Shell

Le but de cette partie est de vous familiariser avec la notion de **shell-script** sous Unix. Vous pourrez constater au cours de ces séances que le **shell** est plus qu'un interpréteur de commandes, c'est également un *langage de programmation* à part entière.

Avant de créer et d'utiliser vos propres **script shells**, vous allez vous familiariser avec diverses notions de bases qui seront indispensables lorsque vous voudrez créer vos propres fichiers scripts.

1 Les shells

Il existe de nombreux **shell**, i.e., `sh`, `csh`, `tcsh`, `ksh`, `zsh`, etc. Le **shell** que vous utilisez se nomme le `bash`[1]. Le `bash` utilise des fichiers de configuration, qui se trouvent dans votre répertoire d'accueil. Par exemple, le fichier **.bashrc** sert à définir les initialisations des sessions interactives (rattachées à un terminal). Le fichier **.bash_profile** permet de personnaliser ses sessions.

Ces fichiers sont exécutés par le **shell** s'ils se trouvent dans votre répertoire d'accueil. Vous pouvez constater que ces fichiers sont présents dans votre espace de travail, et vous êtes invités à les consulter. Vous pouvez également en modifier le contenu afin de personnaliser votre environnement de travail.

IMPORTANT : FAITES UNE SAUVEGARDE PRÉALABLE DE VOS FICHIERS DE CONFIGURATION AVANT D'Y EFFECTUER DES MODIFICATIONS.

1.1 EXERCICE Empilez plusieurs shells en tapant successivement `bash`, `bash`, `tcsh`, `csh`, `sh` et observez la parenté des différents processus par la commande `ps -l`[2].

1.2 EXERCICE Dépilez les shells, en utilisant plusieurs fois et successivement les commandes `exit` et `ps -l` afin de vérifier que vous êtes revenu à votre premier shell dans les liens de parentés.

2 Exécution d'une commande

2.1 Notion de variable

Sous votre shell, il est possible de définir des variables. Par exemple sous votre environnement est défini une variable `DISPLAY` contenant le nom de votre terminal.

2.1 EXERCICE Exécutez les commandes `echo DISPLAY` et `echo $DISPLAY`. Qu'en déduisez-vous ? Comment accède-t-on au contenu d'une variable ?

2.2 EXERCICE Afficher le contenu de la variable `FOO`.

2.3 EXERCICE Exécutez la commande `FOO=DISPLAY`. Afficher le contenu de la variable `FOO`.

2.4 EXERCICE Exécutez la commande `FOO=$DISPLAY`. Afficher le contenu de la variable `FOO`.

1. Le *GNU Bourne-Again SHell* est du domaine public et a été développé par la *Free Software Foundation* sous licence GNU (si ce n'est déjà fait, je vous engage à visiter, **après le TD**, l'URL : `http://www.gnu.org`). Ce shell contient toutes les propriétés d'interactivité du csh et ksh. Son langage de programmation est compatible avec sh.

2. Rappeleons que le champ `PID` fait référence au numéro du processus, tandis que le champ `PPID` fait référence au numéro du processus père.

2.2 Recherche et lancement d'une commande

2.5 Exercice Afficher le contenu de la variable `PATH`. Sauvegardez cette valeur dans une variable `PATHSVG` et vérifier la valeur sauvegardée. Mettre la chaîne vide dans `PATH` et essayez les commandes `cd`, `pwd`, `ls`. Que peut on en conclure ? Comment lancer `ls` sans modifier `PATH` ? Modifier la variable `HOME` en lui affectant le chemin absolu d'un de vos répertoire (pas le répertoire d'accueil). Observez ce que produit la commande `cd` sans argument. **Restaurer les valeurs de `PATH` et de `HOME`.**

2.3 Valeur de retour d'une commande

Les variables `PATH`, `DISPLAY` tout comme `FOO` ont en fait été définies dans votre environnement et vous pouvez en modifier la valeur. Mais il existe quelques variables particulières affectées par le shell lui même. L'une d'entre elles se nomme `?`.

2.6 Exercice Exécutez la commande `ls -l` et affichez le contenu de la variable `?`.

2.7 Exercice Exécutez la commande `ls -z` et affichez le contenu de la variable `?`.

Remarque 1 La variable `$` retourne le `PID` du shell courant. Affichez son contenu et vérifiez sa valeur à l'aide de la commande `ps -l`.

2.4 Action conditionnelle

2.8 Exercice Dans votre fenêtre terminal, exécutez la commande `ls -l && echo OK || echo KO`. Puis exécutez la commande `ls -z && echo OK || echo KO`. Quel est le rôle des opérateurs `&&` et `||` ? Examinez dans chacun des deux cas précédents le contenu de la variable `?`. Expliquez son contenu. Modifiez la commande `ls -z && echo OK || echo KO` de telle sorte que le message d'erreur généré soit redirigé vers `/dev/null`.

3 Mécanismes de substitution

Interprétez et commentez le résultat des commandes suivantes :

1) Substitution des accolades
```
$ echo a{ab,c,edf}f
$ mkdir a{1,2,5}
```
2) Substitution du caractère tilde
```
$ echo ~/bar
$ echo ~oldelmas/bar
```
3) Substitution de variables
```
$ ab=totot
$ echo $ab
$ echo ${ab}abababab
$ echo $abababab
$ y=p ; ${y}wd
```
4) Substitution de commandes
```
$ echo `ls -l`
$ echo $(ls -l)
```
5) Substitution arithmetique
```
$␣echo␣p$((␣3␣+␣4␣))q
```
6) Substitutions de chemins
```
$ cd
$ echo */*
$ echo ../?[x-z]*
$ echo ../?[x-z]?
```
7) Substitutions et execution de commandes
```
$ x=p; y=$(${x}wd); echo cwd=$y
$ x=p; y=`${x}wd`; echo cwd=$y
```

3.1 Quotations

1) Déduire le rôle des quotes et doubles quotes
```
$ x=p; y='${x}wd'; echo $y
$ echo "*"*
$ echo "* $(pwd)"
```

2) Expliquer le comportement des commandes

```
$ A=$(echo *)
$ echo '$A'
$ echo "$A"
$ echo $A
$ $A
$ "$A"
```

3.1 Exercice Faire afficher les chaînes suivantes

`***␣BONJOUR␣***`

`␣-␣␣␣Le␣caractere␣'*'`

`/Fichier␣d'entree\`

`*␣dimanche␣18␣mars␣2012,␣22:41:42␣(UTC+0100)␣*` (la date doit être le résultat d'une exécution de `date`)

`$$␣dimanche␣18␣mars␣2012,␣22:37:33␣(UTC+0100)␣$$` (idem)

`$6248$` (où le nombre entre `$` est le *pid* du *shell*)

`(␣/bin/cat␣/bin/tar␣)` c'est-à-dire la liste de fichiers dans le répertoire `/bin` dont le nom est à trois lettres et dont la deuxième lettre est une de lettres a–d (utiliser une substitution de chemins).

4 Les script shells et mécanismes de substitution

Pour Unix, un fichier contenant une suite de commandes shell est appelé un **script** et ne nécessite que le droit d'exécution pour pouvoir être exécuté.

Pour exécuter le script deux méthodes sont possibles. Soit vous tapez, le nom du shell suivi du nom du script à interpréter :

```
$ bash mon-script
```

soit vous tapez directement le nom du script :

```
$ mon-script
```

Dans ce dernier cas, le fichier script doit posséder la permission en exécution et pour garantir son exécution par un shell de type `sh`, la première ligne de ce fichier doit indiquer le chemin absolu du shell qui sera utilisé. La syntaxe de cette ligne est la suivante :

```
#!/bin/sh
```

Si cette ligne est omise, c'est le shell de type `bash` qui est utilisé par défaut. Dans tous les cas, prenez la bonne habitude d'insérer systématiquement cette ligne particulière dans vos script shells.

4.1 Exercice Créez un fichier exécutable `cmd.sh` contenant :

```
#!/bin/sh
ps -f
```

Dans votre fenêtre `terminal`, lancez la commande `cmd.sh`. Si tout va bien, vous devriez voir s'afficher la liste de vos processus et en particulier un processus (à peu de chose près) de la forme `/bin/sh cmd.sh`. Modifiez la ligne `#!/bin/sh` afin d'utiliser un autre shell que `sh` et réexaminez le résultat produit par l'exécution de la commande `cmd.sh`.

4.2 Exercice Créez un répertoire `~/bin`. Désormais tous vos scripts seront créés dans le répertoire `~/bin`.

4.3 Exercice Modifiez votre environnement de façon à ce que les scripts inclus dans le répertoire `~/bin` soient utilisables depuis n'importe quel endroit sans en spécifier le chemin d'accès. Pour ce faire vous modifierez, dans vos fichiers de configuration, la variable `PATH` de telle sorte que celle-ci prenne en compte votre répertoire `~/bin`[3]. Attention : la variable `PATH` prend peut-être déjà en compte votre répertoire `~/bin`[4]. Pour le vérifier faites `echo $PATH`.

4.4 Exercice Rendre la modification de la variable `PATH` valide d'une connexion sur l'autre.

3. Vous rajouterez à l'endroit approprié la ligne :`PATH=$HOME/bin:$PATH`
4. Dans ce cas vous n'avez rien à modifier.

4.1 Les variables prédéfinies

Le shell offre des variables prédéfinies facilitant la programmation sous le shell. Vous avez déjà pu observer le rôle des variables `?` ou `$`. Cherchez dans le manuel de `bash` le rôle des variables `#`, `*`, `@`, `0`, `1`, `2`, `3`, etc.

4.5 Exercice Insérez dans un script shell, une commande permettant d'afficher le nom absolu du script lancé (chemin depuis la racine + nom de la commande). Pour en vérifier le bon fonctionnement, renommez votre fichier en `autre_nom.sh` et exécutez le.

4.6 Exercice Insérez une commande affichant le nombre de paramètres du fichier de commandes de la manière suivante :

$$nomvar = \text{'}nb\_param\text{'}$$

où *nomvar* est le nom de la variable contenant cette information et *nb_param* son contenu. Vérifiez que cela fonctionne en lançant le fichier de commandes avec plusieurs paramètres.

4.7 Exercice De la même façon, insérez une commande affichant la liste des paramètres du fichier de commandes. Trouvez une autre façon d'écrire la commande précédente pour obtenir le même résultat.

4.2 Substitution de commandes et de variables

4.8 Exercice Examinez les scripts qui suivent,

petit-script-4

```
#!/bin/sh
date
echo Il est $4
```

petit-script-5

```
#!/bin/sh
set ‘date‘
echo Il est $4
```

et exécutez les commandes :

```
$ petit-script-4 donald riri fifi loulou picsou
$ petit-script-5 donald riri fifi loulou picsou
```

afin d'interprétez le rôle de la commande : `set` ≺ *commande* ≻.

4.9 Exercice Écrire un script qui affiche l'heure sous la forme :

```
Il est 17:45:26.
```

Consultez le manuel de `bash` afin d'examiner le rôle de la variable IFS.

4.10 Exercice En modifiant la variable `IFS`, écrire un script qui affiche :

```
Il est 17h 45mn.
```

4.11 Exercice Écrire un script `prepa` qui évoqué comme `prepa` ≺*nom*≻ crée dans le répertoire courant un fichier exécutable ≺*nom*≻ dont la première ligne est `#!/bin/bash` suivi d'une ligne vide.

4.12 Exercice Écrire, sans utiliser de structure de contrôle, les commandes suivantes :

1) `nf` : affiche le nombre de fichiers ne commençant pas par un point, du répertoire courant [5] ;
2) `ra` : affiche `oui` si le répertoire courant est le répertoire d'accueil et `non` sinon [6] ;
3) `prc` : affiche la profondeur du répertoire courant [7] ;

5 Enchaînement des substitutions

5.1 Exercice Expliquer le comportement de chacune des lignes de commandes suivantes :

```
A='echo *'
$A
echo $A
echo '$A'
echo "$A"
```

De même avec cette fois ci le **remplacement de commandes** : `A=‘echo *‘`

5.2 Exercice En observant le comportement des lignes de commandes suivantes, expliquer ce que fait `eval`.

```
CMD=echo
ARG='$$'
LIG='$CMD [$ARG]'
$LIG
eval $LIG
eval eval $LIG
eval eval eval $LIG
```

5.3 Exercice Après avoir bien compris les lignes précédentes, définissez les routines suivantes à exécuter avec `eval`

- Definissez la variable `NOM_CMD` contenant un nom de commande bidon
- `USAGE` affiche un message d'usage de la commande bidon, i.e le nom de la commande avec quelques parametres usuels..., avec comme parametre `NOM_CMD`
- `ERREUR` affiche un message d'erreur contenu dans la variable `MES_ERR` et qui affiche ensuite l'usage de la commande).

6 Environnement

6.1 Exercice Observer et expliquer la suite de commandes :

```
echo $A
A=aaaaa
echo $A
```

5. Vous utiliserez les commandes `echo`, `wc` et `pwd`.
6. Vous utiliserez les commandes `test`, `pwd`, `echo` et les opérateurs `&&` et `||`.
7. Vous utiliserez les commandes `set`, `pwd`, `shift`, `echo` ainsi que la variable `IFS`.

```
bash
echo $A
exit
export A
bash
echo $A
A=bbbbb
echo $A
exit
echo $A
```

6.2 Exercice Enlever et remettre l'attribut *export* à la variable `DISPLAY` et vérifier à chaque fois le résultat en empilant un *shell*.

6.3 Exercice Que fait le script suivant :

```
#!/bin/sh

echo " --> $$"
```

Testez ce script et comparez avec le résultat affiché en entrant directement la commande `echo $$` dans votre bash. Expliquez ce qui se produit à l'exécution. Essayez en préfixant la commande par `.` ou par `source`. Que peut-on en conclure ?

6.4 Exercice Que fait le script suivant :

```
#!/bin/sh

if [ $# = 0 ]
then
    cd ..
else
    cd $1
fi
echo " --> ‘pwd‘"
```

Tester cette commande et expliquer ce qui se produit à l'exécution. Essayer en préfixant la commande par `.` ou par `source`. Que peut-on en conclure ?

6.5 Exercice L'instruction `if` peut également utiliser entre les crochets des tests sur des fichiers ou des mots. Exemples :
`[ -f fichier ]` vrai si “fichier“ existe et est un fichier régulier,
`[ -z mot ]` vrai si la longueur de “mot” est zero.
Ecrire un script qui fait une archive (à l'aide de la commande `tar -czf nom_archive liste_fichiers`) dont le nom sera donné en paramètre de la commande et qui contiendra tous les fichiers python du répertoire courant. Prévoir des messages d'erreur si la commande n'a pas de paramètre ou si le répertoire ne contient pas de fichiers python.

7 Itérations

7.1 Exercice Tester la boucle `for` au moyen des programmes suivants :

```
for N in un deux trois quatre
do
  echo "**$N**"
done
```

```
for N in $(ls)
do
  echo "Fichier -> $N"
done
```

Tester la boucle `while` et la commande `shift` avec le script, `decalage.sh`, suivant

```
while [ $# != 0 ]
do
  echo $1
  shift
done
```

et l'appeler avec `decalage.sh "a b" 'c' d e`

7.2 Exercice On suppose s'être placé dans un répertoire contenant N images numérotées de manière désordonnée et partielle. Exemple :

```
image0002.JPG   image0015.JPG   image0054.JPG   image0120.jpg
originals
image0010.jpg   image0023.jpg   image0067.jpg   image0127.JPG   toto tata
image0012.jpg   image0045.JPG   image0088.jpg   image0147.JPG
```

Le répertoire courant peut comporter d'autres fichiers, et les extensions des fichiers images peuvent être de casse incohérente (jpg et JPG).
Ecrire un script permettant de renuméroter les images de `0` à `N-1` et de fixer toutes les extensions à `jpg`.

Indications : Vous pourrez vous servir de la commande `printf` (penser à regarder le `man`).