Scripts shell - Instructions de contrôle

1. Tests

Lorsqu'elles se terminent, toutes les commandes exécutées par le shell renvoient un code de retour (également appelé code de statut ou de sortie).

La convention établie veut que si la commande s'est bien exécutée, le code de statut ait pour valeur zéro.

Si la commande a eu un problème lors de son exécution (les raisons peuvent être nombreuses : manque de droits d'accès, absence de fichier, saisie incorrecte, …), son code de status aura une valeur différente de zéro.

Il faut se référer au manuel de la commande pour connaître les différentes valeurs du code de retour prévues par les développeurs.

Le code de retour n'est pas visible directement, mais est enregistré dans une variable spéciale : **$?**.

```
[stagiaire] $ mkdir repertoire
[stagiaire] $ echo $?
0
[stagiaire] $ mkdir /repertoire
mkdir: impossible de créer le répertoire
[stagiaire] $ echo $?
1
```

L'affichage du contenu de la variable **$?** avec la commande **echo** se fait immédiatement après la commande que l'on souhaite évaluer. Cette variable étant mise à jour après chaque commande.

Il est également possible de créer des codes de retour dans un script. Il suffit pour cela d'ajouter un argument numérique à la commande **exit**.

```
[stagiaire] $ exit 2
```

```
[stagiaire] $ echo $?
2
```

Outre la bonne exécution d'une commande, le shell offre la possibilité d'exécuter des tests sur de nombreux motifs :

- Fichiers (existance, type, droits, comparaison) ;
- Chaînes de caractères (longueur, comparaison) ;
- Numériques entiers (valeur, comparaison).

Le résultat du test :

- $?=0 : le test s'est correctement exécuté et est vrai ;
- $?=1 : le test s'est correctement exécuté et est faux ;
- $?=2 : le test ne s'est pas correctement exécuté.

1.1. Tester le type d'un fichier

Syntaxe de la commande test pour un fichier.

```
test [-d|-e|-f|-L] fichier
```

Table 1. Options de la commande test sur les fichiers

Option	Observation
-e	Teste si le fichier existe
-f	Teste si le fichier existe et est de type normal
-d	Teste si le fichier existe et est de type répertoire
-L	Teste si le fichier existe et est de type lien symbolique
-b	Teste si le fichier existe et est de type spécial mode bloc
-c	Teste si le fichier existe et est de type spécial mode caractère
-p	Teste si le fichier existe et est de type tube
-S	Teste si le fichier existe et est de type socket
-t	Teste si le fichier existe et est de type terminal
-r	Teste si le fichier existe et est accessible en lecture

Option	Observation
-w	Teste si le fichier existe et est accessible en écriture
-x	Teste si le fichier existe et est est exécutable
-g	Teste si le fichier existe et est a un SGID positionné
-u	Teste si le fichier existe et est a un SUID positionné
-s	Teste si le fichier existe et est non vide (taille > 0 octets)

1.2. Comparer deux fichiers

La commande **test** peut également comparer des fichiers :

Syntaxe de la commande test pour la comparaison de fichiers.

```
test fichier1 [-nt|-ot|-ef] fichier2
```

Table 2. Options de la commande test pour la comparaison de fichiers

Option	Observation
-nt	Teste si le premier fichier est plus récent qu ele second
-ot	Teste si le premier fichier est plus ancien que le second
-ef	Teste si le premier fichier est un lien physique du second

1.3. Tester une variable

Syntaxe de la commande test pour les variables.

```
test [-z|-n] $variable
```

Table 3. Options de la commande test pour les variables

Option	Observation
-z	Teste si la variable est vide
-n	Teste si la variable n’est pas vide

1.4. Tester une chaîne de caractères

Syntaxe de la commande test pour les chaînes de caractères.

```
test chaîne1 [=|!=] chaîne2
```

Exemple :

```
[stagiaire] $ test "$var" = "Hello world !"
[stagiaire] $ echo $?
0
```

Table 4. Options de la commande test pour les variables

Option	Observation
=	Teste si la première chaine de caractères est égale à la seconde
!=	Teste si la première chaine de caractères est différente de la seconde
<	Teste si la première chaine de caractères est avant la seconde dans l'ordre ASCII
>	Teste si la première chaine de caractères est après la seconde dans l'ordre ASCII

1.5. Comparaison de numériques entiers

Syntaxe de la commande test pour les entiers.

```
test "num1" [-eq|-ne|-gt|-lt] "num2"
```

Exemple :

```
[stagiaire] $ test "$var" -eq "1"
[stagiaire] $ echo $?
0
```

Table 5. Options de la commande test pour les entiers

Option	Observation
-eq	Teste si le premier nombre est égal au second
-ne	Teste si le premier nombre est différent au second
-gt	Teste si le premier nombre est supérieur au second

Option	Observation
-lt	Teste si le premier nombre est inférieur au second

Les numériques étant traités par le shell comme des caractères (ou chaînes de caractères) classiques, un test sur un caractère peut renvoyer le même résultat qu'il soit traité en tant que numérique ou non.

```
[stagiaire] $ test "1" = "1"
[stagiaire] $ echo $?
0
[stagiaire] $ test "1" -eq "1"
[stagiaire] $ echo $?
0
```

Mais le résultat du test n'aura pas la même signification :

- Dans le premier cas, il signifiera que les deux caractères ont la même valeur dans la table ASCII.
- Dans le second cas, il signifiera que les deux nombres sont égaux.

1.6. Combinaison de tests

La combinaison de test permet d'effectuer plusieurs tests en une seule commande. Il est possible de tester plusieurs fois le même argument (fichier, chaîne ou numérique) ou des arguments différents.

```
test option1 argument1 [-a|-o] option2 argument 2
```

```
[stagiaire] $ test -d /etc -a -x /etc
[stagiaire] $ echo $?
0
```

Table 6. Options de combinaison de tests

Option	Observation
-a	ET : Le test sera vrai si tous les motifs le sont.

Option	Observation
-o	OU : Le test sera vrai si au moins un motif l'est.

Les tests peuvent ainsi être groupé avec des parenthèses **( )** pour leur donner une priorité.

```
(TEST1 -a TEST2) -a TEST3
```

Le caractère **!** permet d'effectuer le test inverse de celui demandé par l'option :

```
[stagiaire] $ test -e /fichier # test si fichier existe
[stagiaire] $ ! test -e /fichier # test si fichier n'existe pas
```

1.7. Les opérations numériques

La commande **expr** effectue une opération avec des entiers numériques.

```
expr num1 [+] [-] [\*] [/] [%] num2
```

Exemple :

```
[stagiaire] $ expr 2 +2
4
```

Dans le cas d'une multiplication, le caractère joker * est précédé par \ pour éviter une mauvaise interprétation.

Opérateur	Observation
+	Addition
-	Soustraction
*	Multiplication
/	Quotient de la division
%	Modulo de la division

1.8. La commande typeset

La commande **typeset -i** déclare une variable comme un entier.

Exemple :

```
[stagiaire] $ typeset -i var1
[stagiaire] $ var1=1+1
[stagiaire] $ var2=1+1
[stagiaire] $ echo $var1
2
[stagiaire] $ echo $var2
1+1
```

1.9. La commande let

La commande **let** teste si un caractère est numérique.

Exemple :

```
[stagiaire] $ var1="10"
[stagiaire] $ var2="AA"
[stagiaire] $ let $var1
[stagiaire] $ echo $?
0
[stagiaire] $ let $var2
[stagiaire] $ echo $?
1
```

La commande **let** ne retourne pas un code retour cohérent lorsqu'elle évalue le numérique 0

```
[stagiaire] $ let 0
[stagiaire] $ echo $?
1
```

La commande **let** permet également d'effectuer des opérations mathématiques :

```
[stagiaire] $ let var=5+5
[stagiaire] $ echo $var
```

```
10
```

2. Structures conditionnelles

Si la variable $? permet de connaître le résultat d'un test ou de l'exécution d'une commande elle ne peut qu'être affichée et n'a aucune incidence sur le déroulement d'un script.

Mais nous pouvons nous en servir dans une condition. **Si** le test est bon **alors** je fais cette action **sinon** je fais telle autre action.

Syntaxe de l'alternative conditionnelle if.

```
if commande
then
    commande si $?=0
else
    commande si $?!=0
fi
```

La commande placée après le mot **if** peut être n'importe quelle commande puisque c'est son code de retour, **$?**, qui sera évalué. Il est souvent pratique d'utiliser la commande **test** pour définir plusieurs actions en fonction du résultat de ce test (fichier existe, variable non vide, droits en écriture positionnés.). Utiliser une commande classique (mkdir, tar, …) permet de définir les actions à effectuer en cas de succès ou les messages d'erreur à afficher en cas d'échec.

```
if test -e /etc/passwd
then
    echo "Le fichier existe"
else
    echo "Le fichier n'existe pas"
fi

if mkdir rep
then
    cd rep
fi
```

Si le bloc **else** commence par une nouvelle structure **if**, il est possible de fusionner **else** et **if** :

```
[...]
else
  if test -e /etc/
[...]

[...]
# est équivalent à
elif test -e /etc
[...]
```

La structure **if** / **then** / **else** / **fi** évalue la commande placée après **if** :

- Si le code retour de cette commande est 0 (vrai) le shell exécutera les commandes placées après **then** ;
- Si le code retour est différent de 0 (faux) le shell exécutera les commandes placées après **else**.

Le bloc **else** est facultatif.

Il existe un besoin d'effectuer certaines actions uniquement si l'évaluation de la commande est vraie, et n'avoir rien à faire si elle est fausse.

Le mot **fi** ferme la structure.

Lorsqu'il n'y a qu'une seule commande à exécuter dans le bloc **then**, il est possible d'utiliser une syntaxe plus simple.

La commande à exécuter si **$?** est vrai est placée après **&&** tandis que la commande à exécuter si **$?** est faux est placée après **||** (*facultatif*).

Par exemple :

```
[stagiaire]$ test -e /etc/passwd && echo "Le fichier existe" || echo "Le
 fichier n'existe pas"
[stagiaire]$ mkdir repert && echo "Le repertoire est créé"
```

Il est possible d'évaluer et de remplacer une variable avec une structure plus légère que **if**.

Cette syntaxe met en oeuvre les accolades :

- Affiche une valeur de remplacement si la variable est vide :

```
${variable:-valeur}
```

- Affiche une valeur de remplacement si la variable n'est pas vide :

```
${variable:+valeur}
```

- Affecte une nouvelle valeur à la variable si elle est vide :

```
${variable:=valeur}
```

Exemples :

```
[stagiaire]$ nom=""
[stagiaire]$ echo ${nom:-linux}
linux
[stagiaire]$ echo $nom

[stagiaire]$ echo ${nom:=linux}
linux
[stagiaire]$ echo $nom
linux
[stagiaire]$ echo ${nom:+tux}
tux
[stagiaire]$ echo $nom
linux
```

2.1. Structure alternative conditionnelle case

Une succession de structures **if** peut vite devenir lourde et complexe. Lorsquelle concerne l'évaluation d'une même variable, il est possible d'utiliser une structure conditionnelle à plusieurs branches. Les valeurs de la variable peuvent être précisées ou appartenir à une liste de possibilités.

Les caractères jokers sont utilisables.

La structure **case / esac** évalue la variable placée après **case** et la compare aux valeurs définies. À la première égalité trouvée, les commandes placées entre **)** et **;;** sont exécutées.

La variable évaluée et les valeurs proposées peuvent être des chaînes de caractères ou des résultats de sous-exécutions de commandes.

Placé en fin de structure, le choix * indique les actions à exécuter pour toutes les valeurs qui n'ont pas été précédemment testées.

Syntaxe de l'alternative conditionnelle case.

```
case $ in
  valeur1)
    commandes si variable = valeur1
    ;;
  valeur2)
    commandes si variable = valeur2
    ;;
  [..]
  *)
    commandes pour toutes les valeurs de variable != de valeur1 et
 valeur2
    ;;
esac
else
    commande si $?!=0
fi
```

Lorsque la valeur est sujette à variation, il est conseillé d'utiliser les caractères jokers [] pour spécifier les possibilités :

```
[Oo][Uu][Ii])
  echo "oui"
  ;;
```

Le caractère | permet aussi de spécifier une valeur ou une autre :

```
"oui" | "OUI")
  echo "oui"
  ;;
```

3. Boucles

Le shell bash permet l'utilisation de boucles. Ces structures permettent l'exécution d'un bloc de commandes plusieurs fois (de 0 à l'infini) selon une valeur définie statiquement, dynamiquement ou sur condition :

- **while**
- **until**
- **for**
- **select**

Quelle que soit la boucle utilisée, les commandes à repéter se placent entre les mots **do** et **done**.

3.1. La structure boucle conditionnelle while

La structure **while** / **do** / **done** évalue la commande placée après **while**.

Si cette commande est vrai ($? = 0), les commandes placées entre **do** et **done** sont exécutées. Le script retourne ensuite au début évaluer de nouveau la commande.

Lorsque la commande évaluée est fausse ($? != 0), le shell reprend l'exécution du script à la première commande après **done**.

Syntaxe de la structure boucle conditionnelle while.

```
while commande
do
  commande si $? = 0
done
```

Exemple :

```
while test -e /etc/passwd
do
  echo "Le fichier existe"
done
```

> Si la commande évaluée ne varie pas, la boucle sera infinie et le shell n'exécutera jamais les commandes placées à la suite dans le script. Cela peut être volontaire, mais aussi être une erreur. Il faut donc faire très attention à la commande qui régit la boucle et trouver un moyen d'en sortir.

Pour sortir d'une boucle while, il faut faire en sorte que la commande évaluée ne soit plus vraie, ce qui n'est pas toujours possible.

Il existe des commandes qui permettent de modifier le comportement d'une boucle :

- **exit**
- **break**
- **continue**

3.2. La commande exit

La commande **exit** termine l'exécution du script.

Syntaxe de la commande exit.

```
exit [n]
```

Exemple :

```
[stagiaire]$ exit 1
[stagiaire]$ echo $?
1
```

La commande exit met fin au script immédiatement. Il est possible de préciser le code de retour du script en le précisant en argument (*de 0 à 255*). Sans argument précisé, c'est le code de retour de la dernière commande du script qui sera transmise à la variable $?.

Cette commande est utile dans le cas d'un menu proposant la sortie du script dans les choix possibles.

3.3. La commande break / continue

La commande **break** permet d'interrompre la boucle en allant à la première commande après **done**.

La commande **continue** permet de relancer la boucle en revenant à la première commande après **do**.

```
while test -d /
do
  echo "Voulez-vous continuer ? (oui/non)"
  read rep
  test $rep = "oui" && continue
  test $rep = "non" && break
done
```

3.4. Les commandes true / false

La commande **true** renvoie toujours vrai tandis que la commande **false** renvoie toujours faux.

```
[stagiaire]$ true
[stagiaire]$ echo $?
0
[stagiaire]$ false
[stagiaire]$ echo $?
1
```

Utilisées comme condition d'une boucle, elles permettent soit d'exécuter une boucle infinie soit de désactiver cette boucle.

Exemple :

```
while true
do
  echo "Voulez-vous continuer ? (oui/non)"
  read rep
  test $rep = "oui" && continue
  test $rep = "non" && break
done
```

3.5. La structure boucle conditionnelle until

La structure **until** / **do** / **done** évalue la commande placée après **until**.

Si cette commande est fausse ($? != 0), les commandes placées entre **do** et **done** sont exécutées. Le script retourne ensuite au début évaluer de nouveau la commande.

Lorsque la commande évaluée est vraie ($? = 0), le shell reprend l'exécution du script à la première commande après **done**.

Syntaxe de la structure boucle conditionnelle while.

```
until commande
do
  commande si $? != 0
done
```

Exemple :

```
until test -e /etc/passwd
do
  echo "Le fichier n'existe pas"
done
```

3.6. La structure choix alternatif select

La structure **select** / **do** / **done** permet d'afficher rapidement un menu avec plusieurs choix et une demande de saisie.

À chaque elément de la liste correspond un choix numéroté. À la saisie, la valeur choisie est affectée à la variable placée après **select** (*créée à cette occasion*).

Elle exécute ensuite les commandes placées entre **do** et **done** avec cette valeur.

- La variable "PS3" va permettre de demander à l'utilisateur de faire un choix;
- La variable "REPLY" va permettre de récupérer le numéro du choix.

Il faut une commande **break** pour sortir de la boucle.

La structure **select** est très utile pour de petits menus simples et rapides. Pour personnaliser un affichage plus complet, il faudra utiliser les commandes **echo** et **read** dans une boucle **while**.

Syntaxe de la structure boucle conditionnelle select.

```
PS3="Votre choix :"
select variable in var1 var2 var3
do
  commandes
done
```

Exemple :

```
PS3="Votre choix : "
select choix in café thé chocolat
do
  echo "Vous avez choisi le $REPLY : $choix"
done
```

ce qui donne à l'exécution :

```
1) Café
2) Thé
3) Chocolat
Votre choix : 2
Vous avez choisi le choix 2 : thé
Votre choix :
```

3.7. La structure boucle sur liste de valeurs for

La structure **for** / **do** / **done** affecte le premier élément de la liste à la variable placée après **for** (*créée à cette occasion*).

Elle exécute ensuite les commandes placées entre **do** et **done** avec cette valeur. Le script retourne ensuite au début affecter l'élément suivant de la liste à la variable de travail.

Lorsque le dernier élément a été utilisé, le shell reprend l'exécution à la première commande après **done**.

Syntaxe de la structure boucle sur liste de valeurs for.

```
for variable in liste
do
  commandes
done
```

Exemple :

```
for fichier in /home /etc/passwd /root/fic.txt
do
  file $fichier
done
```

Toute commande produisant une liste de valeurs peut être placée à la suite du **in** à l'aide d'une sous-exécution. La boucle **for** prendra le résultat de cette commande comme liste d'eléments sur laquelle boucler.

Cela peut être les fichiers d'un répertoire. Dans ce cas, la variable prendra comme valeur chacun des noms des fichiers présents :

```
for fichier in `ls /root`
do
  echo $fichier
done
```

Cela peut être les lignes d'un fichier. Dans ce cas, la variable prendra comme valeur chacune des lignes du fichier parcouru, du début à la fin :

```
for ligne in `more /etc/hosts`
do
  echo $ligne
done
```