Le paragraphe 12.5 présente ces mécanismes.

Le tableau 7.2 résume ces différentes possibilités.

7.5 LA SUBSTITUTION DE COMMANDE

La substitution de commande ou backquoting permet d'utiliser le résultat d'une commande comme argument d'une autre commande. Pour utiliser cette fonctionnalité, il faut entourer la commande soit d'accents graves, ou « backquotes » (*`commande`*), soit *$(commande)* . La forme *$(commande)* est plus récente et doit être préférée.

La commande placée entre *$(cmd)* est exécutée en premier, avant l'exécution de la ligne de commandes dont elle fait partie. Son résultat, c'est-à-dire la sortie standard, est entièrement intégré à la ligne de commandes en remplacement de la commande « backquotée ». La ligne de commandes est alors exécutée avec ces nouveaux arguments.

Exemple 1

La commande *echo* affiche à l'écran la chaîne de caractères qui la suit.

```
xstra> echo pwd
pwd
xstra> echo 'pwd'
pwd
xstra> echo `pwd`  § Ancien format
/home/xstra
xstra> echo $(pwd)
/home/xstra
xstra>
```

Dans ce dernier cas, la commande *pwd* qui indique le répertoire courant est exécutée. Son résultat devient l'argument de la commande *echo* (*echo /home/xstra*).

Exemple 2

```
xstra> less $(grep  1 'toto' *)
```

Cette commande permet de visualiser le contenu des fichiers du répertoire courant contenant au moins une fois la chaîne de caractères *toto*.

Exemple 3

```
xstra> echo il y a $(who | wc -l) utilisateurs connectes
```

7.6 LES COMMANDES GROUPÉES

La commande groupée est une succession de commandes séparées par le caractère `;` et considérées comme un ensemble. Cet ensemble, repéré par des parenthèses (…), est exécuté par un nouveau processus shell. Les commandes seront exécutées séquentiellement, sans influence les unes sur les autres.

Le résultat d'une commande groupée est cependant différent de celui qu'auraient les mêmes commandes réalisées séquentiellement.

Exemple

Suppression du fichier *temporaire* dans le répertoire *projet1*.

```
xstra> cd
xstra> (cd projet1; rm temporaire)
xstra> pwd
/home/xstra
 § Le répertoire courant est toujours /home/xstra
```

Même action que la commande groupée mais le répertoire courant sera *projet1*.

```
xstra> cd projet1; rm temporaire
xstra> pwd
/home/xstra/projet1
xstra>
```

La programmation en shell

Le shell est plus qu'un interpréteur de commandes : c'est également un puissant langage de programmation. Cela n'est pas propre à Linux ; tout système d'exploitation offre cette possibilité d'enregistrer dans des fichiers des suites de commandes que l'on peut invoquer par la suite. Mais aucun système d'exploitation n'offre autant de souplesse et de puissance que le shell Linux dans ce type de programmation. Le revers de cette médaille est que la syntaxe de ce langage est assez stricte et rébarbative. De plus, l'existence de plusieurs shells conduit à plusieurs langages différents. Sous Linux, un fichier contenant des commandes est appelé **script** et nous n'emploierons plus que ce terme dans la suite. De même nous utiliserons le terme **shell** pour désigner à la fois l'interpréteur de commandes et le langage correspondant (tout comme " assembleur " désigne à la fois le langage assembleur et le compilateur de ce langage).
Comme tout langage de programmation conventionnel, le shell comporte des instructions et des variables. Les noms de variables sont des chaînes de caractères ; leurs contenus sont également des chaînes de caractères. L'assignation (Bourne-shell, POSIX-shell et Bash) d'une valeur à une variable se fait par un nom ; la référence à cette variable se fait par son nom précédé du caractère *$*, comme dans :
mavariable bonjour § *assignation*
echo $mavariable § *référence*
Le jeu d'instructions lui-même comporte :
● toutes les commandes Linux,
● l'i

nvocation de programmes exécutables (ou de scripts) avec passage de paramètres,

● des instructions d'assignation de variables,
● des instructions conditionnelles et itératives,
● des instructions d'entrée-sortie.
Et bien entendu, les mécanismes de tubes et de redirections sont utilisables dans un script.

8.1 LA PROGRAMMATION DE BASE EN SHELL

Dans ce qui suit, on supposera que l'environnement de l'utilisateur est le Bash, et qu'il écrit ses scripts dans le langage de l'interpréteur de commandes Bash. Les bases de programmation exposées dans ce paragraphe peuvent être considérées comme communes à tous les interpréteurs de commandes issus de la famille des
Bourne-shell (Bourne-shell, POSIX-shell, Bash).

Attention
Toujours commencer un shell script par la ligne *#!/bin/bash*.

8.1.1 Le premier script

Création avec l'éditeur *vi* du fichier *listf* contenant la ligne *ls aCF*.
Un fichier ordinaire n'a pas le droit *x* (il n'est pas exécutable) à sa création, donc :
xstra> **chmod a+x listf §** *ajoute le droit x*
§ *pour tout le monde.*
Il peut donc être exécuté comme une commande :

8.1.2 Le passage des paramètres

Le script *listf* ne s'applique qu'au répertoire courant. On peut le rendre plus général en lui transmettant le nom d'un répertoire en argument lors de l'invocation.
Pour ce faire, les variables 1, 2, …, 9 permettent de désigner respectivement le premier, le deuxième, …, le neuvième paramètre associés à l'invocation du script.
a) Premier script avec passage de paramètres
Modifier le fichier *listf* de la façon suivante :

echo "contenu du repertoire $1 "
ls -aCF $1
L'exécution donne :
xstra> ***listf /tmp***
contenu du repertoire /tmp

b) Généralisation
Le nombre de paramètres passés en argument à un script n'est pas limité à 9 ; toutefois seules les neuf variables 1, …, 9 permettent de désigner ces paramètres dans le script.
La commande *shift* permet de contourner ce problème. Après *shift*, le ième paramètre est désigné par *$i 1*.

Exemple 1
Le script *echopara* contient :
echo $1 $2 $3
P1 $1
shift
echo $1 $2 $3
echo $P1
L'exécution donne :
xstra> **echopara un deux trois**
un deux trois
deux trois
un
xstra>
Cet exemple montre le comportement de *shift*, l'affectation d'une valeur à la variable *P1 (P1=$1)* et la référence à cette variable (*echo $P1*).

Exemple 2
Le script *echopara1* de décalage des paramètres contient :
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
shift
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
L'exécution donne :
xstra> **echopara1 1 2 3 4 5 6 7 8 9 10**
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 10
xstra>

8.1.3 Les variables spéciales
En plus des variables 1, 2, …, 9, le shell prédéfinit des variables facilitant la programmation.
0 contient le nom sous lequel le script est invoqué,

contient le nombre de paramètres passés en argument,
******* contient la liste des paramètres passés en argument,
? contient le code de retour de la dernière commande exécutée,
$ contient le numéro de process (PID) du shell (décimal).

Exemple 1
Le script *echopara2* contient :
echo $0 a ete appele avec $# parametres
echo qui sont : $*
L'exécution donne :
xstra> **echopara2 a b c d**
./echopara2 a ete appele avec 4 parametres

qui sont : a b c d
xstra>
Attention, la variable # n'est pas une variable numérique (qui n'existe pas) mais une variable de type chaîne de caractères (de même pour *$*).
0 permet de savoir sous quel nom ce script a été invoqué. Dans le cas où le script porte plusieurs noms (par des liens), cela permet de prendre telle ou telle décision suivant le nom sous lequel le script a été invoqué.
$ numéro de process (PID) du shell, est unique dans le système : il est fréquemment utilisé pour générer des noms de fichiers temporaires.
Selon un mécanisme général (voir le paragraphe 3.2 et 7.7), le shell peut générer
une liste des noms de fichiers correspondant à un certain modèle (grâce aux caractères * et ?). Cette génération a lieu **avant** l'invocation de la commande (et donc du script) concernée. Par exemple, si le répertoire courant contient uniquement les fichiers *fich1* et *fich2*, lors de la commande *ls fi**, le shell génère la liste *fich1 fich2* et la passe en argument à la commande *ls*. La commande effectivement lancée par le shell est donc : *ls fich1 fich2*.

8.1.5 Les instructions de lecture et d'écriture

Ces instructions permettent de créer des fichiers de commandes interactifs par l'instauration d'un dialogue sous forme de questions/réponses. La question est posée par l'ordre *echo* et la réponse est obtenue par l'ordre *read* à partir du clavier.
read variable1 variable2… variablen
read lit une ligne de texte à partir du clavier, découpe la ligne en mots et attribue aux variables *variable1* à *variablen* ces différents mots. S'il y a plus de mots que de variables, la dernière variable se verra affecter le reste de la ligne.
Exemple
Le script *affiche* contient :
echo n "Nom du fichier a afficher : "
read fichier
more $fichier

8.1.6 Les structures de contrôle

Le shell possède des structures de contrôle telles qu'il en existe dans les langages de programmation d'usage général :
● instructions conditionnelles (if.. then.. else, test, case),
● itérations bornées,
● itérations non bornées.
a) Les instructions conditionnelles
Pour la programmation des actions conditionnelles, nous disposons de trois outils :
● l'instruction if,
● la commande test qui la complète,
● l'instruction case.
➤ L'instruction if
Elle présente trois variantes qui correspondent aux structures sélectives à une, deux ou n alternatives.

α) La sélection à une alternative : if… then… fi
if commande
then commandes
fi
Les *commandes* sont exécutées si la commande condition *commande* renvoie un code retour nul ($? = 0).
Exemple

Le script *echoif1* contient :
if grep i xstra1 /etc/passwd
then echo L\'utilisateur xstra1 est connu du systeme
fi
β) La sélection à deux alternatives : if… then… else… fi
if commande
then commandes1
else commandes2
fi
Les commandes *commandes1* sont exécutées si la commande_condition *commande* renvoie un code retour nul, sinon ce sont les *commandes2* qui sont exécutées.
Exemple
Le script *echoif2* contient :
if grep qi "xstra1" /etc/passwd
then echo L\'utilisateur xstra1 est connu du systeme
else echo L\'utilisateur xstra1 est inconnu du systeme
fi
γ) La sélection à n alternatives : if… then…. elif… then… fi
if commande1
then commandes1
elif commande2
then commandes2
elif commande3
then commandes3
…
else
commandes0
fi

Exemple
#!/bin/bash
ce script a pour nom scriptf
ce script montre comment utiliser
les parametres remplacables
lors de l'invocation d'un script.
exemples: xstra>scriptf
xstra>scriptf 1
xstra>scriptf 1 2 3 4
xstra>scriptf "1 2 3 4"
xstra>scriptf ok un deux trois
#
echo "" #echo d'une ligne vide saut de ligne
echo ""
echo "Exemple de passage de parametres a un script."
echo ""
la ligne suivante montre comment utiliser
le resultat d'une commande:
echo "Le repertoire courant est: `pwd`"
la ligne suivante montre la difference entre
simples et doubles apostrophes:

```
echo 'Le repertoire courant est: `pwd`'
echo "Ce script a pour nom: $0"
if [ $# eq 0 ] # comment tester le nombre de parametres?
then
echo "Il a ete appele sans parametre."
else
if [ $# eq 1 ]
then
echo "Il a ete appele avec le parametre $1"
else
echo "Il a ete appele avec $# parametres, qui sont: $*"
fi
if [ $1 'ok' ] # comment tester le contenu d'un parametre
then
echo "Bravo: le premier parametre vaut $1"
else
echo "Helas, le premier parametre ne vaut pas ok mais $1"
fi
fi
echo "Au revoir $LOGNAME, le script $0 est fini."
```

➤ La commande test

Elle constitue l'indispensable complément de l'instruction *if*. Elle permet très simplement :

- de reconnaître les caractéristiques des fichiers et des répertoires,
- de comparer des chaînes de caractères,
- de comparer algébriquement des nombres.

Cette commande existe sous deux syntaxes différentes :

test expression

ou

[expression]

La commande *test* répond à l'interrogation formulée dans *expression*, par un code de retour nul en cas de réponse positive et différent de zéro sinon. La deuxième forme est plus fréquemment rencontrée et donne lieu à des programmes du type :

if [expression]
then commandes
fi

Attention

Dans *[expression]*, ne pas oublier le caractère espace entre *[* et *expres sion* et entre *expression* et *]*. Si *then* est sur la même ligne, il doit être séparé du *]* par un espace et un caractère ; les expressions les plus utilisées sont :

-d nom vrai si le répertoire nom existe,
-f nom vrai si le fichier nom existe,
-s nom vrai si le fichier nom existe et est non vide,
-r nom vrai si le fichier nom existe et est accessible en lecture,
-w nom vrai si le fichier nom existe et est accessible en écriture,
-x nom vrai si le fichier nom existe et est exécutable,
-z chaîne vrai si la chaîne de caractères chaîne est vide,
-n chaîne vrai si la chaîne de caractères chaîne est non vide,
c1 = c2 vrai si les chaînes de caractères c1 et c2 sont identiques,

c1 != c2 vrai si les chaînes de caractères c1 et c2 sont différentes,
n1 -eq n2 vrai si les entiers n1 et n2 sont égaux.
(Les autres opérateurs relationnels sont *ne, lt, le, -gt* et *-ge*.)
Remarque
Les expressions peuvent être niées par l'opérateur logique de **négation** *!* et
combinées par les opérateurs **ou logique** *o* et **et logique** *a*.

➢ L'instruction case

L'instruction *case* est une instruction très puissante et très commode pour effectuer un choix multiple dans un fichier de commandes.
case chaine in
motif1) commandes 1 ;;
motif2) commandes 2 ;;

....
....
motifn) commandes n ;;
esac
Le shell recherche, parmi les différentes chaînes de caractères *motif1*,
motif2,…, *motifn* proposées, la première qui correspond à *chaine* et il exécute les commandes correspondantes. Un double point-virgule (*;;*) termine chaque choix. La *chaine* dans un *case* peut prendre diverses formes :
● un chiffre,
● une lettre ou un mot,
● des caractères spéciaux du shell,
● une combinaisons de ces éléments.
La *chaine* peut être lue, passée en paramètre ou être le résultat d'une commande exécutée avec l'opérateur backquote ` ` ou $( ). Dans les différentes
chaînes *motif1* à *n*, on peut utiliser les caractères spéciaux (*****, **?**,…). De plus, pour regrouper plusieurs motifs dans une même alternative, on utilise le caractère **|** (obtenu sur un clavier standard par la combinaison <Alt Gr><6>).
Exemple 1
Le script *comptepara* contient :

```
case $# in
0) echo $0 sans argument ;;
1) echo $0 possede un argument ;;
2) echo $0 a deux arguments ;;
*) echo $0 a plus de deux arguments ;;
esac
```

b) Les itérations
La présence des instructions itératives dans le shell en fait un langage de programmation complet et puissant. Le shell dispose de trois structures itératives : *for*, *while* et *until*.

➢ Itération bornée : La boucle for

Trois formes de syntaxe sont possibles :
1) Forme 1
for variable in chaine1 chaine2… chainen
do
commandes

done
2) Forme 2

for variable
do
commandes
done
3) Forme 3
*for variable in **
do
commandes
done
Pour chacune des trois formes, les commandes placées entre *do* et *done* sont exécutées pour chaque valeur prise par la variable du shell *variable*. Ce qui change c'est l'endroit où *variable* prend ses valeurs. Pour la forme 1, les valeurs de *variable* sont les chaînes de *chaine1* à *chainen*. Pour la forme 2, *variable* prend ses valeurs dans la liste des paramètres du script. Pour la forme 3, la liste des fichiers du répertoire constitue les valeurs prises par variable.

Exemple 1

Le script *echofor1* contient :

```
for i in un deux trois
do
echo $i
done
```

L'exécution donne :

```
xstra> echofor1
un
deux
trois
xstra>
```

Exemple 2

Le script *echofor2* contient :

```
for i
do
echo $i
done
```

L'exécution donne :

```
xstra> echofor2 le système Linux
le
système

Linux
xstra>
```

Exemple 3

Le script *echofor3* contient :

```
for i in *
do
echo $i
done
```

L'exécution donne :

```
xstra> echofor3
fich1
fich2
fich3
```

xstra>
Exemple 4
Le script *lsd* contient :
```
echo ”liste des repertoires sous `pwd`”
echo ” ”
for i in *
do
if [ d= $i ]
then
echo $i ” :repertoire”
fi
done
echo ” ”
```
L’exécution donne :
xstra> **lsd**
liste des repertoires sous /home/xstra/test
filon : repertoire
xstra>
➤ Itérations non bornées : while et until
while commandealpha
do commandesbeta
done
until commandealpha
do commandesbeta
done

Les commandes *commandesbeta* sont exécutées tant que (*while*) ou jusqu’à ce que (*until*) la commande *commandealpha* retourne un code nul (la condition est vraie).
Exemple 1
Le script suivant liste les paramètres qui lui sont passés en argument jusqu’à ce qu’il rencontre le paramètre *fin*.
```
#!/bin/bash
# ce script a pour nom while_
# ce script montre le fonctionnement
# de la construction while
# exemple: xstra>while_ 1 2 3 fin 4 5 6
#
while [ $1 != fin ];do
echo $1
shift
done
```
L’exécution donne :
xstra> **while_ 1 2 3 4 fin 5 6 7**
1
2
3
4
xstra>
Exemple 2
```
#!/bin/bash
```

```
# ce script a pour nom until_
# ce script montre l'utilisation
# de la construction until
# exemple: xstra>until_ 1 2 3 fin 4 5 6
#
until [ $1= fin ] ;do
echo $1
shift
done
```

L’exécution donne :

```
xstra> until_ 1 2 3 fin 4 5 6
1
2
3
xstra>
```

Exemple 3

```
#!/bin/bash
# ce script a pour nom while_until
# il illustre l'usage combine
# des constructions while et until
# exemple: xstra>while_until 1 2 3 debut 4 5 6 fin 7 8 9
#
while [ $1 != debut ] ;do
shift
done
shift
until [ $1= fin ] ;do
echo $1
shift
done
```

L’exécution donne :

```
xstra> while_until 1 2 3 debut 4 5 6 fin 7 8 9
456
xstra>
```