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'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.

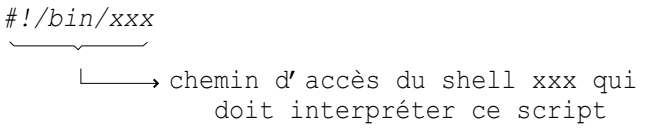
Et bien entendu, les mécanismes de tubes et de redirections sont utilisables dans un script.

Remarques

1) 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).

2) 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 ou un autre 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.

3) Si un script commence par la ligne



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

Notre objectif ici n'est pas une étude exhaustive de la programmation en shell, mais une introduction à cette technique illustrant les notions et instructions principales.

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 :

```
xstra> listf
./    .kshrc    .securite/    bin/   florent/   xavier/
../   .profile  .sh_history   dpt/   jerome/
xstra>
```

Exécution du script en mode mise au point :

```
xstra> sh  x listf   § mode trace
```

ou

```
xstra> sh  v listf   § mode verbose
```

permettent de demander au shell qui interprète le script de tracer le déroulement du script ou de le commenter (ou les deux). Il est également possible d'inclure **dans** le script les lignes :

```
set  x        (pour le mode trace)
set  v        (pour le mode verbose)
```

Le **mode trace** recopie sur la sortie standard chaque ligne telle qu'elle est interprétée. Le **mode verbose** recopie sur la sortie standard chaque ligne avant interprétation.

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

Avec *vi*, 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
```

```
./  ../  df_file
xstra>
```

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>
```

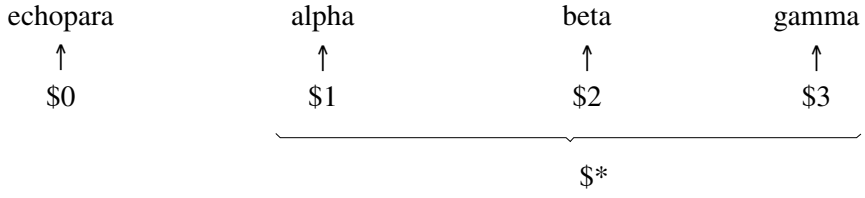
8.1.3 Les variables spéciales

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



Exemple 2

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.

Exemple

```
tempfil /tmp/infile_user_$$
$ utilisation du fichier temporaire
rm $tempfil
```

8.1.4 Les caractères spéciaux

Attention

Ces caractères sont générateurs des noms de fichiers et sont utilisés dans le passage des paramètres.

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 *expression* 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
```

Exemple 2

Le script *append* contient :

```
case $# in
2) if [ ! f $1 ]
    then
    echo le fichier $1 ne se trouve pas dans le repertoire
    elif [ ! s $1 o ! r $1 ]
      then
        echo le fichier $1 est vide ou protege en lecture
    elif [ ! f $2 ]
      then
        cat $1 >$2
        echo $0 a copie $1 en $2
    elif [ ! w $2 ]
      then
        echo le fichier $2 est protege en ecriture
    else
```

```
        cat $1 >>$2
        echo $0 a rajoute $1 en fin de $2
fi
;;
*) cat <<EOT
     append
     Fonction : le fichier fsource est ajoute au fichier fcible
     Si fcible n'existe pas, fsource est copie en fcible
     Syntaxe : append fsource fcible
EOT
;;
esac
```

Exemple 3

Le script *r4d* contient :

```
# r4d : real*4 dump ( r8d : real*8 dump )
# i2d : integer*2 dump
# ce script permet d'examiner un fichier binaire
# contenant des nombres réels ou entiers codés
# sur 2 ou 4 octets
case $(basename $0) in
r4d) option ' t f4' ;;
r8d) option ' t f8' ;;
i2d) option ' t d2' ;;
i4d) option ' t d4' ;;
esac
od $option $1 | less
```

Dans le répertoire contenant *r4d*, on crée les liens *r8d i2d* et *i4d*

```
xstra> ln r4d r8d; ln r4d i2d; ln r4d i4d
```

On peut maintenant utiliser les commandes *r4d, r8d, i2d, i8d* qui correspondent à un seul et même shell script dont le comportement dépend du nom sous lequel il aura été invoqué.

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 systeme Linux
le
systeme
```

```
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
4
5
6
xstra>
```

8.1.7 Script récapitulatif

```
#!/bin/bash
# Le nom de ce script est cherche.
# Ce script illustre: le passage d'arguments,
#                le test sur le nombre des arguments,
#                le test sur le nom d'invocation
#                la combinaison logique de conditions
#                les redirections
#                le "single quoting",le "double quoting"
#                et le "back quoting"
#
# Ce script recherche tous les fichiers
# dont le nom est passe en  premier argument.
# La recherche est faite dans toute l'arborescence,
# a partir d'un repertoire passe en deuxieme argument.
# Si ce deuxieme argument est absent,la recherche
# est faite a partir du repertoire courant.
# Les messages d'erreur pouvant resulter de
# l'absence de droits d'acces a des sous repertoires
# sont ignores. La liste des fichiers trouves est placee
# dans un fichier temporaire.
# La recherche pouvant etre longue, ce script est de
# preference lance en arriere plan.
# Lorsque la recherche est terminee, invoquer ce script
# sans argument permet de connaitre le resultat
```

```
# de la recherche. Il est montre comment utiliser
# le nom d'invocation pour prendre une decision :
# un lien a ete cree par la commande ln cherche montre.
# Invoquer le script par montre, sera equivalent
# a l'invoquer par cherche sans arguments.
#
# exemple: xstra> cherche passwd / &
#                ..............
#                un certain temps s'ecoule,
#                mais on peut faire autre chose...
#                ..............
#          xstra> montre           (ou xstra> cherche)
#
#          Resultat de la recherche:
#          /etc/passwd
#          /bin/passwd
#
#          xstra>
# exemple: xstra> cherche 'pass*' / &
#          meme resultat: dans ce cas
#          les simples apostrophes evitent
#          le remplacement de pass*
#          lors de l'invocation du script, et
#          les doubles apostrophes autour de $1
#          dans le find permet la recherche desiree.
#          Sans les simples apostrophes,
#          le shell en cours remplacerait
#          pass* par la liste de tous les noms
#          de fichier commencant par pass
#          dans le repertoire courant,
#          avant d'invoquer le script.
#          Sans les doubles apostrophes,
#          le shell ferait de meme, mais pendant
#          l'interpretation du script:
#          le resultat depend alors de l'existence
#          de fichiers dont le nom commence par pass
#          dans le repertoire courant.
#
TEMPFILE /tmp/cherche_temporary_file_$LOGNAME
PROG $(basename $0)
if [  $#  ne 0  a $PROG !  montre ] ;then
#                 invoque par cherche avec argument(s)
  rm $TEMPFILE 2>/dev/null
  RACINE `pwd`
  if [  $#  eq 2 ]
    then RACINE $2
  fi
  (find $RACINE  name ”$1”  print) 1>$TEMPFILE 2>/dev/null
  if [  !  s $TEMPFILE ]
    then echo "$1 non trouve a partir de $RACINE" >$TEMPFILE
```

```
  fi
else # invoque sans arguments ou par le nom: montre
 if [  !  s $TEMPFILE ]
    then echo "Pas de recherche en cours."
  else
    echo ""
    echo "Resultat de la recherche:"
    more $TEMPFILE
    echo ""
    rm $TEMPFILE
  fi
fi
```

8.1.8 Arithmétique entière sur des variables

Le Bourne-shell ne permet pas de définir des variables numériques, et pourtant les opérateurs -*eq ne lt le gt ge* existent. Ce n'est pas une incohérence. Toutes les variables définies en Bourne-shell sont de type chaîne de caractères, mais si le contenu de cette chaîne représente un nombre entier, les opérateurs précédents peuvent s'appliquer.

Les opérations arithmétiques sur variables sont très faciles en bash grâce à deux mécanismes non disponibles dans le Bourne-shell traditionnel :
L'évaluation arithmétique :

$((variable1 opérateur variable2))

Le test arithmétique :

if ((variable1 opérateur variable2))

Ces deux mécanismes seront présentés plus loin.

8.2 LA PROGRAMMATION AVANCÉE EN BASH

L'étude exhaustive du Bash en tant que langage de commandes nécessiterait un livre entier. Nous présenterons donc simplement les points les plus importants par rapport aux concepts de base exposés lors du paragraphe précédent. De plus il existe des différences entre les versions du Bash. Dans ce paragraphe nous considérerons être dans la version 2 (*bash version*). Certains points développés ci-dessous ne fonctionnent pas en version 1.

8.2.1 Les variables prédéfinies du Bash (non définies en Bourne-shell)

Les variables mises à jour dynamiquement par le Bash sont les suivantes :

PPID	numéro du processus père (Parent Process IDentifier),
PWD	répertoire de travail,
RANDOM	un nombre aléatoire,
SECONDS	temps écoulé depuis le lancement de ce Korn-shell,
!	numéro du dernier processus lancé en arrière-plan,

_ (underscore) dernier mot de la dernière commande exécutée.

8.2.2 Définition de variable : la commande declare

En plus de la forme *variable=valeur*, le Bash possède une commande générale de définition de variable :

declare [+/ option] [p] [variable[valeur]]

option	affecte l'attribut à variable
+option	enlève l'attribut à variable

L'option *–p*, exclusive des autres options, permet d'afficher les variables et leur valeur. L'option *–f* ou *–F* permet d'afficher les fonctions avec ou sans leur définition. Les autres options sont :

-a	*variable* est de type tableau,
-i	*variable* est de type numérique entier,
-r	variable en lecture seule,
-x	variable exportée (variable placée dans l'environnement)

La commande *typeset* est équivalente à *declare*.

Dans le cas de la création d'un tableau la syntaxe la plus simple est :

declare a tab=(element1 element2 element3)

Le tableau *tab* aura 3 éléments qui seront : *tab[0]=element1*, *tab[1]=element2*, *tab[2]=element3*. La création d'un élément du tableau *tab* est réalisée en utilisant la syntaxe simple *tab[numero element]=valeur*.

Attention

Les éléments d'un tableau de variables sont numérotés de `0` à `N 1`.

Exemple

```
xstra> declare  a arbres=(sapin chene acacia orme)
§ définition du tableau arbres
xstra> declare  p arbres
declare  a arbres '([ 0]  "sapin" [ 1]  "chene" [ 2]  "acacia"
[ 3]  "orme")'
xstra> echo $arbres        § désigne le premier élément sapin
sapin
xstra> echo ${arbres[2]}   § désigne l'élément numéro 2 (0, 1, ..)
acacia
xstra> echo ${arbres[*]}   § désigne la suite de tous les éléments
sapin chene acacia orme
xstra> echo ${#arbres[1]} § désigne la taille de l'élément 1
5
xstra> echo ${#arbres[*]} § désigne le nombre d'éléments
```

```
4
xstra>
```

L'option *+a* n'existe pas. Pour détruire un tableau il faut utiliser la commande *unset tab*.

8.2.3 La commande test

La commande *test expression* ou *[expression]* est la même qu'en Bourne-shell, mais le jeu possible pour *expression* y est plus riche.

a) L'opérateur == peut remplacer = pour la comparaison

c1 == c2 vrai si les chaînes de caractères c1 et c2 sont identiques, (notation préférable au simple =)

b) Test sur le type d'un fichier

expression - Le code de retour est vrai si :

-a fich *fich* existe
-b fich *fich* est de type spécial bloc
-c fich *fich* est de type spécial caractère
-d fich *fich* est un répertoire (directory)
-f fich *fich* est un fichier
-L fich *fich* est un lien symbolique
-p fich *fich* est un tube nommé (pipe)
-s fich *fich* existe et est de taille non nulle

c) Test de relations entre fichiers

expression - Le code de retour est vrai si :

fich1 -ef fich2 *fich2* est un lien sur *fich1* (*ln*)
fich1 -nt fich2 *fich1* est plus récent que *fich2* (newer than)
fich1 -ot fich2 *fich1* est plus ancien que *fich2* (older than)

d) Test des droits d'accès d'un fichier

expression - Le code de retour est vrai si :

-r fich le processus en cours a le droit r sur *fich*
-w fich le processus en cours a le droit w sur *fich*
-x fich le processus en cours a le droit x sur *fich*
-g fich *fich* a le bit SGID positionné
-u fich *fich* a le bit SUID positionné

e) Le test [[…]]

Le test entre doubles crochets diffère du test entre simples crochets pour la comparaison entre chaînes de caractères avec les opérateurs et ! . Entre simples crochets, ces opérateurs comparent une chaîne à une autre chaîne ; entre doubles crochets, ils comparent une chaîne à un motif. Un motif peut contenir des caractères générateurs de noms.

L'exemple suivant illustre cette différence : comment tester si la variable TERM contient vt100 ou vt220 ou vt330 ou vtxxx…

Exemple

```
if [   $TERM    vt100 ] # 1 # correct, mais insuffisant
if [   $TERM    vt*   ] # 2 # FAUX
if [[  $TERM    vt*   ]] # 3 # correct : la bonne solution
# entre simples crochets il faudrait ecrire :
if [  $TERM    vt100  o $TERM    vt220  o $TERM    vt330 ]
```

Dans la ligne 2 de cet exemple, le shell remplace vt* par la liste de tous les noms de fichiers qui commencent par vt dans le répertoire courant avant de faire le test, ce qui n'est sans doute pas le but recherché.

8.2.4 L'arithmétique entière

L'utilisation de variables entières n'est pas prévue en Bourne-shell, et les manipulations arithmétiques y sont très incommodes. En Bash, l'arithmétique entière est facile, grâce à une notation adaptée : le double parenthésage **((…))**. Entre des doubles parenthèses, le bash interprète les caractères **< > () *** selon leur signification arithmétique usuelle, et le caractère $ n'est pas nécessaire devant un nom de variable. Le parenthésage y est possible, et sans parenthésage, la priorité des opérateurs arithmétiques est la priorité usuelle. Cette notation offre un cadre cohérent pour l'évaluation arithmétique et le test arithmétique.

a) L'évaluation arithmétique : $((…))

Quelques exemples valent mieux qu'une description formelle.

```
declare  i n1 n2 n3 x
n1 17
n2 3          # jusqu'ici rien de nouveau
n3 $((17/3))  # n3   17/3   5 (en entier)
n3 $((17%3))  # % : le reste de la division : n3 2
# la ligne suivante montre un calcul plus complexe
n1 $((n2*(n1+27) 5)) # n1 127
# l'écriture est très claire entre (( et ))
# c'est l'écriture arithmétique usuelle.
x $((1<<4))   # << : décalage à gauche : x 16
```

b) Le test arithmétique : if ((…))

Le test arithmétique offre les mêmes facilités d'écriture. Ceci est particulièrement pratique pour les caractères **<** **>** qui ne désignent plus des redirections, mais les opérateurs relationnels arithmétiques.

Exemples

```
if ((x>1000)) ; then           # valide et très lisible
# entre [ ] il faudrait écrire
if [  $x  gt 1000 ] ; then     # moins lisible

# encore plus convaincant :
if (( n2*(n1+27) 5) > n3 )) ; then # valide
if ((record_size*nbr_record > bufferlength)) ; then
```

Sans la notation ((…)), les deux dernières lignes de l'exemple précédent exigeraient des contorsions complètement illisibles.

Notons qu'il n'est pas nécessaire de déclarer les variables entières par la commande `declare i` pour utiliser l'évaluation et le test arithmétique. Il suffit que les variables contiennent des chaînes dont l'évaluation soit un entier.

8.2.5 L'écriture de script

Le Bash est un langage très puissant, sa syntaxe est complexe. Nous présenterons brièvement les mécanismes de parenthésage et de substitution, sources d'erreur de syntaxe.

a) Le parenthésage

Une suite de commandes peut être parenthésée par `()` ou par `{}` :

```
(commande1 ; commande2 ; commande3)
```

Les trois commandes sont exécutées dans un sous-shell.

```
{ commande1 ; commande2 ; commande3 ; }
```

Les trois commandes sont exécutées dans le shell en cours.

Attention

Ne pas oublier l'espace après `{` et le `;` avant `}`.

b) La substitution

`$variable` ou `${variable}`

Désigne la valeur de la variable. Si `variable` est un tableau, `$variable` ou `${variable}` désigne son premier élément (voir l'exemple arbres au paragraphe 8.2.2). Dans un shell script, la notation `${variable}` doit être préférée à la notation `$variable`.

Attention

Si *variable* n'est pas définie, le Bash substitue la chaîne vide.

${#variable}

Désigne la longueur de la *variable* (ou de son premier élément si c'est un tableau). Cas particulier : *${*}* ou *$** désigne le nombre de variables de position.

${#variable[*]}

Désigne le nombre d'éléments du tableau *variable* (voir l'exemple arbres au paragraphe 8.2.2).

${variable: chaine}

Désigne la valeur de la *variable* si celle-ci est définie et non vide, sinon désigne la chaîne *chaine*. Très utile pour fixer une valeur par défaut, par exemple :

```
TERM ${ TERM: VT220}
```

${variable:=chaine}

Désigne la valeur de la *variable* si celle-ci est définie et non vide. Sinon, la valeur *chaine* lui est affectée, puis la substitution a lieu.

${variable:?chaine}

Désigne la valeur de la *variable* si celle-ci est définie et non vide. Sinon, *chaine* est envoyée par le shell sur la sortie standard. Si ce shell n'est pas interactif, il se termine.

${variable:+chaine}

Désigne la chaîne *chaine* si la *variable* est définie et non vide. Sinon, désigne la chaîne vide.

c) La substitution de commande $(commande)

Le mécanisme de **substitution de commande** ou **backquoting** a été présenté au paragraphe 7.5. Le Bash reconnaît la notation POSIX *$(commande)*, beaucoup plus claire en cas d'utilisation à plusieurs niveaux d'imbrication. L'exemple suivant illustre cette possibilité. Comment copier dans le répertoire */tmp/backup* tous les fichiers dont le nom est **log** et qui se trouvent dans les répertoires listés par la variable PATH. La commande *tr* qui permet de remplacer le caractère *:* par un espace est expliquée au paragraphe 14.5.

```
xstra> cp $(find $(echo $PATH | tr ':' ' ') \
-type f -name '*log*' -print) /tmp/backup
xstra>
```

d) Les extensions Bash dans la génération de noms

Le Bash possède un mécanisme étendu de génération de noms de fichier permettant de décrire des noms de fichiers respectant certains motifs. Sa syntaxe diffère de la

syntaxe des expressions régulières utilisées par *sed* et *awk*. Le tableau suivant résume cette syntaxe.

**(motif)*	0 ou 1 ou plusieurs occurrences de motif
+(motif)	1 ou plusieurs occurrences de motif
?(motif)	0 ou 1 occurrence de motif
@(motif1|motif2)	motif1 ou motif2
!(motif)	tout sauf motif

Ce mécanisme peut être utilisé dans un test, ce qui le rend très pratique en programmation. Les exemples suivants illustrent cette possibilité :

Exemples

1) Supprimer tous les fichiers sauf les fichiers ayant l'extension *.c* ou *.h* et les fichiers dont les noms commencent par *Makefile* ou *makefile* ou *README* :

```
rm !(*.c|*.h|[Mm]akefile*|README*)
```

2) Dans un script, prendre une décision sur le nom de fichier. Si le nom de fichier comporte une extension *.c* ou *.h*, faire ceci, sinon, faire cela :

```
#!/bin/bash
for filename in * ; do
   if [  $filename   @(*.c|*.h|[Mm]akefile*|README*) ]
   then
   fi
done
```

Attention : Ne pas oubliez d'inclure soit dans le script soit dans le fichier *~/.bashscript* l'option *extglob* à l'aide de la commande *shopt s extglob*.

3) Si la variable TERM contient vt100 ou vt200 ou vtxxx, ou xterm, faire

```
if [  $TERM   @(vt+([0 9])|xterm) ]; then
```

e) Le quoting (neutralisation)

Nous avons vu au paragraphe 7.8 que les caractères spéciaux du shell peuvent être neutralisés de plusieurs façons différentes. Ce problème est encore plus souvent rencontré dans un shell script qu'en ligne de commande interactive, et la question est : « Quels sont les caractères spéciaux neutralisés dans une chaîne placée entre doubles apostrophes (doubles quotes) ? »

Maintenant que tous les mécanismes d'évaluation ont été présentés, nous pouvons énoncer une règle très simple à mémoriser :

Dans une chaîne entre doubles quotes, tous les caractères spéciaux sont neutralisés **sauf le caractère $ sous toutes ses formes** :

Évaluation de variable	**`$TERM, ${TERM:-vt100}`**
Évaluation de commande	**`$(cmd)`**, et donc aussi **`` `cmd` ``**
Évaluation arithmétique	**`$((LINES*COLUMNS))`**

Nous n'en dirons pas plus sur la programmation en Bash. Toute la description de la programmation décrite précédemment s'applique pour l'écriture de scripts en Bash.

8.3 EXERCICES

Exercice 8.3.1

Écrire un shell script qui écrit sur sa sortie standard les messages suivants :
mon nom est xxx
je suis appele avec yyy arguments
qui sont: 111 222 333 444
(xxx sera remplacé par le nom sous lequel ce shell script aura été invoqué, yyy par le nombre d'arguments et 111, 222, etc. par les arguments en question). Quand ce script fonctionnera correctement, invoquez le avec les cinq arguments :

Bienvenue dans le monde Linux
puis avec un seul argument contenant la chaîne de caractères : Bienvenue dans le monde Linux

Exercice 8.3.2

Écrire un shell script démontrant que le shell fils hérite de son père, mais que le père n'hérite pas de son fils.

Exercice 8.3.3

En utilisant exclusivement les commandes *`cd`* et *`echo`*, écrire le shell script "*`recurls`*" réalisant la même fonction que la commande *`ls -R`*. C'est à dire que la commande *`recurls rep1`* devra lister les noms de tous les fichiers et répertoires situés sous le répertoire rep1, y compris les sous-répertoires et les fichiers qu'ils contiennent. Une solution très simple consiste à rendre le script recurls récursif. (un shell script est récursif s'il s'invoque lui-même).

Exercice 8.3.4

Écrivez le script "*`rename`*" permettant de renommer un ensemble de fichiers. Par exemple *`rename '.c' '.bak'`* aura pour effet de renommer tous les fichiers d'extension *`.c`* en *`.bak`*. les fichiers f1.c et f2.c deviennent f1.bak et f2.bak. Utilisez la commande *`basename`*.

Exercice 8.3.5

Et si `vi` gardait une copie de secours ?

`vi` est un peu délicat à maîtriser au début, et si on sort par `:wq` après avoir fait une gaffe, tout est perdu. Ecrivez donc un petit script en Bash qui crée une copie de secours. Appelons le `sv`i, pour Safe VI. La commande `svi fich1` devra invoquer `vi fich1`, mais laisser derrière elle un fichier fich1.bak contenant la version d'origine de fich1.

Pensez aux deux cas suivants :

Si je dis : `svi tralala` et que tralala n'existe pas?

Si je dis : `svi fich1.bak`, que se passe-t-il?

Exercice 8.3.6

L'espace disque est précieux ! Une idée pour économiser : Ecrire un script qui recherche dans toute mon arborescence tous les fichiers qui n'ont pas été accédés depuis un temps T et dont la taille est supérieure a MIN, et les comprimer par l'utilitaire gzip. T et MIN sont des constantes définies au début du script par des valeurs judicieusement choisies. Au fait, a quoi sert MIN ?

Un tel script pourrait être lancé une fois par semaine.

Exercice 8.3.7

Écrire un script dont le nom est process permettant de copier dans un tableau la liste des processus de l'utilisateur exécutant ce script, puis afficher le nom de chaque processus.