Introduction à Unix

version 1.1 (2006)

Partie II : Programmation shell

Programmation shell : plan (1)

I. Introduction

- Qu'est-ce qu'un script shell ?
- Contenu d'un script shell
- Variables et paramètres
- Exécution et sortie d'un script shell

Programmation shell : plan (2)

II. Variables, paramètres et substitutions

- Variables de shell
- Substitutions
- Variables et affectations

Programmation shell : plan (3)

III. Structures de contrôle

- Groupage de commandes
- La commande if
- La commande test
- La commande case
- Les commande for et seq
- Les commandes while et until
- Les commandes break et continue
- Redirections

Programmation shell : plan (4)

IV. Les fonctions

- Définition
- Utilisation
- Exemples

Programmation shell : plan (5)

V. Commandes utiles

- La commande expr
- La commande eval
- La commande exec
- La commande getops
- La commande select

Programmation shell : plan (6)

VI. Mini-projets

- Gestion de sauvegardes
- Gestion d'une poubelle
- Un mécanisme d'annulation

I. Introduction

- **Qu'est-ce qu'un script shell ?**
- **Contenu d'un script shell**
- **Variables et paramètres**
- **Exécution et sortie d'un script shell**

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 traîtés 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/bash` (pour le Bourne Again 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 (1)

- **Variables spéciales**
  - `$0` : le nom du script (du fichier)
  - `$#` : 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 paramètres (2)

Variables prédéfinies en bash et mise à jour automatiquement

- PPID : le numéro du processus père
- PWD : le nom du répertoire courant
- RANDOM : un nombre entier aléatoire
- SECONDS : le temps (en secondes) écoulé depuis le lancement du shell courant
- ! : le numéro du dernier processus lancé en arrière-plan
- _ :le dernier mot de la dernière commande exécutée

Variable d'environnement pour l'exécution de script

- BASH_ENV : le nom du fichier bash exécuté à chaque début de script

Exemple de script (1)

```
#!/bin/bash
# début du script
echo "Hello $USER"
echo "You are currently on $HOST"
echo "The content of your homedir:"
/bin/ls
# affectation de la variable NBFILES
NBFILES=$(ls | wc -l)
echo "Total: $NBFILES 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 `bash my_script.sh`
  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 exécité correctement
  Différent de 0 signifie qu'une erreur est survenue

Exercices I.1

1. Ecrivez la commande `args` qui prend un nombre quelconque d'arguments eux-même quelconques et qui, lorsqu'on l'exécute, affiche son nom, le nombre de ses arguments et la liste de ses arguments :

```
$ args fichier $USER 777 tutu

args est appelé avec 4 argument(s) : fichier zorro
  777 tutu
```

2. Modifiez *éventuellement* la commande `args` pour qu'elle fonctionne sans aucune modification, même après qu'on l'ait renommée (par exemple, par `mv args script`). Autrement dit, la commande doit toujours afficher son nom !

3. Ecrivez la commande `words` qui affiche chaque mots du fichier (de texte) qu'on lui passe en argument, à raison d'un mot par ligne, chaque mot étant précédé de son nombre d'occurences. Les mots doivent apparaître triés par ordre alphabétique (consdérez les commandes `tr`, `sort` et `uniq`).

4. Modifiez la commande `words` pour qu'elle prennent plusieurs fichiers (de texte) en paramètres.

II. Variables, paramètres et substitutions

- **Variables de shell**
- **Substitutions**
- **Variables et affectations**

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`
- `echo $PRINTER`
- `export JAVA_SOURCE=$HOME/java/src`
  ```

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 varible `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)

Exercices II.1

1. Ecrivez une première version de la commande `welcome1` qui ne prend aucun argument, et qui,, lorsqu'elle est exécuté par l'utilisateur `zorro` sur la machine de nom `hal`, affiche les informations suivantes :

```
$ welcome1
Hello zorro, you're logged on hal and current date
and time are: Fri Oct  6 17:56:03 CEST 2006
```

2. Ecrivez une première version de la commande `nospace1` qui prend en argument un fichier ou un répertoire dont le nom contient des *espaces*, et qui renomme son argument en remplaçant les *espaces* par des `_` (soulignés). Vous devez utiliser la commande `tr` pour réaliser la commande `nospace1`. Exemple :

```
$ ls Bon*
Bonjour tout le monde
$ nospace1 Bonjour\ tout\ le\ monde
$ ls Bon*
Bonjour_tout_le_monde
```

III. Structures de contrôle

- Groupage de commandes
- La commande if
- La commande test
- La commande case
- Les commande for et seq
- Les commandes while et until
- Les commandes break et continue
- Redirections

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 differents"
```

- 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 chaine` (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 caractères "\"

Exemple

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

Exercices III.1

1. Ecrivez le script `welcome2`, deuxième version du script `welcome1` (exercice II.1.1), qui affiche le message de bienvenue adéquat en fonction de l'heure de la journée. Le message doit être `Good morning` entre 0 heure et 12 heures, `Good afternoon` entre 13 heures et 18 heures, et `Good evening` entre 19 heures et 23 heures :

```
$ welcome2
Good morning zorro, you're logged on hal
```

2. Ecrivez le script `info1` qui affiche les informations de l'unique fichier ou répertoire qui est son argument :

```
$ info tutu                $ info lien.sym
tutu n'existe pas          lien.sym est un lien symbolique
$ info /etc                $ info .bashrc
/etc est un répertoire     .bashrc est un fichier (rw)
```

Si le script `info1` est appelé avec zéro ou plus d'un argument, il doit produire un message d'erreur.

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
```

Exercices III.2

1. Ecrivez la commande `viz1` qui prend un fichier en argument et le visualise. Le programme de visualisation sera choisi en fonction de l'extension du fichier. La commande doit traîter les extensions `.ps` (fichier PostScript) et `.pdf` (fichier PDF). Pour visualiser les fichiers PostScript, vous utiliserez la commande `kghostview` et pour les fichiers PDF la commande `kpdf`.

2. Ecrivez la commande `viz2`, deuxième version de la commande `viz` précédente, qui maintenant peut visualiser également des fichiers d'extension `.txt` (fichier de texte). Un fichier de texte sera affiché par la commande `more` dans un `xterm` (considérez l'option `-e`). Prévoyez un message et une confirmation de fin de visualisation.

3. Ecrivez la commande `viz3`, troisième version de la commande `viz` , qui maintenant peut visualiser des fichiers d'extension *ext*, mais aussi des fichiers d'extension *ext*`.gz` (fichiers compressés avec la commande `gzip`) où *ext* est l'une des trois extensions précédentes. Pour les fichiers compressés, vous devez décompresser les fichiers sans toutefois modifier les fichiers arguments !

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
```

Exercices III.3

1. Ecrivez la commande `info2`, deuxième version de la commande `info1` (exercice III.1.2), qui maintenant prend un nombre quelconque d'arguments (au moins un) et affiche les informations pour chacun d'entre eux.

2. Ecrivez la commande `nospace2`, deuxième version de la commande `nospace1` (exercice II.1.2), qui prend maintenant en paramètres des fichiers ou des répertoires (au moins un) et qui, seulement quand le fichier ou le répertoire existe et quand son nom contient des *espaces,* le renomme en remplaçant ces *espaces* par des *soulignés*. Dans tous les cas, la commande ne provoque pas d'erreur.

3. Ecrivez la commande `size1` qui prend en arguments des fichiers et qui, pour chacun , affiche sa taille en (kilo-)octets suivie de son nom. La commande affiche `???` à la place de la taille si le fichier n'est pas un fichier régulier.

4. Ecrivez la commande `afind` qui prend en argument un répertoire et un nom, et qui cherche dans quelles bibliothèques `C` appartenants au répertoire le nom est défini. Une bibliothèque `C` est un fichier d'extension `.a` et on trouve en général les bibliothèques standards sous `/usr/lib`. Utilisez la commande `nm` pour connaître les noms externes définis dans une bibliothèque.

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
  echo $1; shift
done
```

```
until [ $# -eq 0 ]; do
  echo $1; shift
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 &
```

Exercices III.4

1. Ecrivez la commande `mycal` qui améliore la commande `/usr/bin/cal` en autorisant alternativement de qualifier les mois à l'aide de l'un des mots `jan`, `feb`, `mar`, `apr`, `may`, `jun`, `jul`, `aug`, `sep`, `oct`, `nov` et `dec`, et ce indépendemment de la casse (par exemple, `jan`, `Jan`, `JAN`, `JaN`, `jAn`, etc. sont équivalents). La commande `mycal` doit se comporter comme `/usr/bin/cal` et en particulier admettre les mêmes options. Quand l'unique argument est le mois en lettre, l'année est l'année en cours.

2. Ecrivez la commande `size2`, deuxième version de la commande `size1` (exercice III.3.3), qui affiche maintenant ses résultats en colonne (considérez la commande `column`).

3. Ecrivez la commande `chlnk` qui prend en paramètre un ou plusieurs répertoires, et qui affiche tous les liens symboliques invalides appartenant récursivement à ces répertoires. Un lien symbolique est invalide s'il ne pointe pas vers un fichier ou un répertoire existant.

4. Ecrivez la commande `chext1` qui prend en paramètre deux extensions `ext1` et `ext2` ainsi qu'un répertoire, et qui renomme tous les fichiers du répertoire de nom `xxx.ext1` en `xxx.ext2`. Si le nom du répertoire n'est pas fourni, la commande s'applique au répertoire courant.

IV. Les fonctions

- **Définition**
- **Utilisation**
- **Exemples**

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'autre 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 est interprétée dans le shell courant (pas de sous-shell)
  - une sorte d'alias avec paramètres
  - peut être définie dans le `.bashrc` pour être conservée
- Une fonction peut être exécutée dans le shell interactif courant
  - modification **possible** des variables d'environnement
- Une fonction peut être exécutée dans le sous-shell interprétant un script
  - Attention : l'évaluation de la commande `exit` depuis l'intérieur d'une fonction termine le **script shell englobant**
- Une fonction peut être récursive

Les fonctions (3)

Une fonction pour se déplacer facilement dans un répertoire

```
cdd() {
   for dir in $(find $HOME -name $1 -type d); do
        echo -n "$dir ? "
        read yes
        if [ -n "$yes" ]; then
            cd $dir; return 0
        fi
    done
}
```

Les fonctions (4)

- 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 }
```

Exercices IV.1

1. Ecrivez la commande `move1`, clone de la commande `mv`, mais qui n'admet aucune option et exactement deux arguments. Bien sûr, il est interdit d'utiliser la commande `mv` !

2. Ecrivez la commande `move2`, deuxième version de la commande `move1` précédente, qui est toujours un clone de `mv` sans option mais qui cette fois peut prendre plus de deux arguments.

3. Ecrivez la commande `move3`, troisième version de la commande `move1`, commande identique à `move2` mais qui admet éventuellement l'option `-i`.

4. Ecrivez la commande `copy1`, clone de la commande `cp`, mais qui n'admet aucune option et exactement deux arguments (un fichier régulier suivi d'un fichier ou un répertoire).

5. Ecrivez la commande `copy2`, deuxième version de la commande `copy1` précédente, qui est toujours un clone de `cp` sans option mais qui cette fois peut prendre plus de deux arguments.

6. Ecrivez la commande `copy3`, troisième version de la commande `copy1`, commande identique à `copy2` mais qui admet éventuellement les options `-i` et `-r`.

V. Commandes utiles

- La commande expr
- La commande eval
- La commande exec
- La commande getops
- La commande select

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 occurence de `caractère` dans `chaine`, ou 0 si `caractère` n'est pas présent dans `chaine`
- `expr length chaine`
  Affiche la longueur de `chaine`

Arithmétique entière

La notation ((...)) permet d'effectuer simplement des tests ou des calculs arithmétiques sur les entiers

- Test arithmétique :

```
if (( x > 1 )) ; then
   ...
```

Le `$` devant la variable `x` n'est pas nécessaire et les caractères `<` et `>` perdent leur signification spéciale

- Calcul (et substitution) arithmétique :

```
x=10
y=20
z=$((x + (2 * y)/5))
```

On peut utiliser les caractères +, *, /, (,)selon leur signification arithmétique usuelle

La commande eval (1)

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 eval (2)

Une fonction qui modifie une variable

```
function chvar() {
   eval $1="$2"
}
```

Exemple : modification de la variable PATH

```
[hal@/home/bob] echo $PATH
/bin:/usr/bin:/usr/local/bin
[hal@/home/bob] chvar PATH '~/bin:$PATH'
[hal@/home/bob] echo $PATH
/home/bib/bin:/bin:/usr/bin:/usr/local/bin
```

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 ficher 3

La commande getopts

Une commande pour traîter 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
```

La commande select

Une commande pour fabriquer simplement un menu de saisie

```
select action in ajouter supprimer modifier
do
   echo "Vous avez choisi $action"
   break
done
case $action in
     ajouter) ...;;
   supprimer) ...;;
    modifier) ...;;
esac
```

Exercices V.1

1. Ecrivez la commande `myhead`, clone de la commande `head`, mais admettant seulement les deux options `-n` et `-q` et ne lisant que dans un ou plusieurs fichiers (et pas sur l'entrée standard). Pour simplifier l'écriture de `myhead`, on interdira la forme `myhead -15` au profit de `myhead -n 15`. Enfin, lorsque l'option -n n'est pas présente, le nombre de lignes à afficher sera la valeur de la variable d'environnement `MYHEAD_DEFAULT` si elle existe, ou bien `10` sinon.

2. Ecrivez la commande `myuniq`, clone de la command `uniq`, mais admettant uniquement les trois options `-c`, `-d` et `-u` et ne prennant aucun argument (l'entrée et la sortie de la commande sont l'entrée et la sortie standards).

3. Ecrivez la commande `chext2`, deuxième version de la commande `chext1` (exercice III.4.4), commande identique à `chext1` mais qui admet éventuellement les options `-i` et `-r`. Si l'option `-i` est présente, la commade demande confirmation à l'utilisateur avant d'effectuer le changement d'extension. Si l'option `-r` est présente, la commande cherche les fichiers d'extension `.ext1` récursivement dans le répertoire.

4. Ecrivez la commande `kproc` qui prend en argument des mots et qui tue interactivement tous les processus dont le nom contient un ou plusieurs de ces mots. Pour chaque processus, la commande affiche toutes les informations qui lui sont associées.

VI. Mini-projets

- **Gestion de sauvegardes**
- **Gestion d'une poubelle**
- **Un mécanisme d'annulation**

Gestion de sauvegardes

Le but de ce mini-projet est d'écrire les deux commandes `backup` et `restore`, commandes permettant de gérer simplement des sauvegardes de fichiers et de répertoires.

La commande `backup` prend comme arguments un ou plusieurs fichiers ou répertoires, et en crée des copies. Par exemple, `backup xxxx` fabrique la copie `xxxx.bak`, que `xxxx` soit un fichier ou un répertoire. Si `xxxx` est un répertoire, son contenu est copié récursivement dans `xxxx.bak` mais les éléments qu'il contient ont le même nom relatif, à moins qu'on utilise l'option `-r`. Afin d'éviter les sauvegardes de sauvegardes de sauvegardes... il est interdit de sauvegarder un fichier de sauvegarde avec la commande `backup`.

La commande `restore` prend comme arguments un ou plusieurs fichiers ou répertoires qui ont été sauvegardés, et les remplace par leur copie. Après restauration, les copies sont effacées. Pour faciliter l'utilisation de la commande `restore`, on peut l'appeler indifféremment avec le nom du fichier à restaurer ou avec le nom de la sauvegarde. Ainsi, `restore xxxx` et `restore xxxx.bak` sont équivalents. La commande `restore` admet également l'option `-r`.

L'extension utilisée pour les fichiers de sauvegarde est la valeur de la variable `BACKUP_EXTENSION` ou bien `back` si cette variable n'est pas définie.

Gestion d'une poubelle

Le but de ce mini-projet est de concevoir et d'écrire un ensemble de commandes pour gérer une *poubelle*. La poubelle est un répertoire dans lequel on déplace des éléments (fichiers ou répertoires) qu'on peut éventuellement récupérer plus tard. La poubelle peut être vidée définitivement, après quoi les éléments qui s'y trouvaient ne sont plus accessibles. Les commandes à réaliser ne se limitent pas à déplacer les fichiers ou les répertoires simplement dans le répertoire poubelle. Par exemple, on peut mettre à la poubelle consécutivement deux fichiers de même nom relatif, ou plus délicat, de même nom absolu (mais néanmoins différents, car existant à des moments différents) et vouloir récupérer chacun d'eux ! L'ensemble des commandes à développer comprend au minimum les commandes :

- `trash`, qui met à la poubelle un ou plusieurs fichiers ou répertoires
- `intrash`, qui liste de manière lisible et structurée le contenu de la poubelle
- `untrash`, qui permet de récupérer simplement un fichier ou un répertoire contenu dans la poubelle (à partir des informations fournies par intrash)
- `cleantrash`, qui permet de vider tout ou partie de la poubelle en fonction de certains paramètres et options (par exemple, en fonction du temps depuis lequel le fichier ou le répertoire est dans la poubelle)

Un mécanisme d'annulation (1)

Certaines commandes Linux modifient leur environnement, comme les commandes `rm`, `cp` ou `mv` pour ne citer que les plus courantes. Il serait utile de disposer d'un mécanisme d'*annulation* de l'exécution d'une telle commande qui permette de revenir à l'état initial (d'avant l'exécution de la commande). Le but de ce mini-projet est de réaliser un ensemble de commandes pour gérer l'*annulation* de certaines commandes.

Le principe d'annulation proposé repose sur une commande très particulière, la commande `undo`. Par exemple, si on veut pouvoir annuler l'effet de la commande `mv xxx yyy` , il faut que la commande `mv` elle-même produise avant de s'exécuter la commande annulante `mv yyy xxx` . La commande annulante peut être placée toujours dans le même script, la commande `undo` ! Ainsi, les effets de la commande `mv xxx yyy` peuvent être annulés par la commande `mv yyy xxx` , à condition que le fichier `yyy` n'existe pas au moment de l'exécution de la commande `mv`. Dans le cas contraire, le commande `mv` doit effectuer en premier une sauvegarde du fichier `yyy`, par exemple avec `mv yyy /tmp/yyy` , et les commandes annulantes (placées dans le script `undo`) deviennent `mv yyy xxx` suivie de `mv /tmp/yyy yyy`. Pour résumer, on dira que c'est la commande à (éventuellement) annuler qui, connaissant ses paramètres, est à même de produire les commandes suceptibles d'annuler ses effets !

Un mécanisme d'annulation (2)

Notez que **seule** la dernière commande exécutée dans le shell interactif courant peut être (éventuellement) annulée. Aussitôt une nouvelle commande évaluée, si cette dernière n'est pas une commande annulable, la commande `undo` ne doit rien faire si ce n'est délivrer un message d'erreur.

Vous devez développer au minimum les trois nouvelles commandes `rm`, `cp` et `mv` , versions annulables des commandes initiales du même nom. Idéalement, ces commandes doivent admettre les mêmes options et les mêmes paramètres que les commandes intiales correspondantes, mais vous pouvez vous limiter aux options essentielles (`-i` et `-r` par exemple).

La commande `undo` doit admettre au minimum les options `-v` et `-n` :

- `-v` (verbose) indique que la commande `undo` affiche les opérations intermédiaires qu'elle réalise pour annuler la dernière commande tout en les effectuant

- `-d` (dry run) affiche les opérations intermédiaires qu'elle réaliserait pour annuler la dernière commande, mais sans les réaliser ! Remarquez qu'on doit pouvoir exécuter la commande `undo` après qu'on ait exécuté une commande annulable suivie d'un nombre quelconque de `undo -d` !