TP n°10
Introduction à la Programmation Shell Unix

Nous avons vu lors du précédent TP les premiers éléments de la programmation Shell. Nous allons poursuivre cette semaine par l'étude de nouveaux mécanismes qui vont nous permettre d'aller plus loin dans la programmation Shell.

1 Les Structures de Contrôle

Comme dans de très nombreux langages de programmation, il est nécessaire d'avoir des instructions spécifiques pour écrire des tests, des boucles, ... La programmation Shell utilise des structures de contrôle classiques, mais il faut toujours faire très attention à la syntaxe de celle-ci qui est un peu contraignante.
On notera que pour définir les blocs d'instructions, on n'utilisera pas un marqueur spécial (comme une accolade dans beaucoup de langage) ou bien encore l'indentation (comme en python). Chaque instruction possède ses mots clés pour définir le début et la fin du bloc d'instruction.

1.1 Instructions de test et exécution conditionnelle

Dans les scripts, il est utile de faire des tests pour n'exécuter certaines parties que quand les conditions requises sont présentes.

1.1.1 L'opérateur de test : test ou [ ]

Un test est une opération dont le but est d'évaluer la valeur d'une expression. Cette expression peut être simplement l'existence de quelque chose (par exemple un fichier ou une variable), ou ce peut être une proposition.
Concrètement, cela veut dire qu'un programme n'est pas limité à la possibilité de donner des ordres : il peut aussi poser des questions, et agir comme vous le décidez en fonction des réponses apportées aux questions.
Le Shell propose deux principales façons de réaliser un test ; ces deux méthodes sont équivalentes :

```
test expression
[ expression ]
```

Attention une nouvelle fois à la manière d'écrire votre code. La programmation Shell est un peu rudimentaire dans son analyse syntaxique (la manière de « lire » ce que vous avez écrit) et ne permet pas de mettre ou de ne pas mettre des espaces n'importe où (mais vous commencez à avoir l'habitude de cela avec python et son indentation pour définir la notion de bloc d'instruction).

Il est donc impératif d'avoir un espace avant et après le crochet ouvrant et un espace avant et après le crochet fermant.

Un test renvoie un code de retour. Un code de retour est un nombre (0 ou autre), qui correspond à une réponse de type « vrai » ou « faux ». C'est ce code de retour qui permet la manipulation des tests dans des structures de contrôle que nous allons découvrir ensemble dans la suite du sujet. Le code de retour 0 correspond à la réponse « vrai ». Pour répondre « faux », le programme répond... autre chose (ce peut être 1, 2, -1 ou autre).

1.1.2 L'opérateur if

```
if conditions
then
   #1er bloc d'instructions
else
   #2nd bloc d'instructions
fi
```

```
if [ "$VAR" = "VALEUR" ]
then
        echo VAR contient bien VALEUR
else
        echo VAR ne contient pas VALEUR
fi
```

Juste après le *if*, on trouve un nombre quelconque d'instructions (pouvant être sur plusieurs lignes). Elles seront toutes exécutées jusqu'à ce qu'un *then* soit trouvé. Ensuite ce qui importera est le code de retour de la dernière instruction. Si la valeur est 0 (ce qui correspond à aucune erreur pour bash) le premier bloc d'instructions est alors exécuté. Dans le cas contraire, ce sera le second. Le else et le second bloc d'instructions sont facultatifs. Si le résultat est faux et que l'on n'a pas cette deuxième partie, rien ne sera fait. Le *fi* devra dans tous les cas être présent pour indiquer la fin du bloc de test.

1.1.3 L'opérateur case

Pour comparer une variable à plusieurs valeurs, on peut utiliser des if imbriqués (else if peut alors être abrégé en elif). Mais il y a un moyen plus simple en utilisant case *in*. Un exemple permettra de mieux comprendre son utilisation :

```
case $variable in
   valeur1)
     #1er bloc d'instructions
     ;;
   valeur2)
     #2ème bloc d'instructions
     ;;
   valeur3)
     #3ème bloc d'instructions
     ;;
   )*
     #Traitement des autres valeurs
     ;;
esac
```

Entre case et *in* se trouve la chaîne à comparer. Ensuite on a un nombre quelconque de motifs qui sont terminés par une parenthèse fermante. On peut mettre plusieurs motifs séparés par un caractère | dans le cas ou plusieurs valeurs conduisent à exécuter un bloc d'instructions. Le bloc d'instructions qui suit indique ce qu'il faut faire si la chaîne correspond à ce modèle. La fin de celui-ci est indiquée par ;; (double point-virgule).

On peut aussi indiquer un bloc d'instructions devant être exécuté si aucune correspondance n'est trouvée. Il se note par * (étoile). Il est facultatif. S'il n'est pas présent et que la chaîne à tester ne correspond à aucune des valeurs, aucun des traitements n'est effectué. Si une valeur correspond, les instructions sont exécutées et le déroulement du script se poursuit après esac qui indique la fin de ce bloc de tests.

1.2 Les boucles

On a parfois besoin de faire un même traitement plusieurs fois de suite. Pour cela bash met à disposition plusieurs types de boucles. *while* et *until* permettent cela de manière quasiment similaire et on dispose aussi de l'instruction *for*.

1.2.1 La commande while

La boucle while exécute les commandes de manière répétée tant que la première commande réussit ; en anglais, while signifie « tant que ».

```
 while conditions do
   # Traitement à répéter
done
```

Pour le while, le traitement est répété tant que la dernière des instructions précédant le mot do se déroule bien (renvoie 0).

1.2.2 La commande until

until signifie « jusqu'à ce que » en anglais. Cette boucle est le pendant de la boucle while, à ceci près que **la condition ne détermine pas la cause de la répétition de la boucle, mais celle de son interruption.**

```
until conditions do
   # Traitement à répéter
done
```

Pour le until, le traitement est répété jusqu'à ce que la dernière des instructions précédant le mot do se déroule bien (renvoie 0).

1.2.3 La commande for

Il existe un autre type de boucle différent qui s'utilise avec for. Contrairement aux précédentes, le nombre d'itérations est connu à l'avance. Sa syntaxe générale est de la forme :

```
for variable in liste_de_valeurs
do
  #Traitement à répéter
done
```

```
for i in 1 2 3 4 5
do
   echo $i
done
```

La liste suivant le *in* est constituée de plusieurs valeurs séparées par des espaces. Le traitement sera alors répété et on aura accès à *$variable* qui prendra successivement ces valeurs dans l'ordre indiqué. Dès que toutes les valeurs auront été parcourues, la boucle s'arrêtera.

2 Enchainement de commandes

Dans tous les exemples suivants, on prendra pour hypothèse que la variable x est initialisée à la valeur 1, et la valeur y est initialisée à 0.

2.1 Enchaînement simple de commandes « ; »

On notera l'utilisation de ; comme séparateur d'instructions. Si on exécute deux instructions et qu'on les met sur une même ligne (enchaînement de commandes), il faut les séparer par un ;.

```
cd ; ls
```

Ces deux commandes seront exécutées l'une après l'autre.

2.2 Enchaînement conditionnel de commandes

2.2.1. Opérateur &&

L'opérateur && n'exécute ce qui suit que si le test est vrai. Donc dans le cas présent si le fichier *fichier.txt* existe, le message « Le fichier existe » s'affiche.

```
ls fichier.txt && echo "Le fichier existe"
```

2.2.2 Opérateur ||

L'opérateur ll n'exécute la commande suivante que si le test est faux.

```
ls fichier.txt || echo "Le fichier n'existe pas"
```

Donc dans ce cas, si *fichier.txt* n'existe pas, le message s'affichera.

3 Les Opérateurs

Les instructions que nous venons de voir utilisent pour bon nombre des conditions. Nous avons vu l'opérateur *test* ou la *syntaxe [ ]* qui permettent de faire un test de condition. Mais comment écrire ces conditions et quels sont les opérateurs qui vont permettre d'écrire ces conditions ?

3.1 Les opérateurs arithmétiques

3.1.1 Tests d'égalité et d'inégalité arithmétiques

Le Shell permet d'opérer des tests arithmétiques, même s'il est moins puissant que d'autres langages. Voici les principaux opérateurs dont vous pourrez avoir besoin pour faire des comparaisons avec des nombres :

- eq (equal) : « égal à » (signe « = ») **à utiliser dans le cas où l'on cherche à tester l'égalité sur des nombres. Pour comparer une chaîne de caractère, on utilise le signe =.**
- ne (not equal) : « différent de » (signe « ≠ ») à utiliser dans le cas où l'on travaille avec des nombres. Pour des tests sur des chaînes de caractères, on utilisera le point d'exclamation comme par exemple ! $x = "" (cf opérateur logique ci-dessous).
- gt (greater than) : « strictement supérieur à » (signe « > »).
- lt (lesser than) : « strictement inférieur à » (signe « < »).
- ge (greater or equal) : « supérieur ou égal à » (signe « ≥ »).
- le (lesser or equal) : « inférieur ou égal à » (signe « ≤ »).

3.1.2 Evaluation d'expressions arithmétiques

Mais nous pouvons aussi faire des opérations mathématiques pour faire des additions, soustractions, multiplications, …
Contrairement à la plupart de langages, l'évaluation d'une expression mathématique nécessite de dire que l'on va faire un calcul. Pour cela, on utilise la commande interne ((expr)), où *expr* est l'expression mathématique que vous souhaitez calculer.

```
a=1 ; (( b = a + 1 ))
```

Dans l'exemple précédent, vous mettez le résultat du calcul de a + 1, donc 2, dans la variable b.

3.2 Les opérateurs logiques

Dans la suite des exemples, on supposera que x=1 et y=0.

3.2.1 Opérateur !

L'opérateur ! est l'opérateur logique non et donc inverse le code de retour d'une commande, c'est à dire renvoie vrai si elle renvoie faux et vice-versa. Dans le cas ci-dessous, le message « Nombre non nul » s'affichera.

```
[ ! $x -eq 1 ] || echo "Nombre non nul"
```

3.2.2 Opérateur logique ET

L'opérateur « et » renvoie 0 (vrai) si et seulement si les différentes conditions sont toutes réalisées ; si au moins l'une d'entre elles ne l'est pas, le code de retour est 1 (faux). On note cet opérateur en insérant « -a » entre les différentes conditions. Pour s'en souvenir, -a correspond à la première lettre de *and*.

```
[ "$x" -eq 1 -a "$y" -eq 0 ] && echo "Toutes les conditions sont vraies"
```

Dans le cas ci-dessus, le message « Toutes les conditions sont vraies » s'affichera.

3.2.3 Opérateur logique OU

Pour réaliser la condition de l'opérateur « ou », il suffit qu'une seule des conditions qu'il rassemble soit vraie. Si aucune des conditions incluses n'est remplie, la condition d'ensemble ne l'est pas non plus. L'opérateur « ou » est noté -o comme la première lettre du mot or. Dans l'exemple suivant, le message s'affichera.

```
[ $x -eq 1 -o $y -eq 1 ] && echo "Au moins une condition est vraie"
```

3.3 Les opérateurs sur les fichiers

Une grande partie de la puissance du Shell se déploie dans sa faculté de manipuler les commandes Unix et donc les fichiers, leur nom ou encore leur contenu. Il est donc impératif d'avoir des opérateurs qui traitent ces données.

Voici les principaux opérateurs disponibles suivant :

- nature du fichier :
  - -e (exists) : vérifie l'existence d'un fichier
  - -f (file) : vérifie l'existence d'un fichier, et le fait qu'il s'agisse bien d'un fichier au sens strict
  - -d (directory) : vérifie l'existence d'un répertoire
  - -L (link) : vérifie si le fichier est un lien symbolique
- attributs du fichier :
  - -s (size) : vérifie qu'un fichier n'est pas vide
- droits sur le fichier :
  - -r (readable) : vérifie si un fichier peut être lu
  - -w (writable) : vérifie si un fichier peut être écrit ou modifié
  - -x (executable) : vérifie si un fichier peut être exécuté
- comparaison de fichiers :
  - -nt (newer than) : vérifie si un fichier est plus récent qu'un autre
  - -ot (older than) : vérifie si un fichier est plus ancien qu'un autre

Les opérateurs s'utilisent de la manière suivante:

```
if [ -e ~/.bashrc ] ; then ... # teste si le fichier ~/.bashrc existe
```

4 Une nouvelle commande

Pour finir ce cours / TD, voici une nouvelle commande qui vous sera utile par la suite : grep. grep est un programme en ligne de commande de recherche de chaînes de caractères dans un flux de texte (dans un fichier par exemple). La commande s'utilise de la manière suivante :

```
grep [options] chaine_recherchee [fichier1 ...]
```

Le comportement habituel de *grep* est de recevoir en premier paramètre une expression rationnelle, de lire les données (dans un fichier ou une liste de fichier par exemple) et d'écrire les lignes qui contiennent des correspondances avec l'expression rationnelle.

Attention : *grep* recherche une chaîne de caractères et non un mot.

Les options les plus courantes sont les suivantes (mais cela ne dégage pas votre responsabilité de la lecture de la page de manuel)

Option	Description
-v	(*invert*) Affichage des lignes ne contenant pas la chaîne indiquée
-l	(*list files only*) Affichage uniquement des noms des fichiers contenant la chaîne recherchée. Les lignes ne sont pas affichées.
-c	(*count*) Affichage du nombre de lignes trouvées sans *affichage* de ces lignes. Si recherche dans plusieurs fichiers, affichage du nom de chaque fichier suivi du nombre correspondant.
-n	Précéder chaque ligne contenant la chaîne recherchée de son numéro dans le fichier d'entrée.
-s	Les messages d'erreur ne seront pas affichés.

TP n°10
Introduction à la Programmation Shell Unix

Exercices

Pour ce deuxième TD sur la programmation Shell, nous allons faire quelques exercices utilisant les opérateurs du langage et commencer à écrire des programmes un peu plus conséquents.

Exercice n°1:

Écrire un script Shell *oui.sh* affichant « OUI » si l'utilisateur a saisi le caractère o ou O, « NON » si l'utilisateur a saisi le caractère n ou N et « Réponse incorrecte » dans tous les autres cas.

Exercice n°2:

Ecrivez le script *info.sh* qui prend exactement un paramètre, et qui affiche des informations sur cet argument, à savoir :

- n'existe pas, si le paramètre ne correspond ni à un fichier ni à un répertoire
- est un fichier, si le paramètre est un fichier
- est un répertoire, si le paramètre est un répertoire
- est un lien, si le paramètre est un lien

Pour les fichiers et les répertoires, le script affiche également les droits dont l'exécutant du script dispose sur le paramètre :

```
$ ./info.sh filenotexisting.txt
filenotexisting.txt n'existe pas
$ ./info.sh ~/fichier.pdf
/home/stef/fichier.pdf est un fichier (rw)
$ ./info.sh /etc
/etc est un répertoire (rx)
```

Exercice n°3:

Écrire le programme Shell *dix.sh* qui affiche les dix chiffres (0 à 9), chaque chiffre étant sur une ligne.

Exercice n°4:

Recherchez si le fichier des mots de passe contient un utilisateur qui s'appelle *root*. Ecrire un script Shell *user_exist.sh* qui utilise cette commande en prenant le nom d'utilisateur en paramètre et qui affichera un message pour dire si l'utilisateur existe ou pas et renverra vrai si l'utilisateur existe sinon il renverra faux.

```
$ ./user_exist.sh root
L'utilisateur recherché a un compte sur la machine
$ echo $?
0
 $ ./user_exist.sh toto
L'utilisateur recherché n'a pas de compte sur la machine
$ echo $?
1
```