Sommaire

Introduction

Objectif

Être capable d'écrire des *shell* scripts simples.

Rappels

Les langages interprétés

Du point de vue du programmeur, les avantages et inconvénients d'un langage interprété peuvent être résumées ainsi :

- Le code source fraîchement écrit peut être testé immédiatement ; il suffit de lancer l'exécution en fournissant le nouveau fichier source à l'interpréteur.
- Pour que le fichier source puisse être utilisé, il faut que l'interpréteur soit également présent. La portabilité est assurée sur une machine différente si le même interpréteur y est disponible.
- Le fichier à transporter est un code source de type texte, généralement petit (quelques kilo-octets). En revanche, la mémoire nécessaire pour l'exécution est plus importante que dans le cas d'un fichier compilé, car l'interpréteur doit aussi y être présent.
- L'exécution du programme nécessite que l'interpréteur analyse chaque instruction, et la traduise. Le processus est donc plus lent.
- Le fichier exécutable est naturellement lisible et modifiable par tout utilisateur. Si le programme a une bonne documentation interne, il se suffit parfaitement. C'est un gros avantage dans l'esprit des logiciels libres.

Remarque : certains interpréteurs modernes (Perl, Python, PHP, etc.) procèdent à une phase de compilation avant de lancer le programme pour permettre une optimisation et donc des gains en rapidité. De toute manière, l'utilisateur/programmeur ne verra pas de phase intermédiaire entre la rédaction du code et son exécution.

Les langages de scripts les plus populaires sont : Perl, Tcl, Ruby, Python, PHP et bien évidemment les shells UNIX.

Automatisation de tâches

Les administrateurs (ou parfois les programmeurs) ont souvent le besoin d'automatiser des traitements (surveillance, sauvegarde, maintenance, installation, ...). Pour cela, ils peuvent :

- faire des groupement de commandes à partir d'un shell : fréquent
- créer des scripts (avec le shell ou avec un autre langage de script comme Perl, Python, PHP, etc ...) : très fréquent
- créer des programmes utilisant l'API de l'OS : moins fréquent

L'automatisation des tâches est un domaine dans lequel les scripts shell prennent toute leur importance. Il peut s'agir de préparer des travaux que l'on voudra exécuter ultérieurement grâce à un système de programmation horaire (*crontab*) mais on utilise aussi les scripts pour simplifier l'utilisation de logiciels complexes ou écrire de véritables petites applications.

Les scripts sont aussi très utiles lorsqu'il s'agit de faire coopérer plusieurs utilitaires système pour réaliser une tâche complète. On peut ainsi écrire un script qui parcoure le disque à la recherche des fichiers modifiés depuis moins d'une semaine, en stocke le nom dans un fichier, puis prépare une archive tar contenant les données modifiées, les sauvegarde sur une bande, puis envoie un e-mail à l'administrateur pour rendre compte de son action. Ce genre de programme est couramment employé pour automatiser les tâches administratives répétitives.

Les shells proposent un véritable langage de programmation, comprenant toutes les structures de contrôle, les tests et les opérateurs arithmétiques nécessaires pour réaliser de petites applications. En revanche, il faut être conscient que les shells n'offrent qu'une bibliothèque de routines internes très limitée. Nous ferons alors fréquemment appel à des utilitaires système externes.

En résumé, les scripts sont utilisés surtout pour :

- automatiser les tâches administratives répétitives
- simplifier l'utilisation de logiciels complexes
- mémoriser la syntaxe de commandes à options nombreuses et utiliser rarement
- réaliser des traitements simples

Les shell scripts

Un *shell* script est :

- un fichier texte ASCII (on utilisera un éditeur de texte)
- exécutable par l'OS (le droit x sous Unix/Linux)
- interprété par un shell (/bin/bash par exemple)

Remarque : on a l'habitude de mettre l'extension .sh au fichier script.

Exemple d'un shell script :

```
$ ls -l script.sh
-rwxr-xr-x 1 tv tv 38 2010-08-05 11:22 script.sh

$ cat script.sh
#!/bin/bash
echo "je suis un script"
```

Remarque : un script doit commencer par une ligne "shebang" qui contient les caractères '#!' (les deux premiers caractères du fichier) suivi par l'exécutable de l'interpréteur. Cette première ligne permet d'indiquer le chemin (de préférence en absolu) de l'interpréteur utilisé pour exécuter ce script.

Avec un script, il est possible :

- d'exécuter et/ou grouper des commandes
- d'utiliser des variables prédéfinies ($?, ...), des variables d'environnement ($HOME, $USER, ...)
- de gérer ses propres variables
- de faire des traitements conditionnels (if, case) et/ou itératifs (while, for)

Les Bases

Les commentaires

Les commentaires sont introduits par le caractère '#'. On peut aussi utiliser l'instruction nulle ':'.

L'affichage sur la sortie standard

Pour afficher du texte, on utilisera echo ou `printf` :

```
#!/bin/bash
CHAINE="world"

echo -n "Un message : hello $CHAINE"
echo -e "\tUn message : hello $CHAINE"

echo 'hello $CHAINE'
echo "hello \$CHAINE"

printf "Un message : hello %s\n" $CHAINE
```

La saisie de données

Pour réaliser la saisie sur le périphérique d'entrée (généralement le clavier), on utilisera read :

```
#!/bin/sh
# Saisie du nom.
echo -n "Entrez votre nom: "
read nom
# Affichage du nom.
echo "Votre nom est $nom."
exit 0
```

Les variables

Une variable existe dès qu'on lui attribue une valeur. Une chaîne vide est une valeur valide. Une fois qu'une variable existe, elle ne peut être détruite qu'en utilisant la commande interne `unset`.

Une variable peut recevoir une valeur par une affectation de la forme :

```
nom=[valeur]
```

Si aucune valeur n'est indiquée, la variable reçoit une chaîne vide.

Le caractère '$' permet d'introduire le remplacement des variables. Le nom de la variable peut être encadré par des accolades, afin d'éviter que les caractères suivants ne soient considérés comme appartenant au nom de la variable.

Créer une variable :

```
$ chaine=bonjour
```

Afficher une variable :

```
$ echo $chaine
$ echo ${chaine}
$ echo $HOME
```

Remarques : Il n'existe pas de typage fort dans le shell et, par défaut, TOUT EST UNE CHAINE de caractères. Il existe une différence d'interprétation par le shell des chaînes de caractères. Les protections (quoting) permettent de modifier le comportement de l'interpréteur. Il y a trois mécanismes de protection : le caractère d'échappement, les apostrophes (quote) et les guillemets (double-quote) :

- *Le caractère \ (backslash) représente le caractère d'échappement. Il préserve la valeur littérale du caractère qui le suit, à l'exception du <retour-chariot>.*
- *Encadrer des caractères entre des apostrophes simples (quote) préserve la valeur littérale de chacun des caractères. Une apostrophe ne peut pas être placée entre deux apostrophes, même si elle est précédée d'un backslash.*
- *Encadrer des caractères entre des guillemets (double-quote) préserve la valeur littérale de chacun des caractères sauf $, `, et \. Les caractères $ et ` conservent leurs significations spéciales, même entre guillemets. Le backslash ne conserve sa signification que lorsqu'il est suivi par $, `, ", \, ou <fin-de-ligne>. Un guillemet peut être protégé entre deux guillemets, à condition de le faire précéder par un backslash.*

Substitutions de variables

Les substitutions de paramètres (ou expressions de variables) sont décrites ci-dessous :

```
$nom La valeur de la variable.
${nom} Idem.
${#nom} Le nombre de caractères de la variable.
${nom:-mot} Mot si nom est nulle ou renvoie la variable.
${nom:=mot} Affecte mot à la variable si elle est nulle et
renvoie la variable.
${nom:?mot} Affiche mot et réalise un exit si la variable est
non définie.
${nom:+mot} Mot si non nulle.
${nom#modèle} Supprime le petit modèle à gauche.
${nom##modèle} Supprime le grand modèle à gauche.
${nom%modèle} Supprime le petit modèle à droite.
${nom%%modèle} Supprime le grand modèle à droite.
```

Les formes les plus utilisées sont :

Obtenir la longueur d'une variable :

```
$ PASSWORD="secret"
$ echo ${#PASSWORD}
```

Fixer la valeur par défaut d'une variable nulle :

```
$ echo ${nom_utilisateur:=`whoami`}
```

Utiliser des tableaux :

```
$ tableau[1]=tv
$ echo ${tableau[1]}
```

Supprimer une partie d'une variable :

```
$ fichier=texte.htm
$ echo ${fichier%htm}html
$ echo ${fichier##*.}
```

Substitution de commandes

La substitution de commandes permet de remplacer le nom d'une commande par son résultat. Il en existe deux formes :

```
$(commande) ou `commande`
```

Cette syntaxe effectue la substitution en exécutant la commande et en la remplaçant par sa sortie standard, dont les derniers sauts de lignes sont supprimés.

Déterminer le nombre de fichiers présents dans le répertoire courant :

```
$ NB=$(ls | wc -l)
$ echo $NB
```

Récupérer le chemin d'accès :

```
$ CHEMIN=`dirname /un/long/chemin/vers/toto.txt`
$ echo $CHEMIN
/un/long/chemin/vers
```

Récupérer le nom du fichier :

```
$ NOM_FICHIER=`basename /un/long/chemin/vers/toto.txt`
$ echo $NOM_FICHIER
toto.txt
```

Afficher le chemin absolu de son répertoire personnel :

```
$ getent passwd | grep $(whoami) | cut -d: -f6
$ echo $HOME
```

Évaluation Arithmétique

L'évaluation arithmétique permet de remplacer une expression par le résultat de son évaluation. Le format d'évaluation arithmétique est :

```
$((expression))
```

Calculer la somme de deux entiers :

```
$ somme=$((2+2))
$ echo $somme
4
```

Pour manipuler des expressions arithmétiques, on pourra utiliser aussi :

```
=> la commande expr :
$ expr 2 + 3

=> la commande interne let :
$ let res=2+3
$ echo $res

=> l'expansion arithmétique ((...)) (syntaxe style C) :
$ (( res = 2 + 2 ))
$ echo $res

=> la calculatrice bc (notamment pour des calculs complexes ou
sur des réels)
$ echo "scale=2; 2500/1000" | bc -lq
2.50
$ VAL=1.3
$ echo "scale=2; ${VAL}+2.5" | bc -lq
3.8
```

Séparation des mots

Les résultats du remplacement des variables, de la substitution de commandes, et de l'évaluation arithmétique, qui ne se trouvent pas entre guillemets sont analysés par le shell afin d'appliquer le découpage des mots.

L'interpréteur considère chaque caractère de la variable IFS comme un délimiteur, et redécoupe le résultat des transformations précédentes en fonction de ceux-ci. Si la valeur du paramètre IFS est exactement <espace><tabulation><retour-chariot>, (la valeur par défaut), alors toute séquence de caractères IFS sert à délimiter les mots.

```
$ set | grep IFS | head -1
IFS=$' \t\n'
```

Portée d'une variable du shell

Par défaut, les variables sont **locales** au shell.Les variables n'existent donc que pour le shell courant. Si on veut les rendre accessible aux autres shells, il faut les exporter avec la commande export (pour bash), ou utiliser la commande setenv (csh).

Il est possible de modifier ce comportement avec les commandes suivantes :

```
export variable          Marque la variable pour qu'elle soit
exportée.
export -p             Affiche les éléments exportés.
export -n variable    Supprime l'attribut exportable de la
variable.
export -f fonction    Marque une fonction pour être exportée.
declare -x variable   Identique à 'export variable'.
declare               Identique à 'export -p'.
declare +x variable   Identique à 'export -n variable'.
declare -f fonction   Identique à 'export -f fonction
```

Les variables internes du shell

C'est variables sont très utilisés dans la programmation des scripts :

```
$0            : Nom du script ou de la commande
$1,$2, ...    : Paramètres du shell ou du script
$*            : Tous les paramètres
$@            : Idem (mais "$@" eq. à "$1" "$2"...)
$#            : Nombre de paramètres
$-            : Options du shell
$?            : Code retour de la dernière commande
$$            : PID du shell
$!            : PID du dernier processus shell lancé en
arrière-plan
$_            : Le dernier argument de la commande précédente.
Cette variable est également mise dans l’environnement de chaque
commande exécutée et elle contient le chemin complet de la
commande.
```

Comme n'importe quel programme, il est possible de passer des paramètres (arguments) à un script. Les arguments sont séparés par un espace (ou une tabulation) et récupérés dans les variables internes $0, $1, $2 etc ... (voir le man bash et la commande shift).

Afficher quelques variables internes :

```
#!/bin/bash
echo "Ce sript se nomme : $0"
echo "Il a reçu $# paramètre(s)"
echo "Les paramètres sont : $@"
echo
echo "Le PID du shell est $$"

$ ./variablesInternes.sh
Ce sript se nomme : ./variablesInternes.sh
Il a reçu 0 paramètre(s)
Les paramètres sont :

Le PID du shell est 8807

$ ./variablesInternes.sh le petit chat est mort
Ce sript se nomme : ./variablesInternes.sh
Il a reçu 5 paramètre(s)
Les paramètres sont : le petit chat est mort

Le PID du shell est 8078
```

Gérer les arguments reçus par un script :

```
#!/bin/bash
echo "nb d'arguments = $#"
echo "nom du script \$0 : $0"
test -n "$1" && echo "\$1 = $1" || exit 1
test -n "$2" && echo "\$2 = $2"
test -n "$1" && echo "les arguments = $@"
exit 0
```

Tester le script :

```
$ ./testArg.sh bts 2010
```

Code de retour

Au niveau du shell, une commande qui se termine avec un code de retour (variable interne $?) nul est considérée comme réussie. Le zéro indique le succès. Un code de retour non-nul indique un échec.

```
$ ls
$ echo $?
0

$ ls fff
$ echo $?
2
```

Les codes de retour sont très souvent tester dans un script.

Les tests et conditions

Les tests peuvent être lancé par la commande interne 'test' qui prend en argument les conditions, et renvoie '0' si le test est vrai, et '1' sinon.

La forme la plus courante est l'utilisation de crochets ([ou [[depuis la version 2) qui encadrent le test.

Pour connaître la syntaxe des tests sur les fichiers, les chaînes de caractère, les valeurs et les associations, faire :

```
help test
help [[
```

Remarque : Chaque élément du test, et les crochets, doivent être bien délimités par au moins un espace. C'est une erreur courante que d'oublier les espaces.

Les structures conditionnelles

La structure if-then-else

Il existe plusieurs formes syntaxiques autorisées :

Cas 1 :

```
if liste de commandes
then  liste de commandes
fi
```

Cas 2 :

```
if liste de commandes ;then liste de commandes ;fi
```

Cas 3 :

```
if liste de commandes
then  liste de commandes
else  liste de commandes
fi
```

Cas 4 :

```
if liste de commandes
then  liste de commandes
elif  liste de commandes
then  liste de commandes
else  liste de commandes
fi
```

Exemples :

SI le répertoire n'existe pas

ALORS

    on le crée

FIN

```
#SHELL script: if1.sh <nom_repertoire>
if [ ! -d "$1" ]
then
   mkdir "$1"
fi
```

```
#SHELL script : if2.sh
#depuis la version 2, on peut utiliser la syntaxe étendue [[
# [[ est un mot clé, pas une commande
if [[ $# != 1 ]]  #ou: if [ $# -ne 1 ]
then echo "Usage: $(basename $0) <nom_fichier>"; exit 1
fi
# la commande test ou [
if [ -e $1 ]
then echo "Le fichier $1 existe"
else echo "Le fichier $1 n'existe pas"
fi

#SHELL script : if3.sh
#on peut utiliser directement des commandes dans le if
if cmp a b &> /dev/null
then echo "Les fichiers a et b sont identiques."
else echo "Les fichiers a et b sont différents."
fi
```

Les choix multiples case - select

- Contrôle conditionnel à cas : les comparaisons se font au niveau des chaînes de caractères. La structure est alors :

```
case "$variable" in
"motif1" ) action1 ;;
"motif2" ) action2 ;;
"motif3a" | "motif3b" ) action3 ;;
...
* ) action_defaut ;;
esac
```

Exemples :

```
#SHELL script : case1.sh
case $1 in
1|3|5|7|9) echo "chiffre impair";;
2|4|6|8)   echo "chiffre pair";;
0) echo "zéro";;
*) echo "c'est un chiffre qu'il me faut !!!";;
esac

#SHELL script : case2.sh
echo "Des vacances ?"
read reponse
case $reponse in
[yYoO]) echo "fainéant !";;
[nN])   echo "alors pas de vacances !";;
*) echo "erreur saisie invalide";;
esac
```

- La construction select, adoptée du Korn Shell, est souvent utilisée pour construire des menus :

```
select nom [ in mot ] ; do liste ; done

select variable [in liste]
do
commande ...
break
done
```

La liste de mots à la suite de in est développée, créant une liste d'éléments. Le symbole d'accueil PS3 est affiché, et une ligne est lue depuis l'entrée standard.

Si la ligne est constituée d'un nombre correspondant à l'un des mots affichés, la variable nom est remplie avec ce mot. Si la ligne est vide, les mots et le symbole d'accueil sont affichés à nouveau. Si une fin de fichier (EOF) est lue, la commande se termine. Pour toutes les autres valeurs, la variable nom est vidée. La ligne lue est stockée dans la variable REPLY.

La liste est exécutée après chaque sélection, jusqu'à ce qu'une commande break ou return soit atteinte.

Exemple :

```
#Affiche l'invite.
PS3='Menu: '
select choix in "création" "modification" "suppression"
"quitter"
do
echo "Votre choix est $choix"
break  #  il faut savoir s'arrêter !
done
```

Les contrôles itératifs (les boucles for – while – until)

Il existe de nombreuses utilisations des boucles for, while et until.

La boucle for

Cas 1 : Autant de tours de boucle que d'éléments dans la liste et variable prenant successivement chaque valeur.

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

Cas 2 : Autant de tours de boucle que de fichiers dans le répertoire courant et variable prenant successivement chaque nom de fichier (non caché).

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

Cas 3 : Autant de tours de boucle que d'arguments dans $ et variable prenant successivement $1 $2 etc... .*

```
for   variable
do liste de commandes
done
```

Exemples :

```
#SHELL script : for1a.sh
for i in 1 2 3 4 5 6 7 8 9 10
do
echo "\$i=$i"
done

#SHELL script : for2.sh
for fic in *
do
if [[ -f $fic ]]
then echo "$fic est un fichier"
elif [[ -d $fic ]]
then echo "$fic est un répertoire"
fi
done

#SHELL script : for1b.sh
#Syntaxe standard améliorée en utilisant la commande seq
START=1
LIMITE=10
SEQUENCE=$(seq -s ' ' $START $LIMITE)
for i in $SEQUENCE
do
  echo -n "$i "
done
```

```
#Idem (style C)
#Double parenthèses, et "LIMITE" sans "$".
for ((i=START; i <= LIMITE; i++))
do
  echo -n "$i "
done
#Encore mieux : la virgule chaîne les opérations.
for ((i=START, j=LIMITE; i <= LIMITE; i++, j--))
do
  echo -n "$i-$j "
done

#SHELL script : for3.sh
#Autres boucles for simples
for PLANETE in Mercure Vénus Terre Mars Jupiter Saturne Uranus
Neptune Pluton
do
  echo $PLANETE
done

#Possibilité de substitution de commande
EXT="sh"
for ligne in $( find . -type f -name "*.$EXT" | sort )
do
  echo "$ligne"
done

# Si la 'liste' est manquante, la boucle opère sur '$@'
for arg
do
 echo -n "$arg "
done
```

La boucle while

Cas 1 : Tant que la condition est vraie.

```
while condition
do liste de commandes
done
```

Cas 2 : Boucle infinie dans laquelle il faudra prévoir la sortie par exit, return ou break.

```
while true        ou while :
do liste de commandes
done
```

Exemples :

```
#!/bin/bash
#SHELL script : while.sh
#Compter jusqu'à 10 dans une boucle "while".
LIMITE=10
a=1
while [ "$a" -le $LIMITE ]
do
  echo -n "$a "
  let "a+=1"
done

#Idem : syntaxe C
((a = 1))      # a=1
while (( a <= LIMITE ))
do
  echo -n "$a "
  ((a += 1))   # let "a+=1"
done
```

La boucle until

Cas 1 : Jusqu'à ce que la condition soit vraie.

```
until liste de commandes
do liste de commandes
done
```

Cas 2 : Boucle sans fin (idem while true).

```
until false
do liste de commandes
done
```

Exemples :

```
#SHELL script : until.sh
ctr=0
until (( ctr >= 10 ))
do
((ctr+=1))
echo ”tour numéro $ctr”
done
```

Remarques : break, continue

Les commandes de contrôle de boucle break et continue correspondent exactement à leur contre partie dans d'autres langages de programmation. La commande break termine la boucle (en sort), alors que continue fait un saut à la prochaine itération de la boucle, oubliant les commandes restantes dans ce cycle particulier de la boucle.

La commande break peut de façon optionnelle prendre un paramètre. Un simple break termine seulement la boucle interne où elle est incluse mais un break N sortira de N niveaux de boucle.

Les fonctions

Les fonctions permettent l'appel de commandes dans l'environnement courant (partage des variables du script père).

Les propriétés des fonctions sont :

- passage de paramètres possibles ($0 $1 $2 ...) ;
- variables locales possibles ;
- rapidité car la fonction est lue à la déclaration et non à l'exécution.
- retourner une valeur de retour (return)

Exemples :

```
#!/bin/bash
#SHELL script : rootCheck.sh
CLEARSCR=clear

function root_check()
{
   # seul root peut executer ce script :)
   id | grep "uid=0(root)" > /dev/null  2>&1
   if [ $? != "0" ]
   then return 1
   else return 0
   fi
}

function say_hello()
{
   if [[ $1 == "true" ]]
   then
      $CLEARSCR
   fi
   echo -e "\x1B[49;34;1m
============================================================
Copyleft (C) 2010 <tv>
============================================================\x1B[0m\n
"
}

# main :
say_hello true

say_hello false
```

```
root_check
if [ $? = "1" ]
then
   echo -e "\x1B[49;31;1mERREUR: ce script ne peut etre execute
que sous le compte root !\x1B[0m"
   echo "";
   exit 1;
fi

#...
exit 0


#SHELL script : fonction.sh
function maFonction
{
   echo "<début de fonction>"
   echo "j'ai reçu $# arguments : \$1=$1 \$2=$2"
   echo "je connais les variables du script père : \$a=$a et \
$b=$b"
   echo "il est possible de les modifier"
   typeset a="nouveau -a"
   b="nouveau -b"
   echo "a=$a b=$b"
   echo "<fin de fonction>"
   return 0
}

#deux variables :
a="ancien -a"
b="ancien -b"
echo "-> a=$a b=$b"

maFonction AZERTY 12345       #Appel avec 2 arguments
echo "-> code retour=$? a=$a b=$b"

unset -f maFonction
maFonction              #Appel (erreur elle n'existe plus !)
```

Code de sortie des scripts

Il faut utiliser des codes de sortie d'une façon systématique et significative.

Exemple :

```
E_MAUVAIS_ARGUMENT=65
...
...
exit $E_MAUVAIS_ARGUMENT
```

On utilisera donc la commande `exit x` pour terminer un script en retournant un code de sortie x au *shell* (x est un nombre entier décimal compris entre 0 et 255)

La convention pour la valeur du code de sortie est la suivante :

- on renvoie un 0 en cas de succès
- on renvoie une valeur différente de zéro en cas d'erreur. Dans ce cas, la valeur de retour peut préciser le type d'erreur (argument manquant, fichier inexistant, ...)

Code sortie ayant une signification particulière (codes réservés)

Code de sortie	**Signification**	**Commentaire**
1	Standard pour les erreurs générales	erreurs diverses (comme une division par 0)
2	mauvaise utilisation de commandes intégrées, d'après la documentation de *bash*	rare
126	la commande appelée ne peut s'exécuter	problème de droits ou commande non exécutable
127	commande introuvable	erreur de frappe, voir aussi **`$PATH`**
128	argument invalide pour **`exit`**	**`exit`** prend seulement des arguments de type entier compris entre 0 et 255
128+n	le script s'est terminé par réception du signal n	voir **`kill -l`** (liste des signaux)
130	script terminé avec Control-C (130=128 + 2)	Control-C est le signal 2 (SIGINT)

Remarque : Il y a eu un essai de normalisation des codes de sortie (voir /usr/include/sysexits.h) pour les programme C et C++. Un standard similaire pour la programmation de script pourrait être approprié.

Convention conseillée : Il est conseillé de restreindre les codes de sortie définis par l'utilisateur à l'intervalle 64 à 113 (en plus du code 0, en cas de succès) pour se conformer au standard C/C++. Ceci permet d'utiliser 50 codes valides et facilitera le débogage des scripts (utiliser un code prédéfini entraînerait une confusion).

Nettoyage

Il faut sortir proprement d'un script, c'est-à-dire :

- en libérant les ressources utilisées (le plus souvent des fichiers temporaires)
- en prévoyant les sorties brutales (par exemple un Ctrl-C ou une erreur)

La commande interne `trap` permet de déclencher une action (commandes, appel de fonction) sur réception de signaux. Les signaux intéressants à intercepter sont : 0 (EXIT), 2 (INT), 3 (QUIT), 15 (TERM), ... Le signal intercepté n'a plus son action d'origine mais celle précisée par `trap`. La liste des signaux gérés par `trap` sont obtenus avec :

```
$ trap -l

$ kill -l
```

Un signal est un simple message (un événement) envoyé au processus, soit par le noyau soit par un autre processus. La commande `kill` permet d'envoyer un signal à un processus. L'utilisation des signaux est un des moyens pour faire communiquer plusieurs processus ensemble.

Pour interrompre l'exécution d'un processus, il suffit de lui envoyer le signal 2 (SIGINT) : en tapant Ctrl-C ou kill -2 pid (pid étant le numéro du processus). La commande ps lx permet de lister les processus de la machine pour un utilisateur.

Exemple 1 :

```
# un nom de fichier temporaire unique :
# nom du script + identifiant du processus
FICHIER_TEMP="/tmp/$(basename $0).$$"
# voir aussi : man mktemp

# Code erreur de sortie = 130 pour Ctrl-c
((E_INT=128+2))

# Nettoie le fichier temporaire si le script est interrompu avec
Ctrl-C
trap "rm -f $FICHIER_TEMP; echo 'Interruption'; exit $E_INT" INT
# ou: trap "rm -f $FICHIER_TEMP; echo 'Interruption'; exit
$E_INT" 2
```

```
# crée le fichier temporaire
touch $FICHIER_TEMP

# simulons un traitement qui nous laisse le temps de faire un
Ctrl-c
while true
do
   sleep 10000
done
```

Il est conseillé de prévoir une fonction de sortie et de traiter toutes les situations.

Exemple 2 :

```
nettoyer_fichiers()
{
   # rm -f ...
}

arreter()
{
   nettoyer_fichiers
   echo "Script annulé !"
   exit 1
}

quitter()
{
   nettoyer_fichiers
   echo "Script terminé"
   exit 0
}

trap arreter 1 2 3 15
trap quitter 0    #ou: trap quitter EXIT
```

Tracer une variable

On peut aussi utiliser la commande `trap` pour tracer une variable lors de l'exécution d'un script. Si on utilise l'argument DEBUG, la commande `trap` exécutera son action après chaque commande simple du script (une affectation de variable, une opération arithmétique, ... ou tout simplement une commande).

Exemple :

```
#!/bin/bash
# Affiche la valeur de $variable après chaque commande.
trap 'echo "<TRACE> \$variable = \"$variable\""' DEBUG

variable=29
let "variable *= 3"
((variable++))
echo "Fin"
exit 0

On obtient alors l'affichage suivant:
<TRACE> $variable = ""
<TRACE> $variable = "29"
<TRACE> $variable = "87"
<TRACE> $variable = "88"
Fin
<TRACE> $variable = "88"
```

Annexe 1 : Règles De Codage

Nom de fichier

Le nom du fichier doit refléter la fonction d'usage réalisé par le script.

Il est composé d'un nom simple ou d'une expression nominale dont les noms sont reliés par des _.

Extension

L'extension doit :

- indiquer que le fichier est un script (par défaut, utiliser l'extension .sh)
- préciser l'interpréteur de commandes utilisé par le script (.sh pour un shell sh ou bash, .ksh pour un kornshell, .csh pour un C-shell, etc ...)

Dépendance

La première ligne d'un script doit préciser l'interpréteur de commandes (et son chemin absolu complet).

Exemple :

```
#!/bin/bash
```

Entêtes descriptives

Le script :

```
#!/bin/bash
#*********************************************#
# Nom:     <nom>.sh                           #
# Auteur:  djumbo <djumbo@nowhere.com>        #
# Date:    05/05/2004                         #
# Version: 1.0                                #
#                                             #
# Rôle:    supprime les fichiers temporaires  #
# Usage:   <nom>.sh [dossier]                 #
# Limites: <limites d'utilisation>            #
# Contraintes:                                #
#*********************************************#
```

Les fonctions :

```
#-------------------------------------------------------#
# Nom: nettoyer_fichiers()                              #
# Description: supprime tous les fichiers dans le       #
#              répertoire indiqué                       #
# Paramètres:                                           #
# Renvoie: 0 en cas de succès, sinon 1                  #
# Variables globales modifiées: aucune                  #
#-------------------------------------------------------#
```

Les identificateurs

Un identificateur est un nom de donnée (constante ou variable) ou de fonction. Un identificateur doit :

- être uniquement composé de lettres, de chiffres et de soulignés (_)
- se distinguer par au moins deux caractères
- ne pas être un mot clé du langage (for, while, ...)

Identificateur de variable

Un identificateur de variable est un **nom** principal suffisamment éloquent, éventuellement complété par une caractéristique d'usage, un qualificatif ou d'autres noms.

Les identificateurs de variable s'écrivent en **minuscules** en mettant en majuscule la première lettre d'un nouveau mot.

Exemples :

```
distance, distanceMax, nombreDeFichiers, codeRetour
```

Remarques : Les lettres i,j,k utilisées seules sont usuellement admises pour les indices de boucle

Certaines abréviations sont admises quand elles sont d'usage courant : max, min, prec, suiv, ...

Identificateur de constante

L'identificateur d'une constante obéit aux mêmes règles sémantiques qu'un identificateur de variable.

L'identificateur d'une constante s'écrit en **majuscules** en séparant chaque mot par un souligné (_)

Exemples :

```
DISTANCE_MAX=50, ERREUR_FICHIER=2
```

Identificateur de fonction

Un identificateur de fonction est construit à l'aide d'un **verbe** (à infinitif ou au présent de l'indicatif représentant l'action réalisée par la fonction) et éventuellement d'éléments supplémentaires.

Exemples :

```
ajouter(), estPresent(), estVide(), sauverValeur(), …
```

Annexe 2 : Règles De Programmation

La syntaxe de ce type de langage étant très concise, on risque d'obtenir des programmes illisibles. Cette partie précise les règles d'écriture à adopter afin d'en assurer la lisibilité et donc de faciliter les tâches de correction et de maintenance.

Les commentaires

Les commentaires servent à documenter un fichier source afin d'aider à la compréhension des programmes et apporter une « plus value » :

- entête descriptif d'un fichier ou d'une fonction
- décrire le rôle d'une variable
- résumer les étapes ou l'idée générale d'un traitement
- délimiter une zone de code à maintenir
- …

On utilise le caractère dièse (#) pour indiquer un commentaire dans un script.

Exemple : Il faut éviter ce type de commentaire

```
(( i = i + 1)) # on incrémente i !!! (qui est i ? pourquoi on
l'incrémente ?)
```

Mais, utiliser plutôt ceci par exemple :

```
(( i = i + 1)) # on passe au fichier suivant
```

Remarque : cas particulier

Les commentaires avec # ne sont pas considérés comme des instruction par un script. Dans ce cas, cette partie provoquera une erreur (de syntaxe) :

```
if (( a == 1 ))
then
   echo "\$a est égal à 1"
else
   # à faire (TODO)
fi
```

Dans ce cas on utilisera la commande nulle (:) :

```
if (( a == 1 ))
then
   echo "\$a est égal à 1"
else
   : à faire
fi
```

Tout ce qui se trouve derrière la commande ":" est ignorée.

Initialisation des variables

Les langages de script n'imposent pas de déclaration prélalable des variables. La variable sera donc déclarée et initialisée lors de sa première utilisation ce qui peut conduire à un dysfonctionnement.

On conseille donc d'initialiser toutes les variables en début de script.

Indentation du code

L'indentation du code facilite la lisibilité du code, notamment dans les structures de contrôle (if, for, while, ...).

On préconise une indentation correspondant à trois espaces.

Remarques : avec les éditeurs de texte, on peut indenter avec la touche tabulation (insertion du code ASCII HT ou TAB 0x09 équivalant à '\t') ou avec des espaces. L'utilisation des espaces garantit une réutilisation dans chaque éditeur (format fixe) par contre l'utilisation d'une identation avec TAB permettra une reconfiguration aisée de la largeur d'identation par chaque éditeur de texte.

Lisibilité

Il faut éviter d'utiliser des structures de codes complexes ou trop concises qui rendent le code plus difficile à comprendre.

Exemple :

```
# A éviter
repertoire=${1-`pwd`} # !?!
```

Utiliser une structure plus lisible :

```
# Premier argument de la ligne de commande = le répertoire à
traiter
# donc 1 argument à traiter
NB_ARGS=1
# Aucun argument reçu ?
if [ $# -ne "$NB_ARGS" ]
then
   repertoire=`pwd`     # par défaut, le répertoire courant
else
   repertoire=$1         # le répertoire passé en paramètre
fi
```

La modularité

Variables et constantes

L'objectif est d'éviter de « coder en dur » des limites ou des contraintes qui peuvent changer en cas de modification ultérieure : on centralisera dans une variable ou une constante.

Exemple 1 : à éviter

```
# le fichier de log existe ?
if [ -e /var/log/messages ]
then
   #on affiche les dernière lignes
   tail /var/log/messages
fi
# on vide le fichier
echo "" > /var/log/messages
# on met à jour la date
touch /var/log/messages
```

Vous décidez de modifier le script pour traiter le fichier /var/log/syslog.

Il est maintenant nécessaire de changer manuellement le script, instance par instance, et espérer ne rien oublier.

Pour assurer la modularité, on fera :

```
# Seulement cette variable sera à modifier plus tard !
FICHIER_LOG=/var/log/messages

# le fichier de log existe ?
if [ -f "$FICHIER_LOG" ]
then
   #on affiche les dernière lignes
   tail $FICHIER_LOG
fi
# on vide le fichier
echo "" > $FICHIER_LOG
# on met à jour la date
touch $FICHIER_LOG
```

Exemple 2 :

```
# Ne pas écrire
for ((i=0; i <= 10; i++))
do
   : # ...
done

# Mais :
LIMITE_MAX=10
for ((i=0; i <= LIMITE_MAX; i++))
do
   : # ...
done
```

Fonctions

On utilisera des fonctions :

- lorsqu'un traitement est utilisé plusieurs fois : on le regroupe dans une fonction
- lorsqu'un script devient trop complexe à analyser

Fichiers

Il arrive souvent qu'un fonction réalise un traitement qui sera réutilisable dans d'autres scripts. Dans ce cas, il est intéressant d'utiliser un découpage par fichiers et de regrouper des fonctions dans un fichier à part. Le script utilisera la commande interne point (.) pour inclure et exécuter ce fichier par le shell courant.

La commande interne . permet d'exécuter un fichier (de commandes) par le shell courant et non par un sous shell. Dans la programmation de scripts, cela revient à inclure un fichier externe qui peut contenir des déclarations de variables et/ou des fonctions.

Exemple : Le fichier affichage.fonctions.sh contient :

```
echon()
{
        printf "%s" "$*"
}
```

Un script pourra l'utiliser de la manière suivante :

```
#!/bin/bash
# inclure les fonctions d'affichage : echon(), ...
. affichage.fonctions.sh
echon hello world !!
exit 0
```

La fiabilité

Un script est souvent en interaction avec des paramètres passés en argument et des contraintes du systèmes. Il se doit de faire les vérifications suivantes :

- avant de réaliser une action (présence et validité des paramètres, présence de la commande utilisée)
- après l'exécution d'une action (échec ou réussite de l'action réalisée et traitement de l'erreur)

Exemple :

```
# aspell installé ?
if [ -x "`which aspell 2>&-`" ]
then
   spell="aspell list -d french"
   ispell="aspell -a -d french | sed 1d" # cf. 1
else
   echo "Erreur ! aspell n'est pas installé."
   exit 2
fi
```

Annexe 3 : La gestion des options

Il arrive souvent qu'un script ait besoin de traiter des options passées en arguments ce qui lui permet d'exécuter des actions différentes. Il existe plusieurs techniques pour traiter des options :

- le faire soi-même avec des if ... (solution lourde et complexe)
- utiliser la commande interne `getopts` (type getopts, donc man bash)
- utiliser la commande externe `getopt` (type getopt, donc man getopt)

Remarque : getopt() existe aussi en C (man 3 getopt).

La commande interne getopts

[Extrait du man bash]

```
Usage: getopts chaîne_d_options nom [arguments]

Description: getopts est utilisé par les scripts shell pour
analyser les paramètres positionnels. La chaîne_d_options
contient l'ensemble des lettres d'options à reconnaître. Si une
lettre est suivie par un deux-points (:), l'option est censée
comprendre un argument, qui peut en être séparé par une espace.
A chaque appel, getopts place l'option suivante dans la variable
shell dont le nom est fourni...
```

Exemple d'utilisation de getopts :

```
#!/bin/bash
# Traitement des arguments de la ligne de commande avec getopts
# Note 1: l'utilisation du tiret/plus (-/+) pour les options
est obligatoire pour getopts

#OPTIONS=":atfou:d:"
# Note 2: si deux-points (:) précède la liste des options, on
aura alors
# la valeur de l'option qui n'est pas dans la liste dans $OPTARG

OPTIONS="atfou:d:"
# Note 3: si deux-points (:) ne précède pas la liste des
options, on aura alors
# le message "illegal option" si l'option saisie n'est pas dans
la liste
# Note 4: Dans les deux cas précédents, la variable OPTION prend
la valeur "?"
```

```
# Note 5: les options u et d acceptent un argument ($OPTARG)
# Remarque: ce script ne traite pas le cas où une option est
précisée plusieurs fois ...

# Pas d'argument ?
if [ $# -eq "0" ]
then
  echo "Usage: $(basename $0) -atfoud [autreChose]"
  exit 1
fi

# Pour aider pendant les tests !!!
echo "Usage: $(basename $0) -atfoud [autreChose]"

while getopts $OPTIONS OPTION
do
  case $OPTION in
    a     ) echo "option -a";;
    t | f | o ) echo "option -$OPTION";;
    u     ) echo "option -$OPTION, avec argument \"$OPTARG\"";;
    d     ) echo "option -$OPTION, avec argument \"$OPTARG\"";;
    *     ) echo "option inconnue: -$OPTION, avec
argument \"$OPTARG\"";;
  esac
done

# shift $(($OPTIND - 1))
# Décrémente le pointeur d'argument pour pointer vers le
prochain.
echo "Argument autreChose sera en \$$OPTIND"
exit 0
```

Quelques tests avec ce script :

```
$ ./scriptGetopts.sh -a -t
Usage: scriptGetopts.sh -atfoud [autreChose]
option -a
option -t
Argument autreChose sera en $3

$ ./scriptGetopts.sh -at
Usage: scriptGetopts.sh -atfoud [autreChose]
option -a
option -t
Argument autreChose sera en $2

$ ./scriptGetopts.sh -a -d rep
Usage: scriptGetopts.sh -atfoud [autreChose]
option -a
option -d, avec argument "rep"
Argument autreChose sera en $4
```