Système d'Exploitation
Travaux Pratiques (3), Licence 2 Informatique
Initiation aux Scripts Shell

Les exercices suivants ont pour but de vous initier à l'écriture de scripts shell.
Il vous est recommandé de consulter les pages `man` du `shell` pour de plus amples informations sur les variables, les commandes, leur syntaxe et leur sémantique.

Les instructions des exercices se repèrent par des icônes, qui sont les suivantes :

	Information	Information concernant l'usage ou le rôle d'une commande, par exemple. Dans certains cas, il s'agit d'une information sur ce que vous êtes en train de faire ou sur ce qui se passe.
	Exemple	Exemple d'utilisation.
	Contrôle	Vérifier le résultat d'une (ou plusieurs) action(s).
	Action	Effectuer la ou les action(s) décrite(s).
	Question	Questions auxquelles vous devez répondre.

De plus, un `texte en police courier` correspond soit à une sortie écran soit à des noms spécifiques (menus, fenêtre, icône, processus, commandes…).
Un **texte en police times gras** correspond à ce que l'utilisateur doit introduire comme valeur de paramètre, ou encore, est utilisé pour attirer l'attention de l'utilisateur.

Les variables

Les variables d'environnement, qui sont des chaînes de caractères, sont des variables dynamiques utilisées par les différents processus d'un système d'exploitation. Par exemple, la variable d'environnement `SHELL` contient le shell par défaut de l'utilisateur. La commande `env` permet d'afficher la liste des variables d'environnement.

L'affectation se fait ainsi : `nom_variable=valeur`.

On accède à la valeur associée à la variable d'environnement `nom_variable` en spécifiant `$nom_variable`.

Mohamed Taghelit , taghelit@univ-tours.fr

Répondez aux questions suivantes :

- Quel est votre `shell` par défaut ?
- Quels sont les autres `shells` disponibles sur votre système ?

Exécutez les commandes suivantes et expliquez les résultats obtenus (comment le `shell` interprète les délimiteurs " et ' ?

```
$ NOM=toto
$ echo NOM
$ echo $NOM
$ echo "$NOM"
$ echo '$NOM'
$ TEXTE=bonjour $NOM
$ TEXTE=bonjour_$NOM
$ echo $TEXTE
$ TEXTE="bonjour $NOM"
$ echo $TEXTE
$ TEXTE='bonjour $NOM'
$ echo $TEXTE
$ echo bonjour $ NOM
$ echo bonjour $NOM
$ echo bonjour \$NOM
```

Exécutez les commandes suivantes et expliquez les résultats obtenus (comment le `shell` interprète le délimiteur ` (backquote) ?

```
$ cmd1=`pwd`; qui=`whoami`
$ echo $cmd1; echo $qui
$ cmd2=`ls`
$ echo $cmd2
$ cmd3=`toto`
$ echo $cmd3
```

Exécutez et comparez les commandes `echo `pwd`` et `echo $(pwd)`. Que constatez-vous ?

Les scripts shell

Un script est un fichier textuel contenant des commandes `shell`. Ce fichier peut être exécuté (à condition que l'utilisateur possède le droit `x` pour ce fichier), c'est à dire, lancé dans le `shell` comme s'il s'agissait d'une commande. Les scripts obéissent à quelques conventions:

- ils contiennent des séquences de commandes et les retours à la ligne sont interprétés comme des points virgules,

- ils commencent par une ligne permettant d'identifier le `shell` à utiliser pour les exécuter. Pour `bash`, la première ligne doit être `#!/bin/bash`,
- toute portion de ligne apparaissant à droite d'un caractère `#` est considérée comme un commentaire,
- pour pouvoir exécuter un script, l'utilisateur doit posséder le droit de lecture (`r`) et d'exécution (`x`) sur le fichier.

Créer un script nommé `bonjour.sh` qui affiche le texte "`Bonjour tout le monde !`".

Ajouter au script précédent, un commentaire indiquant l'auteur du script et la date de création.

Modifier le script précédent de façon qu'il affiche maintenant "`Bonjour nom_connexion`" si votre nom de connexion est `nom_connexion` (login name).

Les paramètres d'un script

Les scripts que vous réalisez peuvent accepter des paramètres, comme toute commande habituelle. Les paramètres sont des chaînes de caractères séparées par des blancs, qui apparaissent en ligne de commande après le nom du script au moment de son appel. À l'intérieur du script, on se réfère à ces paramètres par les variables spéciales `#` (qui contient le nombre de paramètres passés au script), `0` (contenant le nom du script), `i` (compris entre `1` et `$#`, contient le $i^{ème}$ paramètre) et `@` (contenant la liste de tous les paramètres séparés par des espaces). Dans certains cas, on ne connaît pas à l'avance le nombre de paramètres qui seront reçus par le script. On peut donc utiliser la commande `shift`, qui permet de décaler tous les paramètres d'un cran vers la gauche (après l'appel de `shift`, le $i^{ème}$ paramètre est obtenu par `${i-1}`).

Créez un script nommé `affich_param` contenant les instructions suivantes :

```
#!/bin/bash
echo '$0 : ' $0
echo '$# : ' $#
echo '$* : ' $*
echo '$@ : ' $@
echo '$1 : ' $1
echo '$2 : ' $2
echo '$3 : ' $3
shift
echo "Après le premier appel de la fonction shift:"
echo '$1 est devenu: ' $1
shift
echo "Après le second appel de la fonction shift:"
echo '$1 est devenu: ' $1
```

Invoquez votre script des manières suivantes:

```
./affich_param
./affich_param param_1
./affich_param param_1 param_2
./affich_param param_1 param_2 param_3
```

Les expressions arithmétiques

L'évaluation d'une expression arithmétique permet de remplacer une expression par la valeur de son résultat. Cette évaluation est réalisée à l'aide de la notation `$((...))` (ou par `$[...]`).

Exécutez les commandes suivantes et étudiez les résultats.

```
echo 1+2
echo $((2*(1+2)))
un=1
deux=2
echo $un+$deux
echo $(($un+$deux))
echo $((un+deux))
```

Écrivez un script qui calcule et affiche la moyenne de deux notes données en paramètres.

La commande `read` permet de lire des données au clavier (en fait sur le flux d'entrée standard), et de les stocker dans une variable. Sa syntaxe est la suivante: `read variable`.

Écrivez un script qui lit deux entiers, calcule et affiche leur moyenne.

Code de retour des commandes

Chaque commande informe son environnement du succès ou de l'échec de son exécution à l'aide d'un code de retour entier, stocké dans la variable '`?`'.

Quelle commande renvoie le code retour de la dernière commande exécutée ?

Un code de retour égal à `0` signifie que la commande a réussi, et (souvent) un code strictement positif signifie qu'elle a échoué.

Instructions

Le langage d'un shell, tout comme d'autres langages de haut niveau, possède des instructions et des structures de contrôle qui orientent l'exécution des commandes en fonction des conditions souhaitées.

- **Les instructions conditionnelles et de choix**

Les instructions conditionnelles ont la syntaxe suivante :

```
if COMMANDES_DE_TEST; then
COMMANDES_A_EXECUTER_SI_TEST_S_EVALUE_A_VRAI
fi
```

ou :

```
if COMMANDES_DE_TEST; then
COMMANDES_A_EXECUTER_SI_TEST_S_EVALUE_A_VRAI
else
COMMANDES_A_EXECUTER_SI_TEST_S_EVALUE_A_FAUX
fi
```

ou :

```
if COMMANDES_DE_TEST; then
COMMANDES_A_EXECUTER_SI_TEST_S_EVALUE_A_VRAI
elif AUTRES_COMMANDES_DE_TEST; then
COMMANDES_A_EXECUTER_SI_AUTRE_TEST_S_EVALUE_A_VRAI
else
COMMANDES_A_EXECUTER_SI_AUTRE_TEST_S_EVALUE_A_FAUX
fi
```

Lorsque le nombre de choix possibles devient important cela implique des “cascades” de `if-else` de grande profondeur, et le code devient rapidement illisible. La structure `case` permet d'éviter ce problème :

```
case EXPRESSION in
CASE1) LISTE_DE_COMMANDES ;;
CASE2) LISTE_DE_COMMANDES ;;
...
CASEn) LISTE_DE_COMMANDES ;;
esac
```

- **Primitives de tests**

On utilise fréquemment comme tests des commandes de la forme `[[ expression ]]` (notez bien que les espaces à l'intérieur de la condition sont obligatoires entre les crochets, les opérateurs et les arguments), où expression est formée d'un ensemble de primitives de test. Les primitives existantes permettent entre autre de comparer des chaînes de caractères ou des nombres, de tester l'existence d'un fichier, etc ...

L'exemple de code suivant :

```
if [[ -e fic && ( ! -x fic ) ]]; then
echo "C'est OK !"
fi
```

affiche "`C'est OK!`" si le fichier `fic` existe et n'est pas exécutable (`-e` teste l'existence, `&&` signifie "et", `-x` teste si `fic` est exécutable, et `!` exprime la négation).

Le test `[[ "$reponse" != "jaune" ]]` teste si la valeur de la variable `reponse` est différente de la chaîne "`jaune`".

- **Tests triviaux**

  Les commandes `true` et `false` peuvent aussi servir comme commandes de test, leur code de retour vaut toujours `0` et `1` respectivement.

- **Opérateurs de comparaison numériques**

  `-eq` (égal), **`-ne`** (différent), **`-lt`** (inférieur), **`-le`** (inférieur ou égal), **`-gt`** (supérieur), **`-ge`** (supérieur ou égal).

- **Opérateurs de comparaison de chaînes**

  `-z` (chaîne vide ou non définie), **`-n`** (chaîne non vide), **`=`** (chaînes identiques), **`!=`** (chaînes différentes).

- **Opérateurs de test de fichiers**

  `-L` (lien symbolique), **`-d`** (répertoire), **`-e`** (existe), **`-f`** (fichier ordinaire), **`-s`** (fichier non vide), **`-r`** (fichier lisible), **`-w`** (fichier modifiable), **`-x`** (fichier exécutable), **`-nt`** (plus récent que), **`-ot`** (plus ancien que).

  Les opérateurs de fichiers les plus utilisés sont **`-f`** et **`-d`**, à savoir si la variable est bien un fichier ou un répertoire.

- **Répétitions**

  Les actions de répétition peuvent être exprimées en `bash` de 3 façons :

  La boucle `for` provoque l'exécution des `COMMANDES` pour chaque valeur apparaissant dans la `LIST` et attribuée une à une à la `VARIABLE`.

  ```
  for VARIABLE [in LIST]; COMMANDES; done
  ```

  Par exemple, la boucle suivante effectue une copie de sauvegarde pour chaque fichier `.xml` contenu dans le répertoire courant, la liste des fichiers à copier étant le résultat de l'évaluation de la commande `ls *.xml` :

  ```
  for i in `ls *.xml`; do cp "$i" "$i".bck; done
  ```

  La boucle `while` provoque l'exécution des `COMMANDES` tant que la `CONDITION` se termine avec succès.

  ```
  while CONDITION; do COMMANDES; done
  ```

  Par exemple la boucle suivante affiche les paramètres du script dans lequel elle se trouve :

  ```
  i="1"
  while [ "$1" != "" ]; do
  echo "Parametre numero $i = " $1
  shift
  ```

```
i=$[$i+1]
done
```

La boucle `until` est très semblable à la boucle `while` mais, contrairement à cette dernière, elle provoque l'exécution des `COMMANDES` tant que la `CONDITION` se termine avec échec.

```
until CONDITION; do COMMANDES; done
```

Par exemple la boucle suivante fait la même chose que celle de l'exemple ci-dessus, mais en utilisant la construction `until` à la place de `while` :

```
i="1"
until [ "$1" == "" ]; do
echo "Parametre numero $i = " $1
shift
i=$[$i+1]
done
```

La commande `break` permet de quitter la boucle la plus interne. La commande `continue` permet d'interrompre l'itération courante d'une boucle et de passer à l'itération suivante.

Ci-dessous un exemple de script :

```
#!/bin/bash
#Affichage des informations de base concernant
#l'administration système.
#Le choix de l'affichage se fait par un menu.
#Aucun paramètre d'entrée
#Affichage du menu
echo "Que voulez-vous connaître ?"
echo "1 - données sur votre machine"
echo "2 - données sur votre environnement"
echo "3 - quitter"
#Saisie du choix
read choix
#Exécution en fonction du choix
case $choix in
1)
echo "Vous travaillez sur la machine `hostname`. "
echo "Elle est basée sur l'architecture `arch`."
;;
2)
echo "Votre système d'exploitation est `uname`."
echo "Votre répertoire courant est `pwd`."
Echo "Il occupe l'espace mémoire de `du -hs`"
;;

3)
echo "Au revoir..."
exit 0;;
```

```
*)
echo "Mauvais choix!"
exit 1
;;
esac
```

Saisissez le script ci-dessus et exécutez-le plusieurs fois sans aucun paramètre. Observez son résultat en fonction de la saisie au clavier.

Observez son contenu et comprenez le fonctionnement de la structure case.

Complétez le script précédent afin qu'il donne un choix supplémentaire. Il correspondra à l'affichage d'informations vous concernant : votre identité (`whoami`), votre `UID` (`id -u`), les groupes dont vous êtes membre (`groups`) et votre `GID` (`id -g`).

Autre

Il est possible de modifier l'environnement de travail par défaut en le personnalisant. Vous pouvez définir ces changements par l'intermédiaire des fichiers de démarrage. Il est ainsi possible, par exemple, de configurer le prompt (l'invite de commande) pour qu'il soit plus utile. Il existe un certain nombre de caractères spéciaux qui permettent de le modifier. Parmi eux (consultez le manuel pour une liste exhaustive) :

- `\h` : le nom de la machine jusqu'au premier '.'
- `\H` : le nom de la machine
- `\t` : l'heure courante (24 H) au format `HH:MM:SS`
- \T : l'heure courante (12 H) au format `HH:MM:SS`
- `\u` : le nom de l'utilisateur courant

Modifiez votre prompt de façon qu'il affiche : votre nom, le nom de la machine, l'heure courante et le répertoire courant.

Modifiez votre prompt tel que décrit ci-dessus de façon que chaque champs soit d'une couleur différente.

Faites en sorte que les changements précédents soient définitifs (pris en compte à chaque nouvelle session).