Thème 2 – Scripts

TP disponible sur la page web personnelle d'Alexandre Sedoglavic : www.lifl.fr/~sedoglav/SHELL/

2.1 Instructions de contrôle

2.1.1 – Instructions composées conditionnelles

Exercices 2.25 et 2.26 – commandes composées conditionnelles et commande interne shift

On rédige le script suivant, on le nomme can.sh par exemple. Ensuite, on le rend exécutable en utilisant chmod 777 can.sh puis on crée l'alias can=./can.sh.
Le **shift** permet de supprimer le premier argument de la liste des arguments, il peut être utilisé pour se débarassé d'un argument déjà traité.

```
#!/bin/bash
CAN=$HOME/.poubelle
COM=`basename $0`

usage(){
    echo "Commande de gestion d'une poubelle"
    echo "Usage : $COM -l liste le contenu de la poubelle"
    echo "      : $COM -r d\'etruit la poubelle"
    echo "      : $COM foo bar d\'eplace foo et bar dans la poubelle"
}

if [ ! -d $CAN ]; then
    mkdir $CAN
fi

case $1 in
    -l) echo "$CAN :"
 ls -lR $CAN
 exit 0 ;;
    -r) echo "détruit la poubelle"
 rm -rf $CAN/*
 exit 0 ;;
-x) shift
      rm -rf $@
      exit 0 ;;
esac

if [ $# -ge 1 ]; then
    mv $@ $CAN
else
    usage
fi
```

Expliquons les différents blocs de ce script :

- La première ligne définit le shell utilisé par le script, la variable CAN est le répertoire de la poubelle et COM définit le nom du script choisi par l'utilisateur.
- usage() est une fonction qu'on appellera plus tard dans le code.

- Si il n'y a pas de répertoire $CAN, alors on le crée.
- On distingue les différents cas du premier argument ($1) : s'il s'agit de -l alors on affiche l'adresse de la poubelle puis le total contenu en méga-octets puis le nom des fichiers ; s'il s'agit de -r alors on efface son contenu ; s'il s'agit de l'option -x on supprime tous les arguments sauf -x, qui sera supprimé de la liste de tous les arguments grâce au shift.
- S'il y a plus d'un paramètre dans l'instruction appelant can, alors il s'agit de fichiers à envoyer dans la poubelle, on les y déplace.

Exercice 2.27 – commandes composées conditionnelles

Tester le script suivant en modifiant les paramètres de addtime :

```
addtime() {

if [ $# -ne 2 ]; then
  echo "Usage $0-addtime: addtime date1 date2"
  exit 1
fi

jour1=`echo $1 | cut -d+ -f1`
if [ $jour1 = $1 ]; then
  jour1=0
fi
jour2=`echo $2 | cut -d+ -f1`
if [ $jour2 = $2 ]; then
  jour2=0
fi
heure1=`echo $1 | cut -d+ -f2 | cut -d: -f1`
if [ $heure1 = `echo $1 | cut -d+ -f2` ]; then
  echo $0-addtime: Bad Format $1
  exit 1
fi
heure2=`echo $2 | cut -d+ -f2 | cut -d: -f1`
if [ $heure2 = `echo $2 | cut -d+ -f2` ]; then
  echo $0-addtime: Bad Format $2
  exit 1
fi
minute1=`echo $1 | cut -d+ -f2 | cut -d: -f2`
if [ $minute1 = `echo $1 | cut -d+ -f2` ]; then
  echo $0-addtime: Bad Format $1
  exit 1
fi
minute2=`echo $2 | cut -d+ -f2 | cut -d: -f2`
if [ $minute2 = `echo $2 | cut -d+ -f2` ]; then
  echo $0-addtime: Bad Format $2
  exit 1
fi

#echo $jour1 $heure1 $minute1 $jour2 $heure2 $minute2

heure=0
minute=$(( minute1 + minute2 ))
#echo ${minute}
if [ $minute -gt 60 ]; then
  heure=$(( minute / 60 ))
  minute=$(( minute % 60 ))
fi

jour=0
heure=$(( heure + heure1 + heure2 ))
```

```
#echo $heure
if [ $heure -gt 24 ]; then
  jour=$(( heure / 24 ))
  heure=$(( heure % 24 ))
fi

jour=$(( jour + jour1 + jour2 ))

if [ $jour -ne 0 ]; then
  echo $jour+$heure:$minute
else
  echo $heure:$minute
fi
}

addtime 15+19:20 21+7:45
```

2.1.2 – Instructions composées de répétition

La commande externe **ps** permet de lister les processus lancés par l'utilisateur. Avec l'option **ux**, cette commande affiche les informations suivantes :
USER est le nom de l'utilisateur, **PID** est le numéro de processus, **%CPU** et **%MEM** sont les pourcentages d'utilisation du CPU et de la mémoire par le processus, **VSZ** est la taille en kilo-octets du processus…

Exercices 2.28 et 2.29 – commandes composées for et arithmétique, for et if

Les scripts correspondant à ces deux exercices sont les suivants :

```
#!/bin/bash
# memtot

# Une variable pour contenir le résultat
SOMME=0

for i in `ps ux | cut -c26-29`; do
  SOMME=$(( SOMME + i ))
done

echo $SOMME
```

```
#!/bin/bash
# topuser

MAX=0

for i in `ps aux | tail +2 | cut -f1 -d' ' | sort | uniq`
  do
  SOMCPU=0
  for CPU in `ps aux | tail +2 | grep $i | cut -c16-19`
    do
    CPU=${CPU%.*}
    SOMCPU=$(( SOMCPU + CPU ))
  done
  if [ $SOMCPU -gt $MAX ]; then
      MAX=$SOMCPU
      MAXUSER=$i
  fi
done

echo "$MAXUSER utilise environ $MAX pour cent du CPU"
```

2.2 Fonctions

Exercice 2.30 – définition de fonction

Le C-shell fournit deux commandes internes **pushd** et **popd** facilitant la gestion des changement de répertoire à l'aide d'une pile. La commande **pushd** est équivalente à la commande **cd** mais elle mémorise (empile) le répertoire courant. La commande **popd** change de répertoire pour le répertoire qui se trouve en sommet de pile et supprime ce répertoire de la pile.

Une solution est d'implanter ces fonctions via des fonctions shell. On définit les deux fonctions dans un shell script. On utilise la commande interne **source** pour rendre les fonctions visibles au niveau du shell. L'appel aux fonctions sera alors possible directement depuis le shell.

$ source addtime.sh
$ addtime 15+10:00 21+9:30
$ source pushpop.sh
$ push Documents/

```
pushpopfile=$HOME/.pushpop
pushd (){
  pwd >> $pushpopfile
  cd $1
}

popd (){
  tempfile=/tmp/pushpop.$$

  lastline=`tail -1 $pushpopfile`
  cd $lastline

  nblines=`wc -l $pushpopfile | cut -f1 -d' '`
  nblines=$(( nblines - 1 ))
  cp $pushpopfile $tempfile
  head -$nblines $tempfile > $pushpopfile

  rm $tempfile
}
```

Exercice 2.31 – récursivité

Récursivité des shell scripts puis des fonctions.

```
#! /bin/sh -u
FILES=`ls $1`
for file in $FILES
do
  if [ -f $1/$file ]
  then
    echo $1/$file
  elif [ -d $1/$file ]
  then
    $0 $1/$file
  fi
done
```

On désire ajouter une indentation suivant le niveau de profondeur du fichier par rapport à la racine de l'arbre.
Une solution est de passer un paramètre **indent** à la commande. On passe donc par une fonction.

```
#! /bin/sh -u
IndentChar='-'

# treeindent dir indentstring
treeindent (){
  FILES=`ls $1`
  local IndentString=$2

  for file in $FILES
  do
    echo $IndentString $file
    if test -d $1/$file
    then
      treeindent $1/$file $IndentString$IndentChar
    fi
  done
}

treeindent $1 " "
```

La déclaration **local** de la variable **IndentString** permet de se rapprocher du fonctionnement habituel des langages de programmation. Cependant, cette commande **local** est une extension de **bash**. En Bourne shell standard, la variable **IndentString** est unique et donc partagée entre les différentes instances de l'appel à la fonction **treeindent**. Le fonctionnement du script n'est donc pas correct. On peut le modifier en s'appuyant sur le fait que les variables 1, 2… sont locales :

```
#! /bin/sh -u
IndentChar='-'

# treeindent dir indentstring
treeindent (){
  FILES=`ls $1`

  for file in $FILES
  do
    echo $2$file
    if test -d $1/$file
    then
      treeindent $1/$file $2$IndentChar
    fi
  done
}

treeindent $1 " "
```

Présentation de l'éditeur de fichiers emacs

$ emacs fichier &
Présentation de son utilité pour programmer, indenter…
Raccourcis complexes mais modifiables dans le menu.