

Programmation shell

version 1.0 (2005)

par Marc Gaëtano



Programmation shell

- **Variables et environnement**
- **Structure d'un script shell**
- **Structures de contrôle**
- **Commandes usuelles**



Variables d'environnement

- Les shells permettent à l'utilisateur de définir des *variables*. Celles-ci peuvent être réutilisées dans les commandes shell.
Convention : les noms sont en minuscules
- Vous pouvez aussi définir des *variables d'environnement*: des variables qui sont aussi visibles depuis les scripts ou les exécutable appelés depuis le shell.
Convention : les noms sont en majuscules
- La commande `env`
Affiche toutes les variables d'environnement existantes ainsi que leur valeur



Variables de shell

- Un seul type possible : la chaîne de caractères
- Pas besoin d'être déclarée pour recevoir une valeur par affectation
`une_variable=100`
 - **Attention** : il ne faut **PAS** mettre d'espace entre la variable, le signe = et la valeur
- Utilisation d'une syntaxe particulière pour accéder à la valeur
`echo $une_variable`
- Pour qu'une variable garde sa valeur initiale dans un sous-shell
`export une_variable`



Exemples de variables de shell

Variables de shell

- `projdir=/home/bob/unix/projet`
`ls -la $projdir; cd $projdir`

Variables d'environnement

- `cd $HOME`
- `echo $PWD`
- `export JAVA_SOURCE=$HOME/java/src`

Variables de shell

- `$1, $2, $3, ...`



Variables d'environnement standards

Utilisées par de nombreuses applications

- **LD_LIBRARY_PATH**
Chemin de recherche de bibliothèques partagées
- **DISPLAY**
Écran sur lequel afficher les applications X (graphiques)
- **EDITOR**
Éditeur par défaut (vi, emacs...)
- **HOME**
Répertoire de l'utilisateur courant.
- **HOST**
Nom de la machine locale
- **MANPATH**
Chemin de recherche des pages de manuel.
- **PATH**
Chemin de recherche des commandes
- **PRINTER**
Nom de l'imprimante par défaut
- **SHELL**
Nom du shell courant
- **TERM**
Nom du terminal / mode courant
- **USER**
Nom de l'utilisateur courant



Variables d'environnement PATH

- PATH

Spécifie l'ordre de recherche des commandes pour le shell

```
~/bin:/usr/local/bin:/usr/bin:/bin:/sbin
```

- LD_LIBRARY_PATH

Spécifie l'ordre de recherche pour les bibliothèques partagées (codes binaires partagés par les applications, comme la bibliothèque C) pour ld

```
/usr/local/lib:/usr/lib:/lib:/usr/X11R6/lib
```

- MANPATH

Spécifie l'ordre de recherche pour les pages de manuel

```
/usr/local/man:/usr/share/man
```



Mise en garde sur le PATH

Il est conseillé de ne pas avoir le répertoire “.” dans votre variable d’environnement PATH, en particulier au début de la variable PATH

- Un intrus pourrait placer un fichier `ls` malveillant dans vos répertoires. Il serait exécuté à chaque appel de `ls` depuis ces répertoires et pourrait s’attaquer à vos données personnelles
- Plus généralement, si vous avez un fichier exécutable de nom `xxx` dans un répertoire où `xxx` est aussi le nom d'une commande, il sera utilisé à la place de la commande `xxx` par défaut et certains scripts ne fonctionneront plus correctement
- L'exemple standard : la commande `test`
- L'alternative standard : il suffit de lancez vos propres commandes en utilisant la syntaxe `./mon_script`



Alias

Les shells vous permettent de définir des *alias* (des raccourcis pour des commandes que vous utilisez très souvent)

Exemples

- `alias ls='ls -F'`
Utile pour toujours lancer des commandes avec certains paramètres
- `alias rm='rm -i'`
Utile pour faire que `rm` demande toujours une confirmation
- `alias cherche='find $HOME -name '`
Utile pour remplacer des commandes utilisées régulièrement
- `alias cia='. /home/sydney/env/cia.sh'`
Utile pour initialiser rapidement un environnement
(`.` est une commande shell pour exécuter le contenu d'un script shell)
- En **zsh** les alias sont définis dans le fichier `.zshrc`



Les fichiers de zsh (1)

Des scripts shell sont lus automatiquement à chaque fois qu'un shell zsh est lancé ou terminé

- Il existe toujours deux scripts de nom `zxxx`
 - Le script `/etc/zxxx` :
non modifiable par l'utilisateur et lancé en premier
 - Le script `~/ .zxxx` :
modifiable par l'utilisateur et lancé à la suite
- Permet de personnaliser son environnement
- Les scripts `~/ .zxxx` ne sont pas obligatoires



Les fichiers de zsh (2)

- `/etc/zshenv` et `~/.zshenv`
Scripts shell lus à chaque fois qu'un shell zsh est lancé
- `/etc/zprofile` et `~/.zprofile`
Scripts shell lus si le shell lancé est un shell de login
- `/etc/zshrc` et `~/.zshrc`
Scripts shell lus si le shell lancé est interactif
- `/etc/zlogin` et `~/.zlogin`
Scripts shell lus si le shell lancé est un shell de login
- `/etc/zlogout` et `~/.zlogout`
Scripts shell lus à la sortie d'un shell de login



Les fichiers de zsh (3)

- `.zshenv`
Contient les initialisations des variables d'environnement
- `.zprofile`
Ne doit pas être utilisé avec le `.zlogin` !
- `.zshrc`
Contient les *alias*, les *setopt* et les *fonctions* zsh
- `.zlogin`
Contient le lancement de divers programmes
- `.zlogout`
Contient des commandes spécifiques à la sortie d'un shell login (nettoyage)



Scripts shell

- Langages du shell
 - Un langage de programmation interprété
 - Notion de variables, de paramètres, de structures de contrôles, de fonctions,...
 - Autant de langages que de shells (`sh`, `bash`, `ksh`, `zsh`,...)
- Script shell

Un fichier exécutable contenant des instructions d'un shell donné
- Scripts shell et commandes Linux

Equivalents et traités de la même façon par le système



Contenu d'un script shell

- L'en-tête

Indique le shell à utiliser pour interpréter les commandes placées dans le fichier : `#!/bin/sh` (pour le Bourne Shell)

- Des variables

- déclaration/affectation : `une_variable=123`

- valeur : `$une_variable`

- paramètres : `$1, $2,..., $9`

- Des appels à des commandes ou à des scripts

Toutes les commandes Linux et tous les scripts accessibles

- Des instructions

affectation/référence de variables, instructions conditionnelles et itératives, instructions d'entrée/sortie,...



Variables et paramètres

- **Variables spéciales (liste non exhaustive)**
 - `$0` : le nom du script (du fichier)
 - `$#` : le nombre de paramètres du script
 - `$@` : la liste des paramètres du script
 - `$?` : le code de retour de la dernière commande exécutée
 - `$$` : le numéro de processus du shell courant (PID)
- **Paramètres**
 - Seulement 9 noms de variable `$1`, `$2`, ..., `$9`
 - L'instruction `shift` supprime le premier paramètre et décale les autres vers la gauche
 - L'instruction `shift n` supprime les `n` premiers paramètres et décale les autres vers la gauche



Variables et substitutions

- Substitution de la valeur d'une variable
`echo $HOST`
- Substitution protégée de la valeur d'une variable
`TMP=${USER}_$$` (et non `$USER_$$`)
- Substitution du résultat d'une commande
`nb_files=$(ls | wc -l)` (ou ``ls | wc -l``)
- Autres substitutions

Dans les substitutions suivantes, si la variable `var` a une valeur, alors

`${var-VAL}` substitue `var`, sinon substitue `VAL`

`${var=VAL}` substitue `var`, sinon donne à `var` la valeur `VAL` et substitue `var`

`${var?VAL}` substitue `var`, sinon affiche `VAL` et quitte le shell

`${var+VAL}` substitue `VAL`, sinon ne substitue rien



Variables et lecture de valeurs

La commande `read` permet de lire des chaînes de caractères sur l'entrée standard et de les affecter à des variables

- `read line`
Lit une ligne (une suite de caractères terminée par un *newline*) sur l'entrée standard et l'affecte à la variable `line`
- `read var_1 var_2 ... var_n`
Lit une ligne sur l'entrée standard, la découpe en *mots*, et affecte chaque mot aux variables `var_1`, `var_2`, ..., `var_n`
- S'il y a moins de mots que de variables, les variables de traîne sont initialisées avec la chaîne vide ""
- S'il y a plus de mots que de variables, la dernière variable reçoit comme valeur la chaîne formée des mots de traîne



Variables et affectations

La commande `set` permet d'affecter des chaînes de caractères aux variables spéciales d'un script shell (paramètres) `$1`, `$2`, ...

- `set -- val_1 val_2 ... val_n`
Affecte les valeurs `val_1`, `val_2`, ..., `val_n` aux variables `$1`, `$2`, ..., `$n`
- Mais aussi
 - `set` (sans paramètre)
Affiche toutes les variables positionnées avec leur valeur
 - `set -x`
Affiche les commandes avant qu'elles soient exécutées
 - `set -a [+a]`
Exporte toutes les variables affectées entre les instructions `set -a` et `set +a` (seulement dans un script)



Exemple de script (1)

```
#!/bin/sh  
# début du script  
echo "Hello $USER"  
echo "You are currently on $HOST"  
echo "The content of your homedir:"  
/bin/ls  
# affectation de la variable NBFILS  
NBFILS=$(ls | wc -l)  
echo "Total: $NBFILS elements"
```

welcome.sh



Exemple de script (2)

```
[hal@/home/bob] chmod a+x welcome.sh

[hal@/home/bob] ls -l welcome.sh
-rwxr-xr-x  1 bob bob 201 2005-08-01 17:05 welcome.sh*

[hal@/home/bob] ls -F
Desktop/    Trash/    emacs/    tmp/    welcome.sh*
Mail/       bin/      lib/      unix/

[hal@/home/bob] ./welcome.sh
Hello bob
You are currently on hal
The content of your homedir:
Desktop     Trash      emacs      tmp        welcome.sh*
Mail        bin        lib        unix
Total: 9 elements
```



Exécution d'un script shell

- `. my_script.sh` ou `sh my_script.sh` (extension optionnelle)
Exécute les commandes contenues dans le fichiers `my_script.sh` dans le **shell courant**
- `./my_script` (le fichier `my_script` a les droits `rx`)
Exécute les commandes contenues dans le fichiers `my_script.sh` dans un **sous-shell** du shell courant
- Autres commandes qui s'exécutent dans un **sous-shell** du shell courant
 - Les commandes reliées par des tuyaux
`cat *.log | grep -i erreur | sort`
 - Les commandes groupées entre "(" et ")"
`pwd; ( cd ..; pwd ); pwd`
 - Les structures de contrôles (voir plus loin)
`if, case, for, while et until`



Sortie d'un script shell

- `exit n`

Termine et sort du script shell courant avec le code de retour `n`

- `exit`

Termine et sort du script shell courant. Le code de retour est celui de la dernière commande exécutée dans ce script

- Exemple

```
if [ $# -ne 2 ]; then
    echo "wrong number of arguments: $# "
    exit 1
fi
```

- Valeur du code de retour

0 signifie que le script s'est exécuté correctement

Différent de 0 signifie qu'une erreur est survenue



Structures de contrôle

Identiques à un langage de programmation (comme C ou Java)

- Instructions conditionnelles
 - Les formes `&&` et `||`
Déjà vues !
 - L'instruction `if-then-else-fi`
Sélection à une, deux ou plusieurs alternatives
 - L'instruction `case-esac`
Aiguillages à valeurs multiples
- Instructions itératives (boucles)
 - la boucle `for-done`
Itération bornée
 - les boucles `while-done` et `until-done`
Itérations non bornées



Groupage de commandes

- Séquence de commandes avec le symbole ;
`echo "mes fichiers"; ls; echo "-----"`
- Groupage logique avec les opérateurs `||` (ou) et `&&` (et)
 - `cat ~/infos 2>/dev/null || echo "erreur"`
N'exécute `echo` que si la première commande échoue
 - `ls ~/infos 2>/dev/null && cat ~/infos`
N'affiche le contenu d'infos que si la commande `ls` réussit
- Groupage en sous-shell avec les parenthèses `"( " et ")"`
`pwd; ( cd ../; pwd ); pwd`
Affiche successivement les chemins du répertoire courant, du répertoire père, puis à nouveau du répertoire courant



La commande if (1)

- Sélection à une alternative

```
if    test_0
  then commandes_1
fi
```

- Sélection à deux alternatives

```
if    test_0
  then commandes_1
  else commandes_2
fi
```

- Sélection à plusieurs alternatives

```
if    test_0
  then commandes_0
  elif test_1
  then commandes_1
  ....
  then commandes_n-1
  else commandes_n
fi
```

Le code de retour de `test_0` est interprété comme un booléen :

- code de retour 0 : le booléen est VRAI
- code de retour différent de 0 : le booléen est FAUX



La commande if (2)

- Comparaison de fichiers

```
if cmp $1 $2; then
    echo "les fichiers $1 et $2 sont identiques"
else
    echo "les fichiers $1 et $2 sont différents"
```

- Une version protégée de cat

```
if ls $1 &> /dev/null; then
    cat $1
else
    echo "le fichier $1 est introuvable"
fi
```

- Version équivalente avec les constructeurs && et ||
ls \$1 &> /dev/null && cat \$1 || echo "..."
- La forme if s'utilise surtout avec la forme test



La commande test (1)

Une commande spéciale pour effectuer des tests, très utilisée avec les commandes `if`, `while` et `until`

- Syntaxe standard

`test expression`

- Syntaxe alternative

`[ expression ]`

Attention aux espaces entre "`[`", "`]`" et `expression`

- Produit une valeur de retour (un entier) égale à 0 si le test est VRAI, à 1 sinon
- `expression` est un **prédicat** spécifique qui ne fonctionne qu'avec la construction `test`



La commande test (2)

Prédicats sur les chaînes de caractères

- `c1 = c2` (resp. `!=`) VRAI si les chaînes sont égales
(resp. différentes)
- `-z chaîne` (resp. `-n`) VRAI si la chaîne est vide
(resp. non vide)

Prédicats sur les entiers

Ici `n1` et `n2` sont des chaînes de caractères représentant des entiers

- `n1 -eq n2` (resp. `-ne`) VRAI si les entiers sont égaux
(resp. différents)
- `n1 -gt n2` (resp. `-lt`) VRAI si `n1` est strictement supérieur
(resp. inférieur) à `n2`
- `n1 -ge n2` (resp. `-le`) VRAI si `n1` est supérieur (resp.
inférieur) ou égal à `n2`



La commande test (3)

Prédicats sur les fichiers et répertoires

Pour que les tests suivants soient VRAIS, il faut avant tout que `fichier` existe sur le disque

- `-r fichier` VRAI si `fichier` est lisible (droit r)
- `-w fichier` VRAI si `fichier` est modifiable (droit w)
- `-x fichier` VRAI si `fichier` possède le droit x
- `-f fichier` VRAI si `fichier` n'est pas un répertoire
- `-d fichier` VRAI si `fichier` est un répertoire
- `-s fichier` VRAI si `fichier` a une taille non nulle



La commande test (4)

On peut combiner les prédicats à l'aide de trois opérateurs et de parenthèses

- `! expr` VRAI si `expr` est FAUX
- `expr1 -a expr2` VRAI si `expr1` et `expr2` sont VRAIs
- `expr1 -o expr2` VRAI si `expr1` ou `expr2` est VRAI
- `( expr )` Attention : les parenthèses ont une signification pour le shell, il faut donc les faire précéder du caractère `"\"`

Exemple

```
if [ $x -gt 0 -a \"( $y -ge 1 -a $y -lt $x \" ) ]  
then  
...
```



La commande case (1)

- Sélectionne `val_i`, le premier choix parmi `val_1,..., val_2, val_n`, qui correspond à la valeur de `expr`, et exécute ensuite `commandes_i`

```
case expr in
    val_1) commandes_1;;
    val_2) commandes_2;;
    ....
    val_n) commandes_n;;
esac
```

- `expr` peut être une chaîne de caractères quelconque
- `val_i` peut être une chaîne de caractères quelconque incluant des caractères génériques (wildcards)
- `commandes_i` est une suite de zéro, une ou plusieurs commandes séparées par des ";"



La commande case (2)

- Sélections simples sur une chaîne de caractères

```
case $lang in
    french) echo "Bonjour";;
    english) echo "Hello";;
    spanish) echo "Buenos Dias";;
esac
```

- Sélections avec motifs et expressions régulières

```
case $arg in
    -i | -r) echo "$arg is an option";;
    -*) echo "$arg is a bad option"; exit 1;;
    [0-9]*) echo "$arg is an integer";;
    *.c) echo "$arg is a C file"; gcc -c $arg;;
    *) echo "default value";;
esac
```



La commande for (1)

- Exécute les commandes `commandes` en donnant successivement à la variable `var` les valeurs `val_1,..., val_2, val_n`

```
for var in val_1 val_2 ... val_n
do
    commandes
done
```

- `val_i` peut être une chaîne de caractères quelconque incluant des caractères génériques (wildcards)

```
for file in *; do ... done
```

- En l'absence de `val_i`, la variable `var` prend comme valeurs successives les valeurs des **paramètres** du script :

```
for arg
do echo $arg; done
```



La commande for (2)

- Pour afficher les fichiers exécutables du répertoire courant :

```
for file in *  
do [ -f $file -a -x $file ] && echo $file; done
```

- Pour compiler des fichiers sources C du répertoire courant :

```
for file in prog1.c prog2.c prog3.c  
do gcc -c $file; done
```

- Pour imprimer tous les fichiers sources C du répertoire courant :

```
for file in *.c  
do lpr $file; done
```

- Pour tuer tous les processus de l'utilisateur de nom "emacs" :

```
for proc in $(ps x | grep emacs | cut -d' ' -f2)  
do kill $proc; done
```



La commande seq

- Affiche une séquence de nombres

```
seq i j
```

Si *i* est plus petit ou égal à *j*, affiche les entiers *i*, *i*+1,...,*j*, sinon ne fait rien

```
seq -s char i j
```

Remplace le séparateur (un '\n' par défaut) par le caractère *char*

```
seq i k j
```

Si *i* est plus petit ou égal à *j*, affiche les entiers *i*, *i*+*k*,...,*j*, pour tout les *k* tels que *i*+*k* ne dépasse pas *j*, sinon ne fait rien

- Utilisé la plupart du temps avec l'itération `for`

```
for i in $(seq 1 $max); do  
    mkdir "tmp$i"
```

```
done
```



La commande while

- Exécute les commandes `commandes` tant que le test `un_test` est VRAI

```
while un_test
do
    commandes
done
```

- `un_test` peut être une commande **quelconque** mais le plus souvent c'est un appel à la commande `test` (ou `[]`)

```
while [ $# -gt 0 ]; do
    echo $1; shift
done
```

- La commande `un_test` peut être remplacée par `:` ou `true` qui retournent toujours le code 0 (VRAI)



La commande until

- Exécute les commandes `commandes` tant que le test `un_test` est FAUX

```
until un_test
do
    commandes
done
```

- Equivalente à une commande `while` en prenant la négation du test

```
while [ $# -gt 0 ]; do      until [ $# -eq 0 ]; do
    echo $1; shift          echo $1; shift
done                        done
```

- La commande `un_test` peut être remplacée par `false` qui retourne toujours le code 1 (FAUX)



La commande break

La commande `break` permet de sortir d'une boucle sans terminer l'itération en cours et sans passer par le test

```
while true
do
    echo -n "list the current dir? (y/n/q) "
    read yn

    case $yn in
        [yY] ) ls -l . ; break;;
        [nN] ) echo "skipping" ; break;;
        q ) exit ;;
        * ) echo "$yn: unknown response";;
    esac
done
```



La commande continue

La commande `continue` permet de passer directement à l'itération suivante sans nécessairement terminer l'itération en cours

```
for f in *
do
    [ -f $f ] || continue
    [ -r $f ] || continue
    echo "printing file $f:"
    case $f in
        *.ps ) lpr $f;;
        *.txt) a2ps $f;;
        *) echo "don't know how to print $f"
    esac
    echo "done"
done
```



Structures de contrôle et redirections

- L'évaluation d'une structure `if`, `case`, `for`, `while` ou `until` s'effectue dans un sous-shell

- On peut rediriger l'entrée et/ou la sortie d'un tel sous-shell

```
for file in *.c; do  
    gcc -c $file  
done 2> compil.error
```

- On peut lier un tel sous-shell avec un tuyau

```
ls ~dan/images/*.gif | while read file; do  
    [ -f $(basename $file) ] || cp $file .; done
```

- On peut exécuter un tel sous-shell en arrière plan

```
for file in *.c; do gcc -c $file; done &
```



La commande expr (1)

Une commande pour effectuer des calculs arithmétiques sur les entiers

- Utilisation

```
expr 2 + 3  
incr=$(expr $incr + 1)
```

- Erreurs fréquentes

```
expr 2 +3          : +3 est vu comme un unique argument  
=> expr 2 + 3  
expr 2 * 3         : * est un méta-caractère du shell  
=> expr 2 \* 3  
expr 2 * ( 3 + 5 ) : les parenthèses '(' et ')' sont  
                   : interprétées par le shell  
=> expr 2 \* \( 3 + 5 \)
```



La commande expr (2)

Une commande pour effectuer des tests ou des calculs sur les chaînes de caractères

- `expr match chaine modèle` (ou `expr chaine : modèle`)
Teste si chaine correspond à modèle :

```
expr match linux 'li*'
```

- `expr substr chaine i n`
Affiche la sous-chaîne de chaine de longueur n et commençant au caractère à l'indice i (le premier caractère est à l'indice 1)
- `expr index chaine caractère`
Affiche l'indice de la première occurrence de caractère dans chaine, ou 0 si caractère n'est pas présent dans chaine
- `expr length chaine`
Affiche la longueur de chaine



La commande eval

Une commande pour évaluer la commande qui lui est passée en paramètre

- **eval commande**

Evalue commande comme si elle avait été directement interprétée par le shell courant :

- le shell applique le découpage en sous-commandes, les substitutions de variables et de commandes et la génération de noms de fichiers à commande
- le shell applique le découpage en sous-commandes, les substitutions de variables et de commandes et la génération de noms de fichiers au résultat précédent et finalement interprète le tout
- Affectation avec calcul de la variable affectée

```
x=y; eval $x=123; echo $y (y a la valeur "123")
```



La commande exec

Une commande à deux usages

- `exec` commande

Le shell courant exécute la commande `commande` qui **devient** le shell courant

- aucun intérêt dans un shell interactif (ne le testez pas)
- à ne pas confondre avec l'option `-exec` de la commande `find`

- `exec` et redirections

- `exec < file, exec > file, exec 2> file`

Redirigent l'entrée, la sortie ou les erreurs du **shell courant** vers le fichier `file`

- `exec 3<&0`

Affecte l'entrée standard au descripteur de fichier 3



La commande getopts

Une commande pour traiter plus simplement les arguments d'un script

```
while getopts f:ri opt; do
    case $opt in
        r) echo "r present";;
        i) echo "i present";;
        f) echo "f present with argument $OPTARG";;
    esac
done

shift $(expr $OPTIND - 1)
echo "command arguments:"
for arg in $@; do
    echo " $arg"
done
```



Les fonctions (1)

- Syntaxe

Pour définir la fonction de nom `fonction` :

```
fonction () {  
    commandes  
}
```

- Même mécanisme de passage de paramètre qu'un script shell
- Peut contenir des appels à des commandes quelconques, à d'autres fonctions ou à elle-même (fonction récursive)
- L'instruction `return n` permet de sortir d'une fonction avec le code de retour `n`
- Attention : l'évaluation de la commande `exit` depuis l'intérieur d'une fonction termine le **script shell englobant**



Les fonctions (2)

- Une fonction pour les erreurs

```
usage() {  
    echo "usage: $(basename $0) FILE"2>/dev/stderr  
    exit 1  
}
```

- Une fonction pour confirmer une saisie avec la question en paramètre

```
confirm () {  
    echo -n $1  
    while read ANSWER; do  
        case $ANSWER in  
            y|Y) return 0;;  
            n|N) return 1;;  
            *) echo -n "please answer y or n: ";;  
        esac  
    done }
```

