– TD 5 – Scripts BASH –

Lyon 2 / ICOM / L1 informatique / Environnement Informatique et Internet

Didier Puzenat didier.puzenat@univ-lyon2.fr

1 Introduction

Dans le TD 3 nous avons vu qu'un script est un fichier texte contenant des commandes. Dans le présent TD nous allons voir qu'il est possible de faire des scripts plus complexes. Le but de cet enseignement n'est pas d'apprendre à programmer aussi nous n'allons pas étudier et encore moins écrire de scripts de plus de quelques lignes. Il est important de bien comprendre ce que font ces scripts et d'essayer d'anticiper leur comportement avant de les tester, sinon ce TD risque de ne pas vous être très profitable.

2 Structure de contrôle conditionnelle

Dans la plupart des langages, une structure de contrôle conditionnelle est de la forme :

```
si expression alors instruction(s)
```

où *expression* a une valeur de vérité ; par exemple en Python :

```
age=8
if (age>=7 and age<=77) :
  print ("cette BD est pour vous")
```

En BASH, l'expression est « remplacée » par l'exécution d'une commande dont la valeur de retour est utilisée pour décider de la suite du traitement, comme le font les opérateurs d'enchaînement « `&&` » et « `||` » déjà vus. À titre d'exemple reprenons sous la forme d'un script BASH avec un `if` la question 4 de la section 3 de la fiche de TD 3 (« trouver et tester l'enchaînement de commandes qui affiche "Le répertoire /root n'est pas accessible" si c'est bien le cas »). Par exemple :

```
dir=$PWD
if cd /root 2> /dev/null
then
  cd $dir
else
  echo "Le répertoire /root n'est pas accessible !"
fi
```

À noter que l'indentation (le fait de décaler le texte après le `then`) n'est pas prise en compte par BASH, le seul but est de rendre la structure plus lisible. Nous pouvons compléter le script avec également un message si le répertoire est accessible :

```
dir=$PWD
if cd /root 2> /dev/null
then
  echo "Le répertoire /root est accessible !"
  cd $dir
else
  echo "Le répertoire /root n'est pas accessible !"
fi
```

Par ailleurs, nous avons vu en cours la notion de « paramètres positionnels », pour rappel il s'agit d'arguments précisés lors de l'appel du script et accessibles dans le script, ainsi `$1` correspond au premier argument, `$2` au second, etc. Dans l'exemple précédent, nous pouvons par exemple passer en argument le répertoire à tester :

```
dir=$PWD
if cd "$1" 2> /dev/null
then
  echo "Le répertoire $1 est accessible !"
  cd $dir
else
  echo "Le répertoire $1 n'est pas accessible !"
fi
```

À faire, pour chacun des scripts précédents :

1. taper le scipt avec un éditeur (par exemple `nano`), sauver sous le nom `testacces` ;
2. rendre le script exécutable le cas échéant ;
3. lancer le script en n'oubliant pas le chemin (`./testacces`), et sans oublier de préciser l'argument pour le dernier script (par exemple `./testacces /home`).

Enfin, notons que plutôt qu'aller à la ligne nous pouvons enchaîner les commandes sur une même ligne avec « ; » mais ce n'est pas conseillé lorsqu'on débute en BASH ! Par exemple :

```
dir=$PWD
if cd "$1" 2> /dev/null ; then echo "Le répertoire $1 est accessible !" ; cd $dir
else echo "Le répertoire $1 n'est pas accessible !" ; fi
```

3 Built-in test command

Une commande interne à l'interpréteur (donc une « *built-in* ») nommée `test` est très utile dans un script, elle permet de comparer des valeurs, mais aussi de faire diverses vérifications sur les fichiers et répertoires. On peut par exemple utiliser l'opérateur `-ge` pour « *greater or equal* » :

```
test $age -ge 7 && echo "vous êtes assez grand pour lire Tintin"
```

```
if test $age -ge 7 ; then echo "vous êtes assez grand pour lire Tintin" ; fi
```

Afin de rendre les scripts plus lisibles – notamment pour un habitué des langages de programmation classiques – une syntaxe originale de la commande `test` est de n'écrire que son argument encadré par des crochets, par exemple :

```
[ $age -ge 7 ] && echo "vous êtes assez grand pour lire Tintin"
```

```
if [ $age -ge 7 ] ; then echo "vous êtes assez grand pour lire Tintin" ; fi
```

À faire :

1. lire intégralement le man de `test` (pour rappel en tapant « `man test` ») ;
2. déclarer une variable `age` et lui affecter une valeur (par exemple « `age=8` ») ;
3. tester les 4 lignes précédentes (ne pas hésiter à copier-coller) ;
4. recommencer avec d'autres valeurs pour `age`.

Il est évidemment possible d'utiliser des expressions plus complexes en utilisant les trois opérateurs de base de la logique booléenne (« et » (`-a`), « ou » (`-o`) et « non » (`!`)), par exemple :

```
[ $age -ge 7 -a $age -le 77 ] && echo "vous avez l'âge de lire Tintin"
```

Vous vous demandez sans doute pourquoi « `$age -ge 7` » et non simplement « `$age >= 7` » comme dans l'exemple Python donné au début de la fiche ? C'est une conséquence de l'absence de typage, en effet pour le BASH toutes les variables sont des chaînes de caractères, dès lors il faut un opérateur particulier pour spécifier qu'il faut traiter le contenu comme un entier. Les opérateurs possibles sont `-eq` pour « *equal* », `-ne` pour « *not equal* », `-lt` pour « *less than* », `-le` pour « *less or equal* », `-gt` pour « *greater than* » et `-ge` pour « *greater or equal* ».

Il existe de plus de nombreux tests possibles sur les fichiers et répertoires, par exemple l'opérateur unaire « `-d` » teste si un répertoire existe :

```
[ -d "$rep" ] && echo "$rep existe et est un répertoire !"
```

Quelques opérateurs unaires incontournables :

`[ -a fichier ]`	True si `fichier`	existe ;
`[ -d fichier ]`	True si `fichier`	est un répertoire ;
`[ -e fichier ]`	True si `fichier`	existe (que ce soit un fichier normal ou un répertoire) ;
`[ -r fichier ]`	True si `fichier`	existe et est lisible (droit `r`) ;
`[ -s fichier ]`	True si `fichier`	existe et n'est pas vide ;
`[ -w fichier ]`	True si `fichier`	existe et peut être modifié (droit `w`) ;
`[ -x fichier ]`	True si `fichier`	existe et est un fichier exécutable (droit `x`) ou un répertoire traversable (droit `x`).

À faire : réécrire la commande `testacces` en utilisant un des opérateurs de la liste précédente, la solution est à la fin de la fiche, section 8.1.

Il existe aussi des opérateurs travaillant sur les chaînes de caractères, par exemple pour tester si deux chaînes sont identiques :

```
[ "$(pwd)" == "$HOME" ] && echo "Bienvenu à la maison"
```

Les opérateurs de comparaison principaux sur les chaînes sont :

`[ chaîne1 == chaîne2 ]` True si les deux chaînes sont identiques ;
`[ chaîne1 != chaîne2 ]` True si les deux chaînes sont différentes ;
`[ chaîne1 < chaîne2 ]` True si `chaîne1` est alphabétiquement avant `chaîne2` ;
`[ chaîne1 > chaîne2 ]` True si `chaîne1` est alphabétiquement après `chaîne2` ;

et il est aussi possible de tester si une chaîne est vide :

`[ -z chaîne ]` True si la chaîne `chaîne` est vide ;
`[ -n chaîne ]` True si la chaîne `chaîne` n'est pas vide.

À faire :

1. donner une ligne de commandes qui affiche un message si on est en 2022, une solution possible est proposée section 8.2 ;
2. proposer un script qui affiche le titre du roman dans lequel le mot « ballon » apparaît plus souvent, entre « De la Terre à la Lune » et « Le tour du monde en 80 jours » ; les fichiers étant disponibles respectivement aux URL suivantes :
   https://www.gutenberg.org/files/799/799-0.txt,
   https://www.gutenberg.org/cache/epub/800/pg800.txt,
   une solution est disponible section 8.3.

4 Pattern matching

Une alternative aux crochets est l'usage de doubles crochets ; le principal intérêt est l'utilisation d'expressions plus complexes lors de la comparaison de chaînes de caractères, par exemple l'étoile (`*`) équivaut à une suite de caractère quelconques (une suite éventuellement vide), le point d'interrogation (`?`) représente un et un seul caractère quelconque, etc. On parle de « *pattern matching* » ce qui en français pourrait se traduire par « mise en correspondance de formes ».

À faire : tester `[[ $(date) == *2022* ]] && echo "on est en 2022"` ;

voir section 8.4 si vous n'avez pas de machine pour tester (par exemple si vous révisez chez vous).

5 Affectation des paramètres positionnels et découpage de chaîne

L'usage de la commande `set` permet d'initialiser les paramètres positionnels aux arguments de `set`. À tester :

`set a b c ; echo $1 $2 $3 $#` (voir section 8.4 si vous n'avez pas de machine pour tester) ;

`set $(date) ; echo $4` (voir section 8.4 si vous n'avez pas de machine pour tester).

Comme le montre le dernière exemple, l'usage de la commande `set` peut évidemment être combiné avec les expansions, en l'occurrence l'expansion de commandes – aussi appelée substitution de commandes – (`$(date)`). Il est aussi possible de faire une expansion de variables. Pour rappel, lors d'une expansion de variables, une chaîne non protégée par des guillemets « subit » un découpage (en anglais « *word spliting* ») sur la base d'un ensemble de caractères séparateurs précisés par la variable IFS pour « *Internal Field Separator* ». Par défaut, les caractères séparateurs sont l'espace, la tabulation et le retour à la ligne. À tester :

`toto="/home/jlpicard/mission.txt" ; IFS="/" ; echo "$toto"` (solution section 8.4) ;

`toto="/home/jlpicard/mission.txt" ; IFS="/" ; echo $toto` (solution section 8.4) ;

`toto="Title: De la Terre à la Lune" ; IFS=":" ; set $toto ; echo $2` (voir 8.4).

Attention, le « *word spliting* » est uniquement lié à l'expansion de variables ; à tester :

`IFS=":" ; set Title: De la Terre à la Lune ; echo $2` (solution section 8.4).

6 Boucles en BASH

Nous avons vu la structure de contrôle `if then`, mais il en existe bien d'autres, notamment pour itérer sur une commande ou sur une série de commandes. Voici la syntaxe d'un `while` qui boucle tant qu'une commande renvoie une valeur de succès :

`while` *commande(s) de test*
`do` *commande(s) à itérer*
`done`

À tester :

(voir la section 9 si vous n'avez pas de machine pour tester, idem pour les questions suivantes de cette section)

```
num=5
while [ $num -ge 0 ]
 do
   echo $num
   ((num--))
done
```

À noter l'utilisation de la double parenthèses vue en cours et qui permet de faire simplement l'expansion et l'évaluation arithmétique. La syntaxe est proche de celle du langage C que certains d'entre-vous verront d'ici la fin de leur licence. Tester par exemple :

```
a=$(( 5 + 3 )) ; b=$(( a * 2 )) ; echo $a $b
((a++)) ; ((b--)) ; (( c = 42 )) ; echo $a $b $c
```

Comme vous l'avez compris, l'opérateur unaire « `++` » incrémente la variable et l'opérateur unaire « `--` » la décrémente.

Petit challenge facultatif (un peu difficile) : comme en C, il est possible de préciser si la variable doit être incrémentée avant ou après son utilisation, respectivement en plaçant l'opérateur avant ou après la variable. À deviner puis à tester pour vérifier :

```
n=1; (( --n )) && echo "vrai" || echo "faux"
n=1; (( n-- )) && echo "vrai" || echo "faux"
```

7 Petit script pour envoyer plein de mails

Le script suivant devrait vous être familier. La boucle principale (boucle `while`) traite une à une les lignes du fichier « `QCM1.csv` » qui est ni plus ni moins que le fichier résultant de l'exportation des résultats du premier QCM fait en TD. Le format utilisé (csv) n'est qu'une suite de champs séparés par des virgules (nom, prénom, numéro d'étudiant, adresse email, etc.), avec une « tentative » par ligne (et donc un étudiant par ligne). Vous ne pouvez pas tester ce script sans avoir configuré Linux pour l'envoi de mails, et vous n'avez pas le fichier avec les notes, mais cela ne devrait pas vous empêcher de comprendre le script. À noter la première ligne qui précise que le shell BASH doit être utilisé pour ce sript, indispensable dans un environnement dans lequel ce ne serait pas le shell par défaut.

```
#!/bin/bash
while read line; do
 IFS=";" ; set $line ; prenom=$2 ; email=$4 ; date=$6 ; note=${11}
 if [[ $email == *univ-lyon2.fr ]]
 then
  IFS=" " ; set $date ; jour=$1 ; mois=$2
  echo -e "From: Didier Puzenat <didier.puzenat@univ-lyon2.fr>" >  mail.tmp
  echo -e "To: <$email>" >>  mail.tmp
  echo -e "Subject: Note du QCM 1/3 de Environnement informatique et Internet" >> mail.tmp
  echo -e "Bonjour $prenom,\n" >> mail.tmp
  echo -e "votre note au QCM du $jour $mois est $note/20.\n" >> mail.tmp
  echo -e "Pour information la moyenne est de 9,93 et la meilleure note est 20.\n" >> mail.tmp
  echo -e "Deux autres QCM sont prévus, les $((jour+14)) et $((jour+21)) $mois.\n" >> mail.tmp
  echo -e "Cordialement, Didier Puzenat" >> mail.tmp
  cat mail.tmp | ssmtp -v $email
fi
done < "QCM1.csv"
```

8 Solutions

8.1 Solution pour la commande `testacces`

```
if [ -x $1 ]
then echo "Le répertoire $1 est accessible !"
else echo "Le répertoire $1 n'est pas accessible !"
fi
```

8.2 Test de l'année en cours

```
[ $(date +%Y) == "2022" ] && echo "on est bien en 2022 !"
```

Mais évidemment il y a d'autres solutions, par exemple :

```
date | grep -q 2022 && echo "on est bien en 2022 !"
```

8.3 Qui a le plus de ballons ?

```
wget https://www.gutenberg.org/files/799/799-0.txt
wget https://www.gutenberg.org/cache/epub/800/pg800.txt
if [ $(grep -c ballon 799-0.txt) -gt $(grep -c ballon pg800.txt) ]
then echo "De la Terre à la Lune"
else echo "Le tour du monde en 80 jours"
fi
```

8.4 Affichage des commandes des sections 4 et 5

`[[ $(date) == *2022* ]] && echo "on est en 2022"` affiche « `on est en 2022` » ;

`set a b c ; echo $1 $2 $3 $#` affiche « `a b c 3` » ;

`set $(date) ; echo $4` affiche « `2022` » ;

```
toto="/home/jlpicard/mission.txt"
IFS="/" ; echo "$toto"
```

affiche « `/home/jlpicard/mission.txt` » ;

```
toto="/home/jlpicard/mission.txt"
IFS="/" ; echo $toto
```

affiche « `home jlpicard mission.txt` » ;

```
toto="Title: De la Terre à la Lune"
IFS=":" ; set $toto ; echo $2
```

affiche « `De la Terre à la Lune` » ;

`IFS=":" ; set Title: De la Terre à la Lune ; echo $2` affiche « `De` » !!!

9 Affichage des commandes de la section 6

```
num=5 ; while [ $num -ge 0 ] ; do echo $num ; ((num--)) ; done
```

affiche dans la console :

```
5
4
3
2
1
0
```

```
a=$(( 5 + 3 )) ; b=$(( a * 2 )) ; echo $a $b
((a++)) ; ((b--)) ; (( c = 42 )) ; echo $a $b $c
```

affiche dans la console :

```
8 16
9 15 42
```