Scripts Shell

Commandes Shell

Syntaxe Générale

Syntaxe:

commande [± option…] [paramètre…]

Une commande se compose d'un code mnémonique en minuscules suivi ou non d'arguments séparés par au moins un espace.

- les options, presque toujours précédées d'un signe (-) et quelquefois d'un signe (+). Certaines options ont un paramètre propre,
- les arguments positionnels ou paramètres, qui sont en général des noms de fichiers ou de répertoires.

En général, la liste des options précède celle des paramètres.

La ligne de commandes séquentielles

Il est possible de taper plusieurs commandes sur la même ligne en les séparant par des points-virgules g

Les commandes sont exécutées séquentiellement, de façon totalement indépendante, la première n'influençant pas la seconde et ainsi de suite.

Exemple:

pwd ; who ; ls

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

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

```
ls -l /home/user1/develop\
 /essai.f
```

Cette commande est équivalente à :

```
ls -l /home/user1/develop/essai.f
```

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).

Les séparateurs conditionnels de commandes

Exemple 1

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

cd projet1 && rm *

Exemple 2:

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

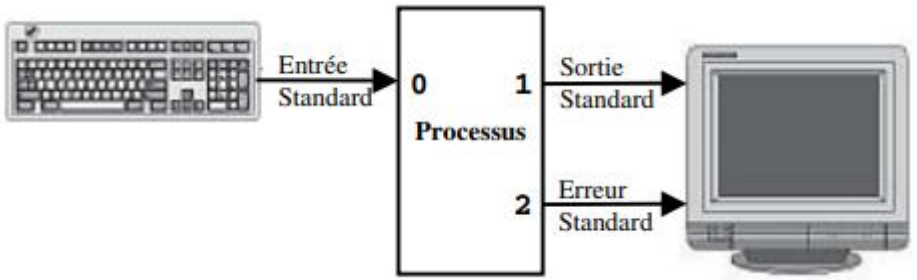
cd projet1 || mkdir projet1

Le redirection des entrées-sorties

Le principe de redirection

On appelle processus, ou tâche, l'exécution d'un programme exécutable.

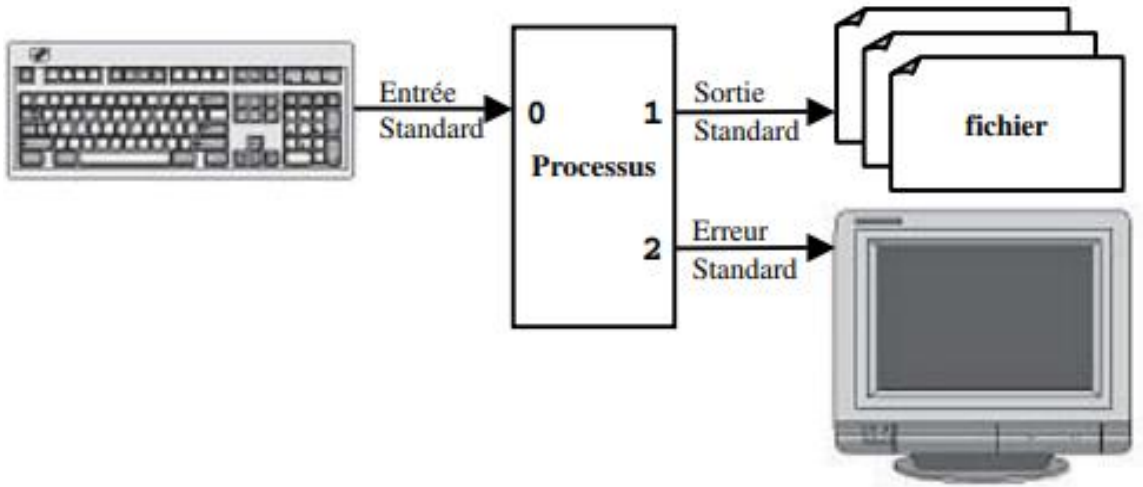
Au lancement de chaque processus, l'interpréteur de commandes ouvre d'office une entrée standard (par défaut le clavier), une sortie standard (par défaut l'écran) et la sortie d'erreur standard (par défaut l'écran)



Le redirection des entrées-sorties

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.

Aucune information n'apparaîtra à l'écran, hormis celles qui transitent par la sortie d'erreur standard.



Le redirection des entrées-sorties

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

Le redirection des entrées-sorties

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

Dans l'exemple suivant on définit le fichier **fentree** comme entrée standard:

<fentree

Le caractère **>** suivi du nom d'un fichier indique la redirection de la sortie standard vers un fichier. Dans l'exemple suivant on a la redirection de la sortie standard vers le fichier **fsortie**:

>fsortie

Si on utilise **>>** la redirection sera ajoutée au fichier **fsortie**

>>fsortie

Le redirection des entrées-sorties

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

Exemple:

2>erfs

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

commande >fs 2>erfs

Le redirection des entrées-sorties

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.

Exemple redirection de la sortie standard et de la sortie d'erreur vers le fichier erfs

&>erfs

La commande cat et les redirections

La commande cat est une commande multi-usage qui permet d'afficher, de créer, de copier et de concaténer des fichiers.

Elle utilise pleinement les mécanismes de redirection.

Elle lit l'entrée standard si aucun fichier n'est spécifié. Ce qui est lu est affiché sur la sortie standard.

Lecture sur clavier et écriture sur écran:

cat

Copie d'un fichier:

cat f1>f2

La commande cat et les redirections

Concaténation des fichiers:

cat f1 f2 f3 >f123

Le fichier f123 contiendra la concaténation des fichiers f1, f2 et f3 dans cet ordre.

Ajout d'un fichier:

cat f1 >>f2

Le fichier f1 est concaténé à la suite du fichier f2. f2 est créé s'il n'existe pas.

Création d'un fichier par saisie au clavier

cat >f1

La commande cat et les redirections

Création d'un fichier avec condition de saisie

cat <<EOT>f1

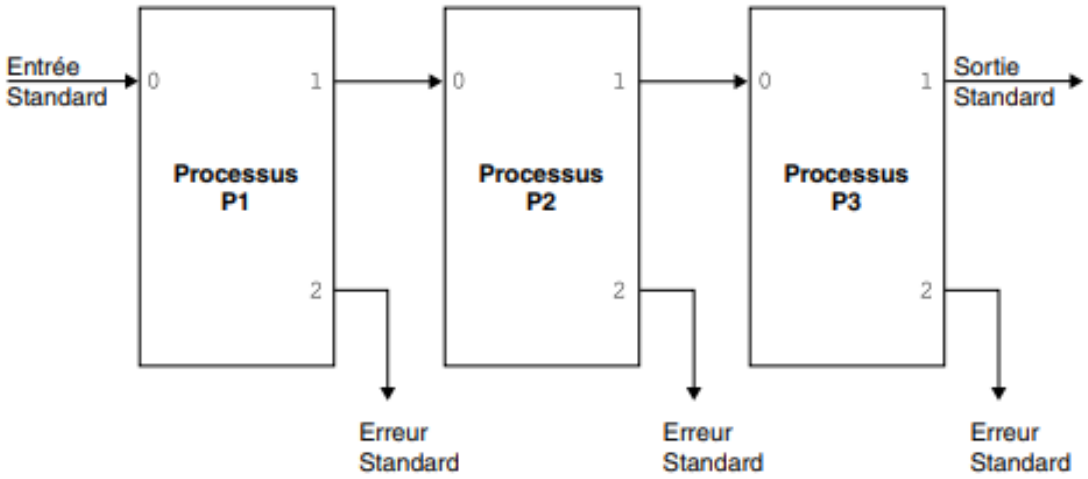
Le fichier f1 est créé par saisie de texte au clavier, jusqu'à la saisie en début de ligne d'une chaîne de caractères prédéfinie (EOT dans ce cas). La chaîne de caractères **< <texte** redirige l'entrée standard jusqu'à apparition du mot **texte.**

Cette formulation est très employée pour la création d'un fichier dans un fichier de commandes (script)**.**

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

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



LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

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

P1 | P2 | P3

Exemple:

ls -l | grep ”rwxr xr x”| less

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

On peut obtenir le même résultat avec la commande suivante:

ls -l >temp1; grep ”rwxr xr x” <temp1 >temp2 ; less temp2; rm temp1 temp2

Cette solution est non seulement plus lourde mais aussi plus lente.

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

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

who | wc -l

ls | wc -w

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

Les filtres

Dans les exemples précédents, nous voyons apparaître, à travers les commandes **wc, less et grep**, une famille particulièrement importante de commandes Linux : **les filtres**.

Un filtre est une commande qui lit les données sur l'entrée standard, les traite et les écrit sur la sortie standard.

Le concept de tube, avec sa simplicité, devient un outil très puissant dans Linux qui propose un choix très vaste de filtres

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

Les filtres les plus utilisés sont les suivants:

grep recherche les occurrences d'une chaîne de caractères.

egrep une extension de **grep**

wc compte le nombre de caractères (ou octets), mots et lignes

less affiche son entrée standard page par page

dd filtre de conversion

sed éditeur de flot : il applique des commandes de l'éditeur **ed** sur l'entrée standard et envoie le résultat sur la sortie standard

awk petit langage de manipulation de texte

sort filtre de tri

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

La commande xargs

Le mécanisme de tube est très pratique pour assembler entre elles des commandes Linux. Cependant, beaucoup de commandes ne lisent pas dans leur entrée standard : ls, rm, cp, ln, mv et bien d'autres ne sont pas des filtres, mais traitent leurs arguments.

Ces commandes ne peuvent donc pas apparaître dans un tube, sauf au début.

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

Ces commandes ne peuvent donc pas apparaître dans un tube, sauf au début. La commande **xargs** permet de lever cette restriction en construisant une liste de paramètres à partir de l'entrée standard.

cmd1 | xargs cmd2 signifie : lancer cmd2 en lui passant en paramètres ce qui arrive dans l'entrée standard de **xargs**, donc la sortie standard de cmd1.

LES TUBES DE COMMUNICATION (PIPE) ET LES FILTRES

Exemple d'utilisation de la commande xargs :

Le fichier **listf** contient une liste de noms de fichiers, à raison d'un nom par ligne.

Si on veut afficher les propriétés des fichiers correspondants on peut le faire avec la commande suivante:

cat listf | xargs ls –i

TÂCHES EN ARRIÈRE-PLAN

Linux attend la fin de l'exécution de la commande en cours d'exécution avant de permettre à l'utilisateur de relancer une nouvelle commande.

Lorsqu'une commande ne nécessite pas de dialogue avec l'utilisateur et que sa durée d'exécution est importante, il est possible de l'exécuter en arrière-plan en ajoutant le caractère spécial **&** à la fin de la commande.

Le système Linux lance alors la commande et redonne immédiatement la main à l'utilisateur pour d'autres travaux.

TÂCHES EN ARRIÈRE-PLAN

Le lancement de commandes en arrière-plan permet à un seul utilisateur de lancer plusieurs tâches à partir d'un même terminal.

Cela correspond à l'aspect multitâche du système d'exploitation Linux.

Ces processus en arrière-plan ne peuvent plus être interrompus directement à partir de la console à l'aide de la touche <Ctrl C> . Pour les interrompre il faut :

- soit sortir de sa session, dans ce cas tous les processus attachés à la session seront interrompus,
- soit rechercher le numéro du processus et lui envoyer un signal à l'aide de la commande kill,

TÂCHES EN ARRIÈRE-PLAN

Il est possible d'éviter qu'un processus en arrière-plan soit interrompu en quittant sa session.

Il faut pour cela utiliser la commande **nohup** :

nohup commande <entree >sortie &

Ce mécanisme est intéressant si l'on souhaite exécuter un programme de très longue durée et libérer la console de lancement.

TÂCHES EN ARRIÈRE-PLAN

La commande **fg** permet de passer en avant plan un job qui s'exécute en arrière plan, et s'il s'agit d'un processus arrêté il se relancera en avant plan

fg numéro_job

La commande **bg** permet de lancer en arrière plan un job arrêté

bg numéro_job

TÂCHES EN ARRIÈRE-PLAN

Lors du lancement d'une commande en arrière-plan, l'interpréteur de commandes Bash attribue un numéro croissant pour chacune des commandes lancées : le numéro job.

La liste des travaux en arrière-plan, avec leur numéro, est obtenue à l'aide de la commande jobs. Cette commande indique si le processus en arrière plan est en cours d'exécution (running) ou suspendu (stopped)

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 ».

Soit (`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

LA SUBSTITUTION DE COMMANDE

Exemple

La commande echo affiche à l'écran la chaîne de caractères qui la suit. Testez les résultats des commandes suivantes:

echo pwd

echo 'pwd'

echo `pwd`

echo $(pwd)

LA SUBSTITUTION DE COMMANDE

grep -n ”motif” $(find . type f –print)

Cette commande permet de rechercher à partir du répertoire courant et récursivement dans tous les sous-répertoires, les fichiers contenant la chaîne de caractères

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.

LES COMMANDES GROUPÉES

Exemple: Suppression du fichier temporaire dans le répertoire projet1.

(cd projet1; rm temporaire)

Une autre particularité de la commande groupée est de permettre de lancer tout le groupe de commandes en arrière-plan. En effet, le caractère **&** à la fin d'une ligne séquentielle ne lance en arrière-plan que la dernière commande.

LES CARACTÈRES SPÉCIAUX GÉNÉRATEURS DE NOMS DE FICHIER

Les caractères génériques permettent de désigner un ensemble d'objets. Ils peuvent être utilisés pour la génération d'un ensemble de noms de fichiers.

Ils s'appliquent donc aux paramètres des commandes qui désignent des noms de fichiers. Ils peuvent aussi désigner un ensemble de chaînes de caractères.

On parle alors d'expressions régulières. Ces expressions s'appliquent aux commandes d'édition (ex, vi, sed, …) ou à des filtres (grep, egrep, awk, …) et permettent la recherche d'une chaîne de caractères dans un fichier. Elles s'appliquent donc aux contenus des fichiers.

LES CARACTÈRES SPÉCIAUX GÉNÉRATEURS DE NOMS DE FICHIER

Il y a des différences d'interprétation entre les caractères spéciaux (ou métacaractères) utilisés dans la génération des noms de fichier (présentation ci-dessous) et ceux utilisés dans les expressions régulières.

L'interpréteur de commandes permet de générer une liste de noms de fichier en utilisant les caractères spéciaux suivants

LES CARACTÈRES SPÉCIAUX GÉNÉRATEURS DE NOMS DE FICHIER

Caractères Spéciaux	**Interprétation**
*	désigne toutes chaînes de caractères, y compris la chaîne vide
?	désigne un caractère quelconque.
[…]	désigne un caractère quelconque appartenant à la liste donnée entre crochets. Deux caractères séparés par un tiret ( ) définissent une liste de caractères rangés par ordre alphabétique, dont le premier élément est le premier caractère et le dernier élément le dernier caractère.
[!...]	

LES CARACTÈRES DE NEUTRALISATION

Nous avons vu que certains caractères spéciaux ont une signification particulière pour l'interpréteur de commandes, comme par exemple :

,*,?,!,... Il est possible, en plaçant le caractère \ (backslash) devant un tel caractère spécial, de neutraliser son interprétation par l'interpréteur de commandes.

La programmation en shell

Script Shell

Le shell est plus qu'un interpréteur de commandes : c'est également un puissant langage de programmation.

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.

Script Shell

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

Script Shell

Le jeu d’instructions lui-même comporte :

- toutes les commandes Linux,
- l’invocation 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.

Script Shell

- Le shell est un langage interprété ; en conséquence tout changement dans le système sera pris en compte par un script lors de sa prochaine utilisation (il est inutile de “recompiler ” les scripts).
- Il est tout à fait possible d’écrire et d’invoquer des scripts dans un certain shell tout en utilisant un autre shell en interactif. En particulier, il est très fréquent (mais non obligatoire) d’utiliser un TC-shell en tant que “login shell” et le Bourne-shell pour l’écriture des scripts. Les scripts les plus simples (listes de commandes) seront identiques quel que soit le shell, mais dès que des instructions de tests ou d’itérations sont nécessaires, les syntaxes du Bourne-shell, du Bash et du C-shell diffèrent.

Script Shell

- Si un script commence par la ligne

#!/bin/xxx

/bin/xxx représente le chemin d'accès du shell xxx qui doit interpréter ce script

il est interprété par le shell **/bin/xxx.**

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

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:

chmod a+x listf

Il peut donc être exécuté comme une commande :

./listf

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.

Le passage des paramètres

Premier script avec passage de paramètres

Avec vi, modifier le fichier listf de la façon suivante :

echo ”contenu du repertoire $1 ”

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.

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).

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.

Les instructions de lecture et d’écriture

Exemple

echo n ”Nom du fichier a afficher : ”

read fichier

more $fichier

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.

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.

- **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).

L’instruction if

Exemple echoif1

```
if grep   user1 /etc/passwd
then echo L\’utilisateur user1 est connu du systeme
fi
```

L’instruction if

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.

L’instruction if

Exemple

Le script echoif2 contient :

```
if grep   user1 /etc/passwd
then echo L\’utilisateur user1 est connu du systeme
else echo L\’utilisateur user1 est inconnu du systeme
fi
```

L’instruction if

La sélection à n alternatives : if… then…. elif… then… f

```
if commande1
then commandes1
elif commande2
then commandes2
elif commande3
then commandes3
…
else commandes0
fi
```

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

La commande test

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

La commande test

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,

La commande test

-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 expressions peuvent être niées avec **!**

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 ;;
…
motif3) commndesn;;
esac
```

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 :

Forme 1

for variable in chaine1 chaine2… chainen

do commandes

done

Itération bornée : La boucle for

Forme 2

for variable

do

commandes

done

Itération bornée : La boucle for

Forme 3

for variable in *

do

Commandes

done

Itération bornée : La boucle for

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.

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).