Cours n°3

1 Les processus « shell » (suite)

Un shell ne se contente pas seulement de lancer des processus tapés par l'utilisateur sur une seule ligne, c'est aussi un langage de programmation impératif, qui possède donc à ce titre des primitives telles que `if`, `while`, `for`, `case`, la gestion des variables, les redirections des entrées et sorties des processus, *etc.*

On peut donc non seulement taper des commandes assez sophistiquées, mais aussi écrire des programmes dans des fichiers et les faire interpréter par le shell comme s'il s'agissait de son entrée standard.

Un cas particulier important de tels fichiers sont les fichiers d'initialisation :

- Lorsqu'on lance un shell `bash`, avant de donner la main à l'utilisateur il exécute le fichier `$HOME/.bashrc` s'il existe. Ceci permet de personnaliser le shell. Par exemple on peut y placer la ligne « `alias ll='ls -l'` » et dès lors il suffira de taper `ll` au lieu de `ls -l` pour obtenir les informations « longues » sur les fichiers.
- Lorsque le shell est lancé directement par un login de l'utilisateur, il exécute d'abord (donc avant le `.bashrc`) le fichier `$HOME/.profile` s'il existe.

Pour terminer cette section, mentionnons que par convention le caractère Ctrl D en début de ligne indique la fin de l'entrée standard d'un processus. Ainsi pour sortir d'un shell, il suffit de taper ce caractère au lieu d'une nouvelle commande ; le shell comprend qu'il ne recevra pas d'autres commandes et il s'arrête donc. On obtient le même résultat avec la commande `exit`. Les shells « de login » (ceux lancés par un login de l'utilisateur) font exception pour éviter de se déloguer intempestivement sur une simple faute de frappe. Il faut utiliser la commande `logout` pour sortir de ces shells là, et donc se déloguer.

2 Les expressions régulières sous Unix

Il y a deux types d'expressions régulières : les expressions régulières *simplifiées* utilisées par les processus shell et les expressions régulières *standard* utilisées par des commandes de manipulation de chaînes de caractères.

Pour le shell, concernant les noms de fichiers, « `*` »remplace n'importe quelle suite de caractères dans les noms de fichiers (ou dans les patterns des `case` comme on le verra plus loin). De même « `?` » remplace n'importe quel caractère. Par exemple, si le répertoire de travail contient des fichiers appelés `yeast01.txt`, `yeast02.txt` jusqu'à `yeast28.txt` au milieu d'autres fichiers dont le nom ne commence pas par `yeast`, alors une commande de la forme

```
cat yeast*.txt > ALLyeast.txt
```

permettra d'avoir le contenu de ces 28 fichiers regroupés dans un seul fichier appelé `ALLyeast.txt`. En effet, le shell va « expander » le *motif* « `yeast*.txt` » de sorte que la commande `cat` recevra 28 arguments rangés dans l'ordre :

```
yeast01.txt yeast02.txt yeast03.txt . . . yeast28.txt
```

et comme son rôle est de recopier sur sa sortie standard le contenu des fichiers qu'on lui donne en argument, il va les recopier les uns après les autres sur sa sortie standard. Comme on redirige cette sortie vers le fichier `ALLyeast.txt`, ce fichier contiendra la concaténation des contenus des 28 fichiers.

La commande `cat yeast??.txt > ALLyeast.txt` aurait donné le même résultat mais avec un motif qui *filtre* mieux les noms de fichiers : par exemple un fichier `yeast231.txt` ou `yeast8.txt` ne seraient pas inclus dans la concaténation alors qu'avec la première commande ils l'auraient été. Ce contrôle n'est cependant pas total car un fichier `yeastAZ.txt` serait toujours inclus.

Les expressions régulières « standard » ne suivent pas ces règles simples du shell mais permettent un contrôle beaucoup plus complet des expressions filtrées par un motif. Les commandes les plus communes qui utilisent des expressions régulières standard sont par exemple `grep`, `ed`, `sed`, `expr`, *etc.*

Pour une définition complète des expressions régulières sous Unix, faire `man ed`. Nous donnons ici seulement les constructions les plus souvent utiles.

Pour ces expressions régulières :

— Un point remplace n'importe quelle lettre.
Par exemple le *pattern* (=motif) « `t.t.` » *matche* (=filtre) aussi bien `toto` que `tutu` ou `toti`, mais aussi `twtr`.
— Pour se limiter à certaines lettres, il faut les énumérer entre crochets.
Par exemple « `t[aiou]t[aiou]` » filtre les 16 possibilités de `tata` à `tutu` en passant par `toti`.
— Avec un tiret, on peut énumérer une séquence de lettres entre crochets.
Par exemple « [A-Z] » filtre toute lettre majuscule, ou encore « `[A-Z0-9]` » filtre tout caractère qui est une majuscule ou un chiffre.
Si l'on souhaite un véritable tiret dans l'énumération de caractères, il faut commencer par lui (comme dans « `[-xy]` »).
— On peut également indiquer les caractères qu'on ne veut pas. Dans ce cas on utilise un accent circonflexe pour indiquer une négation.
Par exemple « `[^0-9]` » signifie « tout caractère qui n'est pas un chiffre ».
— Pour les suites de caractères, une étoile indique une répétitiond'un pattern un nombre quelconque de fois (même 0 fois).
Par exemple « `Gr*oum*f` » filtre `Groumf` ou encore `Grrrroummmf`, mais aussi `Gouf` ou `Grouf`. Si l'on veut au moins un `r` et un `m`, on écrit « `Grr*oumm*f` ».
— Le caractère backslash est un caractère d'échappement. Ainsi « `.*\.jpg` » filtre `toto.jpg` mais pas `totoxjpg`. Si l'on veut un vrai backslash, il faut le doubler (parfois tripler ou quadrupler car backslash est aussi un caractère d'échappement du shell).
— Il y a une exception. Pour beaucoup de commandes de substitution, la forme spéciale « `debut\(milieu\)fin` » permet de ne retenir que le milieu.
Par exemple le pattern « `toto\(.*\)\.jpg` », lorsqu'il est appliqué à `toto123.jpg` retourne la chaîne de caractères `123`.

La commande `expr` sert à calculer toutes sortes de choses. Faire `man expr`. Parmi ses possibilités, il y a la gestion de pattern matching ; dans ce cas, on l'utilise sous la forme :

`expr "`*chaine*`" ':' '`*pattern*`'`

Par exemple la commande « `expr "toto123.jpg" ':' 'toto\([0-9]*\)\.jpg'` » donne la chaîne `123`.

Si le pattern n'est pas reconnu, `expr` retourne une chaîne vide.

La commande `grep` extrait d'un fichier texte les lignes qui matchent le motif donné en argument. Faites `man grep`.

3 Créer un shell script

Un shell comme `bash` est un langage interprété, on écrit donc des « shell scripts » qui sont simplement des fichiers de texte pur, rendus exécutables (mode au moins 5 avec `chmod`) et avec la première ligne d'en-tête « `♯! /bin/bash` » qui indique que c'est `bash` qui doit interpréter le texte qui suit. On peut ensuite programmer exactement comme on taperait dans un terminal chaque ligne pour faire la commande en mode interactif.

Une fois le fichier rempli et rendu exécutable, il suffit de placer dans un répertoire qu'on a fait reconnaître par la variable `$PATH`(typiquement `$HOME/bin`).
Nota : si le fichier n'est pas dans un répertoire de `$PATH`, on peut toutefois s'en servir en donnant son adresse complète au lieu de donner seulement son nom.

Cependant dans ce type d'usage du shell, on utilise plus souvent les *variables* et leur manipulation que lorsqu'on tape au clavier. Leur manipulation est assez simple car, en shell, toutes les données sont des chaînes de caractères :

```
toto="Hello bonjour"
```

affecte pour valeur la chaîne de caractères `"Hello bonjour"` à la variable `toto`. Par la suite, `$toto` fournira cette valeur :

```
echo $toto
```

écrit à l'écran :

```
Hello bonjour
```

Là où ça devient puissant, c'est qu'on peut on peut donner pour valeur à une variable le *résultat* d'une ligne de commande shell. Par exemple :

```
tutu=‘echo $toto | sed -e ’s/ //g’‘
```

donne à `tutu` la valeur `"Hellobonjour"` sans espace, puisque le `sed`, qui a pris la valeur de `$toto` en entrée standard, l'a recopié sur sa sortie standard en supprimant les espaces. Ces processus ont été mis entre accents graves (des « backquotes » à ne pas confondre avec les accents aigus), ce qui transforme leur sortie standard en une chaîne de caractère, que l'on a affectée à `tutu`...

Des variables très utiles sont les variables `$1`, `$2`, `$3`... Supposons que l'on écrive un shell script dont le nom est `macommande` et que l'utilisateur tape

```
macommande toto tutu titi
```

alors dans le shell script, on retrouve dans ces variables les *arguments* donnés par l'utilisateur. Donc `"$1"` vaut `toto`, `"$2"` vaut `tutu` et `"$3"` vaut `titi`. On peut alors les utiliser dans le shell script, ce qui permet de programmer des commandes « paramétrées » par les arguments que donne l'utilisateur.

Le nombre d'arguments qu'a donné l'utilisateur est de plus rangé dans la variable `$♯`. Par exemple ici `"$♯"` vaut `3`.

Mentionnons enfin qu'il y a deux façons de sortir d'un shell script : soit parce qu'on est à la fin du script et qu'il n'y a donc plus de commande à effectuer, soit en utilisant l'instruction `exit` qui termine immédiatement le processus.

4 La programmation en shell

Outre le fait que l'on peut inclure dans un shell script toute ligne que l'on taperait dans un terminal avec shell, on peut aussi bénéficier des primitives de contrôle de tout langage de programmation impérative :

Les expressions conditionnelles :

```
if [ ...ici_une_condition... ]
  then
    ...ici_des_commandes...
elif [ ...ici_une_autre_condition... ]
  then
    ...ici_d_autres_commandes...
  else
    ...encore_d_autres_commandes...
fi
```

la ligne « `fi` » indique la fin de l'expression conditionnelle « `if` » ; « `elif` » est une contraction de « else if » ; il peut y avoir autant de `elif` que l'on veut ; les `elif` et le `else` ne sont pas obligatoires ; en revanche la partie `then` est obligatoire.

Les boucles while :

```
while [ ...ici_une_condition... ]
do
  ...ici_des_commandes...
done
```

Les commandes sont répétées jusqu'à ce que la condition devienne fausse. [1]

Les boucles for :

```
for nomDeVariable in ...une_liste_de_noms_séparés_par_des_espaces...
do
  ...des_commandes qui_peuvent_utiliser_$nomDeVariable...
done
```

Il y a autant de tours de boucles que de noms dans la liste. Par exemple

1. les commandes peuvent être répétées 0 fois si la condition est fausse au moment d'entrer dans le `while`

```
for fichier in *
do
  if [ -d "$fichier" ]
    then
      echo "$fichier est un sous répertoire."
  fi
done
```

donne la liste des sous-répertoires du répertoire courant mais ignore les fichiers plats, liens symboliques, *etc.*

La programmation par cas :

Pour simplifier, commençons par un exemple d'usage

```
case "$uneAdresseDeFichier" in
  /*) echo "c'est une adresse absolue."
    ;;
  */*) echo "c'est une adresse relative dans l'arborescence."
    ;;
  *) echo "c'est une adresse fille directe du répertoire courant."
esac
```

Comme on le voit, le `case` est effectué en filtrant selon les expressions régulières au sens du shell (cf. cours précédent).

Quand on programme, il faut toujours écrire des *commentaires* qui permettent à un lecteur du programme (typiquement vous-même dans 6 mois...) de comprendre sans effort ce que fait le programme. Ces commentaires sont ignorés par le shell. Pour ce faire, le caractère « ♯ » indique à `bash` qu'il doit ignorer ce qui est écrit depuis le ♯ jusqu'à la fin de la ligne.

Pour plus de précisions, faire « `man bash` » et pour les conditions utiles pour les `if` et les `while` « `man test` » est plus simple à lire que la partie décrivant les conditionnelles du manuel de `bash`.

Dans un shell script, on peut aussi créer des fonctions : les curieux pourront faire `man bash` pour en savoir plus...

5 Exemple de shell script

Voici un exemple de shell-script conçu pour simplifier des noms de fichiers de la forme *numéro-Nom_Artiste-Titre_Album-Titre_Morceau*.mp3 en *numéro-Titre_Morceau*.mp3 (partant du principe que l'artiste et l'album sont déjà caractérisés par le répertoire où se trouvent les fichiers). Ce shell prend 3 arguments : la valeur du *Nom_Artiste*, celle du *Titre_Album* et enfin l'adresse du répertoire où se trouvent tous les fichiers à traiter.

```
#! /bin/bash
#
# Par défaut cette commande va dans $3 et fait tous les
#  mv Num-$1-$2-nomMorceau Num-nomMorceau
# possibles.
#
# On suppose qu'aucun des arguments de la commande ne contient d'espace.
# On suppose aussi qu'aucune adresse de fichier ne contient d'espace.
#
# analyse des arguments:
artiste=$1
album=$2
if [ -d $3 -a -w $3 -a -x $3 -a -r $3 ]
  then repertoire=$3
  else echo "Répertoire $3 non modifiable !"
    echo "Abandon."
    exit
fi
# on y va:
cd $repertoire
for nom in *.mp3
do
```

```
  nouveau=‘echo $nom | sed -e "s/-$artiste//1" -e "s/-$album//1"‘
  if [ $nouveau != $nom ]
    then mv $nom $nouveau
      echo "$nom --> $nouveau"
    else echo "NOTA: fichier $nom hors format"
  fi
done
```