Introduction aux systèmes d'exploitation (IS1)

TP 9 – Structures de contrôle et scripts bash

2007

1 Structures de contrôle

Vous avez déjà vu, dans le tp 7, la structure de contrôle conditionnelle *if then else*, dont nous vous rappelons la syntaxe :

```
if commandes1 ; then commandes2 ; else commandes3 ; fi
```

Si la valeur de retour de la liste de commandes *commandes1* est 0, alors *commandes2* est exécuté sinon *commandes3* est exécuté.

Nous vous rappelons également que pour réaliser un test ou une comparaison sur des variables ou des fichiers, on utilise la commande *test*.

Nous allons maintenant étudier d'autres structures de contrôle du shell :

1.1 Boucle for

La boucle *for* permet d'exécuter successivement une suite de commandes pour chaque valeur que l'on donne à un paramètre. Voici deux syntaxes possibles :

La première syntaxe a la forme suivante :

```
for var in liste ; do commandes ; done
```

Ici, le paramètre est une variable (et *var* est son nom), *liste* une liste d'*expressions régulières* séparées par des espaces et *commandes* une liste de commandes. Lors de l'exécution, *var* prend successivement chaque valeur apparaissant dans *liste*, et *commandes* est executée pour chacune de ces valeurs. Voici un exemple de boucle :

```
for prenom in Yvonne J{ean{ne,},acques} ; do echo Bonjour $prenom ; done
```

La sortie produite par cette boucle sur le terminal est :

```
Bonjour Yvonne
Bonjour Jeanne
Bonjour Jean
Bonjour Jacques
```

Notez que la liste de valeurs peut contenir également des expressions régulières du bash. Les valeurs correspondantes seront alors les fichiers représentés par les chemins donnés par ces expressions régulières.

Par exemple,

```
for fichier in *.java ; do echo $fichier ; done
```

est équivalent à :

```
ls *.java
```


Il est également possible d'utiliser une syntaxe plus proche d'autres langages, tels Java, lorsque le paramètre est une variable prenant des valeurs entières :

```
for (( expr1 ; expr2 ; expr3 )) do commandes ; done
```

Ici, *expr1*, *expr2* et *expr3* sont des expressions arithmétiques comme nous en avons déjà rencontrées. À l'initialisation de la boucle, *expr1* est évaluée. En fin de boucle, si *expr2* est évaluée à 0 alors *expr3* est évaluée et un nouveau tour de boucle commence. Sinon, la boucle se termine. Voici un exemple utilisant cette syntaxe :

```
for (( i=1 ; i <= 1024 ; i*=2 )) do echo $i ; done
```

La sortie produite par cette boucle est la suite des puissances de 2 jusqu'à 2^{10}.

Exercice 1 – *Échauffement.*

1. Ecrivez une boucle `for` qui affiche le message « Bonjour ! » indéfiniment.
   Est-ce possible avec les deux types de boucles `for` ?
2. Ecrivez une boucle `for` qui affiche le nombre de fichiers et sous-répertoires visibles dans le répertoire courant.
3. Ecrivez une boucle `for` qui attend un entier n donné par l'utilisateur et affiche la valeur de 2^n. (Indication : utilisez la commande *read*.)
   Que se passe-t-il quand vous donnez un n "trop grand" ?

1.2 Boucles `while` et `until`.

La boucle `while` ou « tant que » suit la syntaxe suivante :

```
while condition ; do commandes ; done
```

Dans une boucle while, la liste de commandes *commandes* est exécutée de façon répétée tant que la commande *condition* donne une valeur de retour égale à 0.

La boucle `until` ou « jusqu'à ce que » fonctionne de manière identique, si ce n'est que la liste de commandes est exécutée jusqu'à ce que la valeur de retour de la condition soit égale à 0 :

```
until condition ; do commandes ; done
```

Exercice 2 – *Échauffement.*

1. Ecrivez une boucle `while` et une boucle `until` qui affichent le message « Bonjour ! » indéfiniment.
2. Ecrivez à l'aide d'une boucle `while` puis `until` une commande qui demande un nombre entier à l'utilisateur et affiche toutes les puissances de 2 inférieures à ce nombre.
   Quelle est l'option de `read` permettant d'afficher une question dans le terminal, avant de lire la réponse donnée par l'utilisateur ?

1.3 `case`

Comme dans tout langage de programmation, on peut être amené à vouloir raisonner par cas sur la valeur d'une variable. La structure de contrôle `case` le permet. En voici la syntaxe :

```
case $var in
     v1) commandes1 ;;
   v2|v3) commandes2 ;;
      *) commandes3 ;;
 esac
```

Si la valeur de la variable *var* est *v1* alors la suite de commandes *commandes1* est exécutée. Si la valeur de *var* est *v2* ou *v3* alors la suite de commandes *commandes2* est exécutée. Si *var* a une autre valeur alors la suite de commandes *commandes3* est exécutée.

Notez le double point-virgule pour terminer chaque liste d'instructions.

Voici, en exemple, une commande permettant d'afficher le nombre de jours que possède le mois correspondant à la valeur de la variable *mois*, le cas échéant (et affichant un message d'erreur, sinon) :

```
case $mois in
     2) echo 28 ou 29 jours;;
     1|3|5|7|8|10|12) echo 31 jours;;
     4|6|9|11) echo 30 jours;;
     *) echo "$mois n'est pas le numéro d'un mois";;
 esac
```

Exercice 3 – *Échauffement.*

1. Ecrivez, à l'aide d'un `case`, une commande demandant à l'utilisateur un caractère ($+$ ou $*$), puis deux entiers, et rendant, respectivement, l'addition ou la multiplication de ces deux entiers selon que le premier caractère est $+$ ou $*$.
2. Ecrivez, à l'aide d'un `case`, une commande donnant en français et en toutes lettres, l'actuel jour de la semaine.
3. Ecrivez, à l'aide d'un `case`, une commande rendant le type (fichier régulier, répertoire, tube, etc...) du fichier désigné par le chemin rentré par l'utilisateur.

2 Scripts

Pour éviter d'avoir à ré-écrire des commandes complexes que l'on utilise souvent, il est préférable de les enregistrer dans des fichiers que l'on pourra exécuter quand il nous plaira. On appelle ces fichiers des `scripts`.

2.1 Règles

- Un script contient des séquences de commandes telles que l'on pourrait les taper dans un terminal. Les commandes successives peuvent être séparées par des retours à la ligne (qui sont interprétés comme des points virgules).
- Tout fichier de script commence par une ligne permettant d'identifier le programme qui doit être utilisé pour l'exécuter. Dans le cas d'un script `bash`, la première ligne du fichier doit contenir : `#!/usr/local/bin/bash` (vérifiez le chemin de votre `bash` avec la commande `which bash`)

Le fonctionnement minimal souhaité du script est le suivant :

- Si l'utilisateur entre la commande `quizz.sh -v` *`ville`*,
  le script demande le nom du pays dont la capitale est *`ville`* et vérifie la réponse .
- Si l'utilisateur entre la commande `quizz.sh -p` *`pays`*,
  le script demande le nom de la ville qui est capitale de *`pays`*, puis vérifie la réponse.

1. Écrivez une commande qui lit une chaine de caractères contenant une virgule, et initialise deux variables `element1` et `element2` avec le contenu de la chaîne avant et après la virgule.

Vous pourrez utiliser pour cette question la commande `cut` ou une expansion bash du genre `${...#...}`.

2. Écrivez le script `quizz.sh` remplissant les fonctionnalités minimales.

Vous pourrez utiliser la commande `grep` pour récupérer la ligne du fichier `capitales` correspondant à un pays ou une ville donnés.

3. En cas d'erreur de l'utilisateur dans le format des paramètres, faites afficher au script un message d'aide rappelant la syntaxe.

4. Votre script utilise le fichier `capitales.txt` pour fonctionner. Décidez d'un endroit où stocker ce fichier, et faites afficher un message d'erreur par le script si le fichier est absent.

5. *Bonus :* Si vous avez fini le reste du tp, vous pouvez rajouter des fonctionnalités plus avancées à votre script. Voici quelques suggestions (ne faites pas tout !) :
- ignorer les accents et majuscules dans la vérification de la réponse ;
- poser plusieurs questions à la suite (nombre fixé à l'avance ou choix du type « continuer (o/n) ? ») ;
- afficher un score, l'enregistrer dans un fichier ;
- si le deuxième argument est omis, tirer une ligne au hasard ;
- question difficile : poser des questions pour toute la liste dans le désordre (sans doublons !) ;
- ajouter des orthographes alternatives pour les pays et les villes ;

Exercice 6 – *Du ménage*

Certains programmes génèrent des fichiers temporaires, de sauvegarde, etc ... que l'on utilise jamais, sauf à la suite d'un plantage. Par exemple :

- Lors de l'ouverture d'un fichier `fic` avec `emacs`, ce dernier commence par en faire une copie de sauvegarde qu'il nomme `fic~`. Après avoir fini de travailler (sans plantage) sur ces fichiers, vous pouvez en supprimer les vieilles versions. De même, `emacs` créé un fichier de sauvegarde `#fic#`, qui ne disparaît pas tout le temps (par exemple, après exécution de `killall -9 emacs`).
- `latex`[1], un outil de rédaction de documents très puissant, génère des fichiers possédant les extensions `.log, .aux, .bbl`

1. Quelle ligne de commande pouvez-vous taper pour supprimer tous les types de fichier précédemment cités se trouvant *dans le répertoire courant* ? On préférera une ligne de commande la plus courte possible, utilisant les expansions de `bash`.

Mettez cette commande dans un script `nettoie` qui constituera la base de travail.

Nous allons rendre ce script plus souple en lui rajoutant des paramètres et des options.

2. Ajoutez un argument qui indique le type des fichiers à supprimer. Par exemple `nettoie emacs` supprime uniquement les fichiers de type `fic~` et `#fic#`. Ce paramètre pourra prendre la valeur `emacs, log, latex, tout`, ce dernier supprimant tous les types de fichier effaçables.

[1] que nous utilisons pour rédiger ces feuilles de TP

3. Avant ce paramètre, rajoutez un argument (une option) qui indique les options à passer à la commande `rm`. Ses valeurs seront les mêmes que les options de `rm` à savoir : `-i` (*interactive*) qui indique que l'on doit confirmer toutes les suppressions de fichiers, et `-f` (*force*) indique que tout doit être fait silencieusement (pas de confirmation, peu de message d'erreur).

4. Pour l'option `-f`, redirigez les messages d'erreurs de `rm` vers `/dev/null` pour les rendre invisibles à l'utilisateur. Est-ce une bonne idée de le faire pour l'option `-i` ?

5. enfin, dans le cas où l'utilisateur aurait donné des arguments erronnés, affichez un message d'erreur indiquant comment utiliser ce script.

(bonus) Voici maintenant quelques questions supplémentaires pour le plaisir :

6. Si vous ne l'avez pas déjà fait, ajouter au fichier `.bashrc` l'extension de la variable `PATH` pour qu'il prenne en compte . Faites le en sachant que vous ne connaissez pas a priori la valeur de `PATH` (elle change selon les sytèmes).

7. Rajoutez des fonctionalités à votre script. La plus importante d'entre-elle est le nettoyage *récursif* de vos répertoires. C'est à dire que les sous-répertoires contenus dans le répertoire courant doivent eux aussi être nettoyés. Pour cela, vous pouvez explorer deux voies :
- appeler le script à l'intérieur de lui-même. Pourquoi est-ce une bonne idée d'avoir effectué la première question bonus d'abord ?
- utiliser la commande `ls -R`, une boucle `for ... in ...` et *deux* variables internes.