TRAVAUX PRATIQUES 4
Scripts sous Linux

L'objectif de ce TP est d'approfondir la réalisation de programmes scripts abordée lors du précédent TP sous les systèmes d'exploitation de type UNIX pour l'automatisation de traitements.
Une fois votre session ouverte sous un système d'exploitation de type UNIX, ouvrez un terminal pour entrer les commandes décrites dans l'énoncé. Dans la suite, le prompt du terminal sera symbolisé par « `$>` ».

Notes utiles de syntaxe
On rappel que dans vos programmes scripts, vous avez la possibilité de déclarer des variables (dont doit suivre la construction suivante, première lettre dans l'ensemble `[a-zA-Z]` et les caractères suivants `[a-zA-Z0-9_]`). L'affectation d'une variable se fait comme suit : `var="val"`, alors que l'accès à son contenu comme suit : `$var`.
De plus, une variable peut être supprimée de la mémoire (supprimer son existence) à l'aide la commande `unset`.

Il est possible de récupérer le résultat d'une commande afin de l'afficher ou de le stocker dans une variable, pour cela il faut utiliser les symboles $( ) comme dans l'exemple suivant affectant le résultat de `ls -l | grep toto` dans la variable `res` :

```
$> res=$(ls -l | grep toto)
```

Certains caractères ont une signification spéciale pour le shell (par exemple, `$, >, >>, <, &, *, ?, |`), pour qu'ils perdent leur signification spéciale il existe trois jeux de caractères à utiliser :

- les simples quotes : doivent être utilisées en nombre pair et retirent la signification de tous les caractères spéciaux entre deux quotes, comme dans l'exemple ci-dessous :

```
$> echo '* ? > >> < << | $PATH $(ls -l) &'
résultat : * ? > >> < << | $PATH $(ls -l) &
```

- le caractère d'échappement (ou *antislash* `\`) : retire la signification d'un caractère celui qui le suit, comme dans l'exemple suivant (où le premier antislash s'applique à l'apostrophe, le second au 3e antislash et le dernier au symbole $) :

```
$> echo L\'exemple \\\$HOME
résultat : L'exemple \$HOME
```

- les guillemets : retirent la signification de tous les symboles exceptés pour les symboles spéciaux `$, $( )` et `\`, comme dans l'exemple ci-dessous :

```
$> echo "> et | sont proteges, $HOME est substitue, $(logname) est execute et l'antislash protege le caractere \"."
résultat : > et | sont protoges, /home/steph est substitue, steph est execute et l'antislash protege le caractere ".
```

Il est possible de mettre des commentaires dans un script, tout ligne commençant par le symbole # sera considéré comme une ligne de commentaire (à l'exception de la première ligne commençant par les symboles #! suivi du chemin d'accès vers le shell à exécuter.

Code de retour
En plus des variables particulières déjà vu précédemment, il existe quelques variables qui peuvent être utiles pour certaines utilisations. Nous donnons ci-dessous certaines de ces variables avec leur signification :

Nom variable	Description
`$?`	Stocke le code retour de la dernière commande appelée, contenant un entier compris entre 0 et 255 (0 : la commande s'est exécutée correctement, sinon erreur)
`$$`	Stocke le PID (identifiant) associé au processus exécutant le script
`$!`	Stocke le PID (identifiant) du processus exécutant une commande en tâche en arrière plan (utilisation du symbole &)

Il est également possible d'utiliser la commande `exit` suivi du code retour afin d'informer le processus du déroulement du script, par exemple `exit 0` pour indiquer que tout s'est bien déroulé.

EXERCICE 1

Dans cet exercice, nous allons considérer le code retourné par une commande après son appel afin de déterminer si elle a été correctement exécutée.

Nous allons utiliser la commande `read` qui affecte dans une variable une chaîne de caractères lue sur l'entrée standard (le clavier par défaut).
Voici ci-dessous deux exemple d'utilisation de cette commande (la première utilisant une seule variable et la seconde plusieurs variables) :

```
$> read var1
une chaine entree au clavier
$> echo $var1
une chaine entree au clavier
```

ou

```
$> read var1 var2
une chaine entree au clavier
$> echo $var1
une
$> echo $var2
chaine entree au clavier
```

Remarques : La commande `read` utilise les caractères de séparation définis dans la variable d'environnement `IFS` qui contient par défaut les caractères espace, tabulation et saut de ligne.
De plus, cette commande retourne un code de retour vrai (valeur égale à 0) si une chaîne de caractères a été récupérée.

Question 1 – Réalisez un script utilisant la commande `read` pour récupérer les nom et prénom dans deux variables différentes demandés à un utilisateur. Le script doit afficher le code de retour de la commande `read` ainsi que le PID du processus exécutant le script.

```
#!/bin/sh

echo "Entrez vos nom et prenom :"
read nom prenom

echo "Votre nom est "$nom" et votre prenom est "$prenom

echo "Le code de retour de read : "$?
echo "L'identifiant du processus executant ce script est : "$$
```

EXERCICE 2

Dans cet exercice, nous allons utiliser la structure de contrôle conditionnelle afin de réaliser des traitements si une condition (formule booléenne) est vérifiée.

La commande `test` *`expression`* permet de faire des tests sur des fichiers, des chaînes de caractères ou des nombres. Cette commande renvoie un code 0 (vrai) ou 1 (faux) consultable à l'aide de la variable `$ ?` présentée à l'exercice précédent.
Il est également possible d'utiliser la syntaxe suivante qui est équivalente (avec un espace avant et après l'expression) : `[` *`expression`* `]`
Il est recommandé d'utiliser des guillemets autour des variables dans une expression, cela permet d'éviter une erreur de syntaxe retournée par `test` dans le cas où la variable est vide.

On rappel ci-dessous les divers paramètres utilisables avec la commandes `test` :

Expression	Code retour
`test -a nom_fichier`	Vrai si le fichier existe
`test -f nom_fichier`	Vrai si le fichier est de type ordinaire
`test -d nom_fichier`	Vrai si le fichier est de type répertoire
`test -h nom_fichier`	Vrai si fichier est de type lien symbolique
`test -s nom_fichier`	Vrai si fichier non vide
`test -r nom_fichier`	Vrai si le fichier est accessible en lecture
`test -w nom_fichier`	Vrai si le fichier est accessible en écriture
`test -x nom_fichier`	Vrai si le fichier est accessible en exécution
`test -z` **`ch1`**	Vrai si la chaine **ch1** est de longueur 0
`test -n` **`ch1`**	Vrai si la chaine **ch1** n'est pas de longueur 0
`test` **`ch1=ch2`**	Vrai si les deux chaines sont égales
`test` **`ch1`**`!=`**`ch2`**	Vrai si les deux chaines sont différentes
`test` **`ch1`**	Vrai si la chaine **ch1** n'est pas vide
`test` **`nb1`** `-eq` **`nb2`**	Vrai si **nb1** est égal à **nb2**
`test` **`nb1`** `-ne` **`nb2`**	Vrai si **nb1** est différent de **nb2**
`test` **`nb1`** `-lt` **`nb2`**	Vrai si **nb1** est strictement inférieur à **nb2**
`test` **`nb1`** `-le` **`nb2`**	Vrai si **nb1** est inférieur ou égal à **nb2**
`test` **`nb1`** `-gt` **`nb2`**	Vrai si **nb1** est strictement supérieur à **nb2**
`test` **`nb1`** `-ge` **`nb2`**	Vrai si **nb1** est supérieur ou égal à **nb2**

Il est possible d'utiliser des conditions plus complexes pour la commande test à l'aide de d'opérateurs logiques listés dans le tableau ci-dessous :

Opérateur de la commande `test`	Description
`!`	Négation
`-a`	ET logique
`-o`	OU logique

L'ordre d'évaluation des opérateurs peut être imposé grâce aux caractères de regroupement qui sont `\(` et `\)`.

Question 1 – Donnez la commande permettant de tester que */etc/passwd* est bien un fichier et d'afficher le résultat du test.
Même question en testant si c'est un répertoire, puis si ce n'est pas un répertoire.

Correction : Voici ci-dessous la commande :

```
$> test -f /etc/passwd
$> echo $?
```

et

```
$> test -d /etc/passwd
$> echo $?
```

et

```
$> test ! -d /etc/passwd
$> echo $?
```

Question 2 – Donnez la commande permettant de tester si vous avez uniquement les droits d'accès en lecture au fichier */etc/passwd* et d'afficher le résultat du test.

Correction : Voici ci-dessous la commande :

```
$> test -r /etc/passwd -a ! \( -w /etc/passwd –o –x /etc/passwd
\)
$> echo $?
```

Question 3 – Donnez la suite de commandes permettant de récupérer deux entiers fournit par l'utilisateur et de tester s'ils sont égaux.
Même question pour tester si le premier nombre entré est strictement supérieur au second.

Correction : Voici ci-dessous la commande :

```
$> echo "Entrez deux nombres :"
$> read $nb1 $nb2
$> test $nb1 –eq $nb2
$> echo $?
```

et

```
$> echo "Entrez deux nombres :"
$> read $nb1 $nb2
$> test $nb1 –gt $nb2
$> echo $?
```

Question 4 – Donnez la suite de commandes permettant de récupérer deux chaînes de caractères fournies par l'utilisateur et de tester si chaque chaîne possède une longueur nulle en affichant le résultat du test, puis de tester si ces deux chaînes sont ne sont pas égales.

Correction : Voici ci-dessous la commande :

```
$> echo "Entrez une premiere chaine :"
$> read $ch1
$> test -z $ch1
$> echo $?
$> echo "Entrez une deuxieme chaine :"
$> read $ch2
$> test -z $ch2
$> echo $?
$> test "$ch1"!="$ch2"
$> echo $?
```

EXERCICE 3

Le shell offre la possibilité d'exécuter une série de commande de façon séquentielle en tenant compte du code de retour des précédentes commandes.
Deux opérateurs sont utilisables && (ET logique) et || (OU logique) ayant le comportement suivant :

- `commande1 && commande2` : la deuxième commande est uniquement exécutée si la première commande renvoie un code de retour à vrai (donc 0) et l'expression globale renvoie vrai si toutes les commandes ont renvoyé un code de retour à vrai,
- `commande1 || commande2` : la deuxième commande est uniquement exécutée si la première commande renvoie un code de retour à faux (donc >0) et l'expression globale renvoie vrai si au moins l'une des commande a renvoyé un code de retour à vrai.

Question 1 – En utilisant les opérateurs décrits ci-dessus, donnez la commande permettant d'afficher le contenu du fichier */etc/passwd* si vous avez pu créer un fichier *test.txt* dans le répertoire */etc*.
Même question mais dans le cas où vous n'avez pu créer le fichier *test.txt* affiche le contenu du fichier */etc/passwd*.

Correction : Voici ci-dessous la commande :

```
$> touch /etc/test.txt && cat /etc/passwd
```

et

```
$> touch /etc/test.txt || cat /etc/passwd
```

EXERCICE 4

Dans cet exercice, nous allons utiliser la commande `test` avec la structure de contrôle `if` afin de tester si une condition logique est respectée ou non, comme suit :

```
if commande1
then
      commande2
      commande3
else
      commande4
      commande5
fi
```

Dans le cas où la commande `test` (ou plus généralement une commande) en paramètre de **`if`** est vrai (c'est-à-dire `$?` vaut 0) alors les commandes situées après le **`then`** sont exécutées, sinon ce sont les commandes après le **`else`** qui le seront (le bloque **`else`** n'est pas obligatoire).

Question 1 – Donnez un script qui test le nombre d'arguments passé au lancement du script. Si au moins un argument a été fournit alors le script affiche la liste des arguments, sinon un message d'erreur est retourné.

Correction : Voici ci-dessous le script :

```
#! /bin/sh

if test $# –ge 1
then
      echo "voici la liste des arguments fournis :"
      echo $*
else

      echo "Usage : $0 arg1 arg2 …"
fi
```

Question 2 – Donnez un script qui test si un fichier passé en argument possède les droit d'exécution, si c'est le cas alors le fichier est exécuté sinon le droit en exécution est ajouté avant d'exécuter le fichier. Vous testerez un nom de fichier est passé en argument du script.

Correction : Voici ci-dessous le script :

```
#! /bin/sh

if test $# –ne 1
then
      echo "Usage : $0 arg1 arg2 …"
fi

if test –x $1
then
      $PWD/$1
else
      chmod +x $1
      $PWD/$1
fi
```

Lorsqu'il est nécessaire de comparer le contenu d'une variable avec différentes valeurs, il est possible d'utiliser la structure contrôle `case` qui permet de remplacer une suite de bloc `if else`.
Celle-ci s'utilise la manière suivante :

```
case $var in
    val1) commande
      …
```

```
        ;;
    val2) commande
        …
        ;;
    val3 | val4 | val5) commande
        …
        ;;
esac
```

Le shell évalue le contenu de la variable avec les différentes valeurs indiquées de haut en bas, si il y a égalité alors les commandes relatives à cette valeur sont exécutées. Les caractères `;;` représentent la fin du traitement et permettent de sortir du case. La prochaine commande exécutée est celle après `esac`. Il est possible d'indiquer une suite d'alternatives de valeurs associées au même bloc de commandes à exécuter, pour cela il faut utiliser le symbole `|` entre les valeurs.

Question 3 – Donnez un script qui utilisant un case permettant d'afficher le menu suivant :

1 – Afficher login
2 – Afficher pwd
3 – Fin

puis en fonction du nombre entré par l'utilisateur, le script affiche l'information choisie.

Correction : Voici ci-dessous un exemple de script réalisant cela :

```
#! /bin/sh

echo "1 – Afficher $USER"
echo "2 – Afficher $PWd"
echo "3 – Fin"

read reponse

echo "Votre choix est : "$reponse

case "$reponse" in
    1) echo "\$USER="$USER
       ;;
    2) echo "\$PWD="$PWD
       ;;
    3) echo "Au revoir!"
       ;;
esac
```

EXERCICE 5

Le shell permet de réaliser des calculs sur des nombres, pour cela chaque expression arithmétique doit être entourée par les symboles suivants `((` et `))`.
Le tableau suivant liste les principaux opérateurs utilisables :

Opérateurs	Description
nb1 + nb2	Addition
nb1 - nb2	Soustraction

nb1 * **nb2**	Multiplication
nb1 / **nb2**	Division
nb1 % **nb2**	Modulo
nb1 = expression	Affectation

Question 1 – Donnez la suite des commandes pour créer une variable x en lui affectant la valeur 10, puis doubler son contenu et afficher sa nouvelle valeur.

Correction : Voici ci-dessous la suite de commandes :

```
$> x=10
$> ((x=$x*2))
$> echo $x
```

Question 2 – Donnez un script qui prend en paramètre un entier x, calcul x^3 et affiche le résultat du calcul.

Correction : Voici ci-dessous la suite de commandes :

```
#! /bin/sh

X=$1
((x=x*x*x))
echo "Le resultat du calcul est : "$x
```