Cours UNIX
Programmation Shell
Les scripts

Sommaire

1 La programmation shell

Un script bash est un fichier de type texte contenant une suite de commandes shell, exécutables par l'interpréteur (ici le programme /bin/bash), comme une commande unique.

Un script peut être lancé en ligne de commande, mais aussi dans un autre script.

Un script BASH n'est pas seulement un enchainement de commandes : on y peut définir des variables et utiliser des structures de contrôle, ce qui lui confère le statut de langage de programmation interprété et complet.

Le langage bash gère notamment :

- la gestion des entrées-sorties et de leur redirection
- des variables définies par le programmeur et des variables systèmes
- le passage de paramètres
- des structures conditionnelles et itératives
- des fonctions internes

1.1 Saisie du script

Les lignes commençant par le caractère dièse # sont des commentaires. En insérer abondamment !

Le script doit débuter par l'indication de son interpréteur, à écrire sur la première ligne : **#!/bin/bash**. Si le shell par défaut est bash, cette ligne est superflue

Exemple

```
#!/bin/bash
# script bonjour
# affiche un salut à l'utilisateur qui l'a lancé
# la variable d'environnement $USER contient le nom de login
echo ---- Bonjour $USER -----
# l'option -n empêche le passage à la ligne
# le ; sert de séparateur des commandes sur la ligne
echo -n "Nous sommes le " ; date
# recherche de $USER en début de ligne dans le fichier passwd
# puis extraction de l'uid au 3ème champ, et affichage
echo "Ton numéro d'utilisateur est " $(grep "^$USER" /etc/passwd | cut -d: -f3)
```

1.2 Exécution du script

Il est indispensable que le fichier script ait la permission x (soit exécutable). Lui accorder cette permission pour tous ses utilisateurs avec chmod :

chmod a+x bonjour

Pour lancer l'exécution du script, taper **./bonjour**, le **./** indiquant comme chemin le répertoire courant. Ou bien indiquer le chemin absolu à partir de la racine. Ceci dans le cas où le répertoire contenant le script n'est pas listé dans le PATH

Si les scripts personnels sont systématiquement stockés dans un sous-répertoire précis, par exemple /home/bin, on peut ajouter ce chemin dans le PATH :

PATH=$PATH:$HOME/bin

On peut passer des arguments à la suite du nom du script, séparés par des espaces. Les valeurs de ces paramètres sont récupérables dans le script grâce aux paramètres de position $1, $2 ... Leur nombre est indiqué par $#

Exemple

```
#!/bin/bash
# appel du script : ./bonjour nom prenom
if [ $# = 2 ]
      then
      echo "Bonjour $2 $1 et bonne journée !"
else
      echo "Syntaxe : $0 nom prenom"
fi
```

1.3 Entrées-sorties

Ce sont les voies de communication entre le programme bash et la console :

echo, affiche son argument texte entre guillemets sur la sortie standard, c-à-d l'écran. La validation d'une commande echo provoque un saut de ligne. echo "Bonjour à tous !"

- On peut insérer les caractères spéciaux habituels, qui seront interprétés seulement si l'option **-e** suit echo
  \n (saut ligne), \b retour arrière), \t (tabulation), \a (alarme), \c (fin sans saut de ligne)

```
echo "Bonjour \nà tous !"
echo -e "Bonjour \nà tous !"
echo -e "Bonjour \nà toutes \net à tous ! \c"
```

- read, permet l'affectation directe par lecture de la valeur, saisie sur l'entrée standard au clavier.
  read var1 var2 ... attend la saisie au clavier d'une liste de valeurs pour les affecter, après la validation globale, respectivement aux variables var1, var2 ..

```
echo "Donnez votre prénom et votre nom"
read prenom nom
echo "Bonjour $prenom $nom"
```

2 Les variables du Shell

2.1 Variables programmeur

De façon générale, elles sont de type texte. On distingue les variables définies par le programmeur et les variables systèmes.

syntaxe : **`variable=valeur`**

Attention, le signe = NE DOIT PAS être entouré d'espace(s).

On peut initialiser une variable à une chaine vide :

chaine_vide=

Si valeur est une chaine avec des espaces ou des caractères spéciaux, l'entourer de " " ou de ' '

Le caractère \ permet de masquer le sens d'un caractère spécial comme " ou '

Référence à la valeur d'une variable : faire précéder son nom du symbole $

Pour afficher toutes les variables : set

Pour empêcher la modification d'une variable, invoquer la commande readonly

2.2 Variables exportées

Toute variable est définie dans un shell. Pour qu'elle devienne globale elle doit être exportée par la commande :

export variable

La commande **export** permet d'obtenir la liste des variables exportées.

2.3 Opérateur {} dans les variables

Dans certains cas en programmation, on peut être amené à utiliser des noms de variables dans d'autres variables. Comme il n'y a pas de substitution automatique, la présence de {} force l'interprétation des variables incluses.

Voici un exemple :

```
user="/home/stage"
echo $user
u1=$user1
echo $u1   --> ce n'est pas le résultat escompté !
u1=${user}1
echo $u1
```

2.4 Variables d'environnement

Ce sont les variables système dont la liste est consultable par la commande **env**.

Les plus utiles sont $HOME, $PATH, $USER, $SHELL, $ENV

Exemple :

```
$ moi=Toto
$ p="Je m'appelle $moi"
$ echo Aujourd\'hui, quel jour sommes nous ? ; read jour
echo aujourd'hui $jour, $p sous le nom $USER, est connecté à la station
$HOSTNAME
```

2.5 Variables prédéfinies spéciales

Elles sont gérées par le système et s'avèrent très utiles dans les scripts. Bien entendu, elles ne sont accessibles qu'en lecture.

Ces variables sont automatiquement affectées lors d'un appel de script suivi d'une liste de paramètres. Leurs valeurs sont récupérables dans $1, $2 ...$9

- **$?**.....C'est la valeur de sortie de la dernière commande. Elle vaut 0 si la commande s'est déroulée sans pb.
- **$0** .....Cette variable contient le nom du script
- **$1 à $9** .....Les (éventuels) premiers arguments passés à l'appel du script
- **$#** .....Le nombre d'arguments passés au script
- **$*** .....La liste des arguments à partir de $1
- **$$** .....le n° PID du processus courant
- **$!**.....le n° PID du processus fils

```
ls -l
echo $?        ----> 0
ifconfig ttyS1
echo $?         ---> 1
```

2.6 Passage de paramètres

On peut récupérer facilement les compléments de commande passés sous forme d'arguments sur la ligne de commande, à la suite du nom du script, et les utiliser pour effectuer des traitements.

Ce sont les variables système spéciales $1, $2 .... $9 appelées paramètres de position.

Celles-ci prennent au moment de l'appel du script, les valeurs des chaines passées à la suite du nom du script (le séparateur de mot est l'espace, donc utiliser si nécessaire des "").

La commande shift

Il n'y a que 9 paramètres de position de $1 à $9, et s'il y a davantage de paramètres transmis, comment les récupérer ?

shift effectue un décalage de pas +1 dans les variables $: $1 prend la valeur de $2, etc...

La commande set

Exemple

```
a=1 ; b=2 ; c=3
set a b c
echo $1, $2, $3
# les valeurs de a, b, c sont récupérées dans $1, $2, $3
```

3 Le code de retour

3.1 Le code de retour

Le code de retour d'une commande est un mécanisme fourni par le shell (quel qu'il soit) qui signale à l'utilisateur si l'exécution de cette commande s'est bien déroulée ou bien s'il y a eu un problème quelconque. Le code de retour est un petit entier positif ou nul, toujours compris entre 0 et 255.

Par convention, un code de retour égal à 0 signifie que la commande s'est exécutée correctement. Un code différent de 0 signifie soit une erreur d'exécution, soit une erreur syntaxique.

Ce code de retour est accessible par le paramètre spécial **?** (à ne pas confondre avec le caractère générique ? du shell).

```
$ pwd
/home/thomas
$ echo $?
0 => la commande s'est exécutée correctement
```

```
$ ls -l vi
ls: vi: Aucun fichier ou répertoire de ce type
$ echo $?
1 => erreur d'exécution !
```

Dans l'exemple précédent, la commande ls ne trouve pas le fichier correspondant à l'éditeur de texte vi dans le répertoire courant (ce qui est tout à fait normal !) et positionne un code de retour à 1.

Attention, chaque commande a ses propres codes de retour. Par exemple, un code de retour égal à 1 positionné par la commande **ls** n'a pas la même signification qu'un code de retour à 1 positionné par la commande **grep**. Il n'existe qu'une seule solution à ce problème : lire le manuel correspondant à la commande.

3.2 Commande Unix grep

Cette commande affiche sur sa sortie standard l'ensemble des lignes contenant une chaîne de caractères spécifiée en argument, lignes appartenant à un ou plusieurs fichiers textes (ou par défaut, son entrée standard).

La syntaxe générale de cette commande peut s'écrire :

grep [option(s)] chaîne_cherchée [fichier_texte(s)]

Les crochets indiquent que ce qui est à l'intérieur est facultatif (les options et les fichiers textes). La chaîne cherchée, elle, est obligatoire.

```
$ cat /etc/passwd
root:x:0:3:Super User:/root:/bin/bash
daemon:x:2:2:daemon:/sbin:
bertrand:x:103:20::/home/bertrand:/bin/bash
albert:x:104:20::/home/albert:/bin/bash
sanchis:x:122:20::/home/thomas:/bin/bash

$ grep daemon /etc/passwd
daemon:x:2:2:daemon:/sbin:
```

Cette commande affiche toutes les lignes du fichier /etc/passwd contenant la chaîne daemon.

grep positionne un code de retour

- égal à 0 pour indiquer qu'une ou plusieurs lignes ont été trouvées.
- égal à 1 pour indiquer qu'aucune ligne n'a été trouvée.
- égal à 2 pour indiquer la présence d'une erreur de syntaxe ou qu'un fichier mentionné en argument est inaccessible.

```
$ grep daemon /etc/passwd
daemon:x:2:2:daemon:/sbin:

$ echo $?
0
```

```
$ grep toto /etc/passwd
$
$ echo $?
1 => la chaîne toto n'est pas présente dans /etc/passwd
```

```
$ grep thomas turlututu
grep: turlututu: Aucun fichier ou répertoire de ce type
$ echo $?
2 => le fichier turlututu n'existe pas !
```

3.3 Code de retour d'une suite de commandes

Le code de retour d'une suite de commandes est le code de retour de la dernière commande exécutée. Par exemple, le code de retour de la suite de commandes cmd1; cmd2; cmd3 est le code de retour de la commande cmd3.

```
$ pwd; ls vi; echo bonjour
/home/thomas
ls: vi: Aucun fichier ou répertoire de ce type
bonjour
$ echo $?
0 => code de retour de echo bonjour
```

Il en est de même pour le pipeline cmd1 | cmd2 | cmd3. Le code de retour sera celui de cmd3.

```
$ cat /etc/passwd | grep daemon
daemon:x:2:2:daemon:/sbin:
$ echo $?
0 => code de retour de grep daemon
```

```
$ cat /etc/passwd | grep toto
$
$ echo $?
1 => code de retour de grep toto
```

3.4 Code de retour d'un programme shell

Un programme shell (ou script shell) peut être vu comme une suite de commandes à laquelle on a donné un nom. Il s'ensuit que le code de retour d'un programme shell est le code de retour de la dernière commande qu'il a exécutée.

Il est parfois nécessaire de positionner explicitement le code de retour d'un programme shell avant qu'il ne se termine pour signaler une erreur particulière à l'utilisateur : on utilise alors la commande interne exit.

3.5 Commande interne exit

Sa syntaxe est particulièrement simple : exit [n]

Elle provoque l'arrêt du programme shell avec un code de retour égal à n. Si n n'est pas précisé, le code de retour fourni est celui de la dernière commande exécutée.

```
$ cat lvi2
#!/bin/sh
ls vi
exit 23

$ lvi2
ls: vi: Aucun fichier ou répertoire de ce type

$ echo $?
23 => code de retour de exit 23
```

Le programme shell lvi2 positionne un code de retour différent (ici égal à 23) après exécution de la commande ls vi.

3.6 Opérateurs && et || sur les codes de retour

Les opérateurs && et || autorisent l'exécution conditionnelle d'une commande cmd suivant la valeur du code de retour de la dernière commande précédemment exécutée

3.6.1 Opérateur &&

Syntaxe : cmd1 && cmd2

Le fonctionnement est le suivant : cmd1 est exécutée et si son code de retour est égal à 0, alors cmd2 est également exécutée.

```
$ grep daemon /etc/passwd && echo daemon existe
daemon:x:2:2:daemon:/sbin:
daemon existe
```

La chaîne de caractères daemon est présente dans le fichier /etc/passwd, le code de retour renvoyé par l'exécution de grep est 0 ; par conséquent, la commande echo daemon existante est exécutée.

3.6.2 Opérateur ||

Syntaxe : cmd1 || cmd2

cmd1 est exécutée et si son code de retour est différent de 0, alors cmd2 est également exécutée. Pour illustrer cela, créons rapidement un fichier titi et supposons que le fichier toto n'existe pas :

```
$ cp /etc/passwd titi => création de titi
$ ls titi toto
ls: toto: Aucun fichier ou répertoire de ce type
titi
```

```
$ rm toto || echo toto non efface
rm: ne peut enlever `toto': Aucun fichier ou répertoire de ce type
toto non efface
```

Le fichier toto n'existant pas, la commande rm toto affiche un message d'erreur et produit un code de retour différent de 0 : la commande interne echo qui suit est donc exécutée.

3.6.3 Combinaisons d'opérateurs && et ||

Les deux règles mentionnées ci-dessus sont appliquées par le shell lorsqu'une suite de commandes contient plusieurs opérateurs && et ||. Ces deux opérateurs ont la même priorité et leur évaluation s'effectue de gauche à droite.

```
$ ls titi || ls toto || echo fini aussi
titi
```

Le code de retour de ls titi est égal à 0 car titi existe, la commande ls toto ne sera donc pas exécutée. D'autre part, le code de retour de l'ensemble ls titi || ls toto est le code de retour de la dernière commande exécutée, c'est-à-dire est égal à 0 (car c'est le code de retour de ls titi), donc echo fini aussi n'est pas exécuté.

Intervertissons maintenant les deux commandes ls :

```
$ ls toto || ls titi || echo fini
ls: toto: Aucun fichier ou répertoire de ce type
titi
```

Le code de retour de ls toto est différent de 0, donc ls titi s'exécute. Cette commande renvoie un code de retour égal à 0, par conséquent echo fini n'est pas exécuté.

Combinons maintenant opérateurs && et || :

```
$ ls titi || ls toto || echo suite et && echo fin
titi
fin
```

La commande ls titi est exécutée avec un code de retour égal à 0, donc la commande ls toto n'est pas exécutée, donc le code de retour de l'ensemble ls titi || ls toto est égal à 0, donc la commande echo suite et n'est pas exécutée, donc le code de retour de ls titi || ls toto || echo suite et est égal à 0, donc la commande echo fin est exécutée ! (ouf !!!).

4 Les tests

4.1 La commande test

Comme son nom l'indique, elle sert à vérifier des conditions. Ces conditions portent sur des fichiers (le plus souvent), ou des chaines ou une expression numérique.

Cette commande courante sert donc à prendre des (bonnes) décisions, d'où son utilisation comme condition dans les structures conditionnelles if.. then ..else

Syntaxe

test expression

[expression] attention aux espaces autour de expression

Code de retour

La commande test retourne 0 si la condition est considérée comme vraie, une valeur différente de 0 sinon pour signifier qu'elle est fausse.

4.2 Tester un fichier

Elle admet 2 syntaxes (la seconde est la plus utilisée) :

- **test option fichier**
- **[option fichier]**

Principales options :

Option	***Signification quant au fichier***
-e	il existe
-f	c'est un fichier normal
-d	c'est un répertoire

-r | -w | -x	il est lisible | modifiable | exécutable
-s	il n'est pas vide

Exemples :

```
[ -s $1 ] ⇒ vrai (renvoie 0) si le fichier passé en argument n'est pas
vide
[ $# = 0 ] ⇒ le nombre d'arguments est 0
[ -w  fichier ] ⇒ le fichier est-il modifiable ?
[ -r "/etc/passwd" ] ⇒ puis-je lire le fichier /etc/passwd ?
echo $? ⇒ 0 (vrai)
[ -r "/etc/shadow" ] puis-je lire le fichier /etc/shadow ?
echo $? ⇒ 1 (faux)
[ -r "/etc/shadow" ] || echo "lecture du fichier interdite"
```

4.3 Tester une chaine

[option chaine]

Options

- -z | -n ⇒ la chaine est vide / n'est pas vide
- = ⇒ chaînes identiques
- != ⇒ chaînes différentes

Exemples

```
[ -n "toto" ] ; echo $?  affiche le code de retour 0
ch="Bonjour" ; [ "$ch" = "bonjour" ] ; echo $?  affiche 1
[ $USER != "root" ] && echo "l'utilisateur n'est pas le \"root\" !"
```

4.4 Tester un nombre

[nb1 option nb2]

Il y a d'abord un transtypage automatique de la chaine de caractères en nombre

Options

- -eq | -ne ⇒ égal | différent
- -lt | -gt ⇒ strictement inférieur | strictement supérieur
- -le | -ge ⇒ inférieur ou égal | supérieur ou égal

Exemples

```
a=15 ; [ "$a" -lt 15 ] ; echo $?
```

4.5 Opérations dans une commande test

[expr1 -a expr2] ⇒ (and) 0 si les 2 expressions sont vraies

[expr1 -o expr2] ⇒ (or) 0 si l'une des 2 expressions est vraie

[! expr1] ⇒ négation

Exemples :

```
f="/root" ; [ -d "$f" -a -x "$f" ] ; echo $?
note=9; [ $note -lt 8 -o $note -ge 10 ] && echo "tu n'es pas convoqué(e) à
l'oral"
```

4.6 Structures conditionnelles

```
if suite-de-commandes
then
      # séquence exécutée si suite-de-commandes rend une valeur 0
      bloc-instruction1
else
      # séquence exécutée sinon
      bloc-instruction2
fi
```

Attention ! si then est placé sur la 1ère ligne, séparer avec un ;

if commande; then

Exemples :

1. toto possède-t-il un compte ? On teste la présence d'une ligne commençant par toto dans /etc/passwd (>/dev/null pour détourner l'affichage de la ligne trouvée)

```
if grep "^toto" /etc/passwd > /dev/null
then
      echo "toto a déjà un compte"
fi
```

2. Si toto a eu une bonne note, on le félicite

```
note=17
if [ $note -gt 16 ] ---> test vrai, valeur retournée : 0
then echo "Très bien !"
fi
```

4.7 Conditionnelles imbriquées

Pour imbriquer plusieurs conditions, on utilise la construction :

```
if commande1
then
      bloc-instruction1
elif commande2
then
      bloc-instruction2
else
      # si toutes les conditions précédentes sont fausses
      bloc-instruction3
fi
```

Exemples :

1. toto a-t-il fait son devoir lisiblement ?

```
fichier=/home/toto/devoir1.html
if [ -f $fichier -a -r $fichier ]
then
      echo "je vais vérifier ton devoir."
elif [ ! -e $fichier ]
then
      echo "ton devoir n'existe pas !"
else
      echo "je ne peux pas le lire !"
fi
```

2. Supposons que le script exige la présence d'au moins un paramètre, il faut tester la valeur de $#, est-elle nulle ?

```
if [ $# = 0 ]
then
      echo "Erreur, la commande exige au moins un argument .."
      exit 1
elif [ $# = 1 ]
then
      echo "Donner le second argument : "
      read arg2
fi
```

4.8 Choix multiples

```
case valeur in
      expr1) commandes ;;
      expr2) commandes ;;
      ...
esac
```

Exemples :

1. Supposons que le script doive réagir différemment selon l'user courant ; on va faire plusieurs cas selon la valeur de $USER

```
case $USER in
      root) echo "Mes respects M le $USER" ;;
      jean | stage?) echo "Salut à $USER" ;;
      toto) echo "Fais pas le zigo$USER \!" ;;
esac
```

2. Un vrai exemple, extrait du script smb (/etc/rc.d/init.d/smb)

```
# smb attend un paramètre, récupéré dans la variable $1
case "$1" in
start)
      echo -n "Starting SMB services: "
      deamon smbd -D
      echo
      echo -n "Starting NMB services: "
      deamon nmbd -D
      ...
stop)
      echo -n "Shutting SMB services: "
      killproc smbd
      ....
esac
```

5 Les boucles

5.1 Boucle FOR

Syntaxe:

```
for variable [in liste]
do
      commandes (utilisant $variable)
done
```

Fonctionnement :

Ce n'est pas une boucle for contrôlée habituelle fonctionnant comme dans les langages de programmation classiques (utiliser pour cela une boucle while avec une variable numérique).

La variable parcourt un ensemble de fichiers données par une liste ou bien implicitement et le bloc commandes est exécuté pour chaque de ses valeurs.

Les mots-clés do et done apparaissent en début de ligne (ou après un ;)

La liste peut être explicite :

```
for nom in jean toto stage1
do
      echo "$nom, à bientôt"
done
```

La liste peut être calculée à partir d'une expression modèle

```
# recopier les fichiers perso. de toto dans /tmp/toto
for fich in  /home/toto/*
do
      cp $fich tmp/toto
done
```

Si aucune liste n'est précisée, les valeurs sont prises dans la variable système $@, c'est-à-dire en parcourant la liste des paramètres positionnels courants.

```
# pour construire une liste de fichiers dans $@
cd  /home/stagex ; set * ; echo $@
for nom in $@
      do echo $nom
done
```

5.2 Boucle WHILE

Syntaxe :

```
while liste-commandes
do
      commandes
done
```

Exemple :

```
echo -e "Entrez un nom de fichier"
read fich
while [ -z "$fich" ]
do
      echo -e "\nSaisie à recommencer" #
      read fich
done
```

5.3 Boucle DO..UNTIL

Syntaxe :

```
until liste-commandes
do
      commandes
done
```

6 grep et les expressions régulières

6.1 grep

La commande « grep chaîne fichier » permet d'extraire de fichier toutes les lignes contenant chaîne.

Si chaîne contient des espaces, elle doit être encadrée par deux guillemets simples.

La commande grep est sensible à la casse. Afin que la commande grep ignore la différence entre Majuscule et minuscule, utilisez l'option –i.

Afin de chercher une phrase, ou un mot en entier, vous devez l'entourer de guillemets simples ou doubles :

grep –i 'eruption volcanique' total.txt

grep –i "volcan " total.txt

Afin d'afficher toutes les lignes qui ne contiennent pas un motif, utiliser l'option –v. On pourrait ainsi combiner plusieurs commandes grep à la suite à l'aide de | :

grep -i volcan VOLCFR.txt | grep -v Rittman | more

L'option **–n** permet d'afficher le numéro de ligne dans le fichier auquel on a trouvé le motif recherché.

grep –i –n VOLCFR.txt

L'option **–c** permet d'afficher uniquement le nombre total de lignes correspondant au motif.

grep –c volcan science.txt

est équivalente à

grep volcan science.txt | wc –l

L'option **–l** permet d'afficher juste les noms des fichiers dans lesquels un motif apparaît, sans afficher les occurrences trouvées.

L'option **–m** NUM permet de s'arrêter après NUM solutions trouvées.

grep –m 20 volcan VOLCEN.txt

nous permet de limiter notre recherche à 20 résultats.

6.2 Extension : l'utilisation de la commande grep avec les expressions régulières.

La syntaxe de la commande grep que nous utiliserons sera de la forme

grep -E "expression" fichier.

- L'argument expression est ce qu'on appelle une expression régulière, une suite de symboles décrivant un ensemble (fini ou infini) de mots. Noter que cet argument est encadré par des guillemets doubles.
- L'argument -E indique que cette expression est une expression étendue, par opposition aux expressions dites de base, dont la syntaxe est différente de celle décrite ci-dessous (mais certainement pas plus basique).

- En sortie, grep renvoie toutes les lignes de fichier contenant au moins un mot décrit par expression.

6.3 Introduction aux expressions régulières

Les expressions régulières sont des chaînes de caractères qui peuvent être utilisées pour retrouver un ensemble de chaînes de caractères. Par exemple, pour retrouver toutes les occurrences du verbe « écrire » dans un texte on aurait besoin d'un nombre considérable de requêtes pour chacune des formes : écris, écrirais, écrit, écrivant, etc. En utilisant une expression régulière comme écri[a-z]+ on peut retrouver toutes ces formes en même temps.

De la même façon en utilisant l'expression [Ww]ork(s|ing|ed)? on peut retrouver les formes Work, work, Works, works, Working, working, Worked ou worked.

Dans cet exemple, la suite de caractères [a-z] désigne l'intervalle a-z (n'importe quel caractère - un seul caractère - dans cette intervalle) alors que le caractère + est un quantificateur indiquant 'une ou plusieurs' occurrences de la sous-expression précédente. L'expression « écri[a-z]+ » dans son ensemble correspond à l'ensemble de chaînes de caractères comportant le motif écri suivi d'au moins une lettre minuscule.

Autres éléments de syntaxe des expressions régulières :

- Le caractère **.** (point) remplace n'importe quel caractère.
    - b.lle correspond aux chaînes de caractères balle, belle, bulle, bille mais aussi b.lle, bplle, b3lle etc.
- [ab2X] décrit l'ensemble de caractères a,b,2,X et l'expression x[ab2X]y décrit les chaînes xay, xby, x2y, xXy
- [a-c],[0-38] et [a-d5-8X-Z] désignent respectivement l'ensemble de caractères (n'importe quel caractère de l'ensemble) a,b,c, un des caractères numériques 0,1,2,3,8 (les caractères dans l'intervalle 0-3 plus le caractère 8) et un des caractères suivants : a,b,c,d,5,6,7,8,X,Y,Z
- Il est possible d'exclure un ensemble de caractères d'une requête à l'aide du caractère ^ : [^0-9] décrit n'importe quel caractère qui n'est pas un chiffre, [^] décrit n'importe quel caractère qui n'est pas un espace, etc.
- Les caractères ^ et $ permettent de retrouver un motif en début respectivement en fin de ligne.
    - NOTE : le caractère ^ n'a la signification de non qu'entre crochets, comme ci-dessus.
- X* désigne une suite quelconque d'occurrences de X (0 ou plus)
- X+ désigne au moins une occurrence de X (1 ou plus)
- X? désigne une occurrence optionnelle de X (0 ou 1)

- X{n, m} entre n et m occurrences de x (au moins n, au plus m)
- Enfin le caractère | (ou logique) permet de retrouver des expressions régulières alternatives : chats|chiens|souris

NOTE : étant donné que les guillemets (" ou ') font partie de la syntaxe de la commande grep, afin de les inclure dans une commande, et les caractères spéciaux *,+,.,?, [,], {,} ont un rôle dans la syntaxe des expressions régulières, on doit les protéger avec un \

6.4 Exemples d'utilisation avec la commande Grep

Exemple	***Explication***
`grep gh`	retrouver les lignes contenant"gh"
`grep -E '^con'`	retrouver les lignes commençant avec"con"
`grep -E 'ing$'`	retrouver les lignes finissant en "ing"
`grep –v gh`	ignorer les lignes contenant "gh"
`grep –v -E '^con'`	ignorer les lignes commençant en "con"
`grep –v -E 'ing$'`	ignorer les lignes finissant en "ing"
`grep -E '[A-Z]'`	les lignes comportant une majuscule
`grep -E '^[A-Z]'`	les lignes commençant avec une majuscule
`grep -E '[A-Z]$'`	les lignes finissant avec une majuscule
`grep -E '^[A-Z]*$'`	les lignes comportant uniquement des Majuscules
`grep -E '[aeiouAEIOU]'`	les lignes comportant une voyelle
`grep -E '^[aeiouAEIOU]'`	les lignes commençant avec une voyelle
`grep -E '[aeiouAEIOU]$'`	les lignes finissant avec une voyelle
`grep –i -E '^[^aeiou]'`	les lignes commençant avec une non-voyelle
`grep –i -E '[^aeiou]$'`	les lignes finissant avec une non-voyelle
`grep –i -E '[aeiou].*[aeiou]'`	les lignes avec deux ou plusieurs voyelles
`grep –iE '^[^aeiou]*[aeiou][^aeiou]*$'`	les lignes comportant exactement une voyelle

7 Exemples de scripts

```
# emploi des variables du shell
# inversion des arguments
echo $1  $2  $3
echo $3  $2  $1
echo
echo La commande $0 possède $# paramètres qui sont:
echo    $*
```

```
# Rechercher une chaine dans un fichier, (la chaîne et le nom du fichier
sont passés en paramètres)
if grep "$1" "$2" ;
then  echo $1 se trouve dans $2 ;
else echo "$1 ne se trouve pas dans $2";
fi
```

```
# Rechercher le nbre de lignes blanches dans un fichier dont le nom est
passé en paramètre
echo Le fichier $1 contient `grep '^$' $1 | wc -l` lignes \ blanches
```

```
# Recherche dans le répertoire courant les fichiers dont le nom est composé
de 4 caractères et a comme extension .c
ls -l `pwd` | grep '....\.c$'
```

```
# Donne le nbre de lignes contenant une chaine dans un fichier (passés en
paramètres)
if [ -z $2  ];
      then echo Syntaxe: $0 chaine  fichier;
      else echo Il y a `grep $1 $2 | wc -l` lignes contenant $1 \ dans le
      fichier $2;
fi

if [ -z "$1" || -z "$2" ];    # OU logique
                              # équivalent à   if [ $# -lt  2] ;
      then echo Il faut 2 arguments !
      elif [ $1 -eq  $2 ] ;
            then echo Les 2 arguments sont égaux
      else echo Les 2 arguments sont différents
fi
```

```
# Utilisation de for dans un script
for i in *
      #  * designe l'ensemble des fichiers du répertoire courant
do    if [ -d  $i ] ;
            then echo $i est un répertoire
            else echo $i est un fichier
      fi
done
```

```
# utilisation de case dans un script
if [ -z "$1" ] ;
 then echo Erreur: Argument manquant !
else
      case $1 in
            [1-9]*)     echo La chaîne commence par un chiffre;;
            ??)         echo La chaîne fait 2 caractères;;
            [aeiuoyAEIUOY]*)  echo La chaine commence par une \ voyelle;;
            w* | z*)    echo La chaîne commence par w ou par z;;
            *)          echo La chaîne est d\'un type non reconnu;;
      esac
fi
```

```
# Affichage un message toutes les n secondes, n étant fourni en paramètre
if [ -z "$1" ]; then echo La syntaxe est :   $0  nbsec
else
      echo Début de la boucle;
      i=0
      while :                 #  while :     pour boucle infinie
      do
            sleep  $1 ; i=`expr $i + $1`
            echo  $i secondes se sont écoulées
      done
fi
```