Programmation de scripts

1 Principes généraux

La programmation d'un script consiste à écrire un programme en langage shell (ou tout autre langage de script) dans un fichier texte. Lorsqu'un programme (et donc un script en particulier) est lancé à partir d'un interpréteur de commandes, il hérite d'un environnement de variables défini dans le contexte de cet interpréteur de commandes.

Ces variables (exemple : `PATH, HOME, USER, ...`) peuvent être utilisées par tout script. Par ailleurs, un ensemble de paramètres dont le nom est prédéfini peuvt être utilisé dans un script :

`$n`	n-ième paramètre d'appel du script (`$0` est le nom du fichier script lui-même)
`$*`	ensemble des paramètres sous forme d'une seule chaîne à partir de `$1` (ne contient pas `$0`)
`$#`	nombre de paramètres
`$$`	numéro du processus exécutant le script
`$?`	code de retour de la dernière commande exécutée

Rappel Le signe `$` devant un nom de variable (ici un nom de paramètre) désigne sémantiquement la valeur de la variable. Par ailleurs, les paramètres sont des paramètres d'entrée (ils ne peuvent être affectés).

2 Les structures de contrôle et leur usage

Il existe 2 groupes de structures de contrôle :

- celles permettant de contrôler l'exécution des commandes selon la valeur d'une variable : boucle `for` et le branchement conditionnel `case` ;
- celles permettant de contrôler l'exécution des commandes selon le bon déroulement ou non de certaines commandes : forme conditionnelle `if` et boucle `while`.

Le langage ne comporte qu'un seul type de variable. En effet, les variables sont **toujours** des chaînes de caractères. Il est cependant nécessaire de pouvoir évaluer des expressions booléennes pour les structures de contrôle conditionnelles (`if`, `while`, `until`). Pour ce faire, toute commande renvoie un code de terminaison sous la forme d'un entier. Cet entier résultat de la commande est interprété comme la valeur vraie s'il est égal à 0 et comme la valeur fausse s'il est différent de 0. Par ailleurs, certaines commandes utilitaires permettent l'évaluation de conditions diverses sur les objets fichiers et/ou des valeurs de variables représentant des entiers : `expr`, `test`, ...

C'est pourquoi, on voit aussi apparaître en position d'expression booléenne une liste de commandes dans les structures de contrôle conditionnelles. La forme de cette liste peut être une suite de commandes reliées par l'opérateur séquentiel `;` , mais aussi `||` ou `&&`, permettant de construire des expressions composées.

Exemples d'expressions booléennes

— Composition par un `&&` : `[ -e $f ] && [ -d $f ]` est vrai si le fichier désigné par $f existe et est un répertoire.

— Composition par un `||` : `[ -e $f1 ] || [ -e $f2 ]` est vrai si l'un des 2 fichiers désignés existe.

Cette syntaxe peut éviter un `if` explicite, par exemple : `[ -e $f ] || touch $f && echo "$f existe`"

2.1 La boucle `for`

syntaxe :

`for` variable `in` liste de valeurs
`do` liste de commandes
`done`

Exemple

```
for f in `ls`
do
   cat $f | wc -l
done
```

2.2 Le branchement conditionnel `case`

syntaxe :

`case` variable `in`
  motif `)` liste de commandes
`;;`
  ...
`esac`

Exemple

```
case $2 in
  -l ) list=on ;;
  -d ) debug=on ;;
  -r ) recursif=on ;;
  * ) ;;
esac
```

2.3 La condition `if`

syntaxe :

`if` liste de commandes *condition*
`then` liste de commandes *then*
`else` liste de commandes *else*
`fi`

Exemple

```
if [ -f $1 ] && [ -d $2
]
then cp $1 $2
else echo "usage :..."
fi
```

2.4 La boucle `while`

syntaxe :

`while` liste de commandes *condition*
`do`
  liste de commandes *bloc*
`done`

Exemple

```
while [ $1 ]
do
  cp $1 projet
shift
done
```

Remarque Cette boucle équivaut à une boucle `for` sur la liste des paramètres `$*`. En effet, la commande `shift` décale tous les paramètres (à partir de 1) d'une position vers la gauche : le paramètre `$i`, pour 0 <`i` < `$#` reçoit la valeur du paramètre `$i+1`. Le paramètre `$#` devient non défini, et la longueur de la liste de paramètres a donc diminué de 1. L'ancienne valeur du paramètre `$1` est perdue.

2.5 La boucle `until`

syntaxe :

`until` liste de commandes *condition*
`do`
  liste de commandes *bloc*
`done`

3 Fonctions et leur usage

Il est possible d'utiliser des fonctions dans un script shell. Malheureusement les différentes versions de shell n'utilisent pas la même syntaxe. En bash, une fonction est déclarée grâce au mot clé `function`. Le corps de la fonction peut être défini entre accolades ou parenthèses :

- si l'on utilise des accolades, la fonction ne possèdera pas de variables locales. Toute variable est implicitement globale exceptés les paramètres `$0,$1,$2, ...`
- si l'on utilise des parenthèses, il y aura création de variables locales au bloc parenthésé. Toute variable définie (affectée) dans le bloc sera locale [1].

En shell de base, une déclaration de fonction est détectée par un suffixe () concaténé au nom de la fonction. La description du corps de la fonction reste identique au cas précédent.

Comme toute commande, une fonction renvoie un entier qui par défaut est égal à 0. La commande `return` permet de fixer la valeur résultat. L'imbrication des fonctions est autorisée.

Quelques syntaxes ≪ bash ≫ possibles :

`function` nom {
liste de commandes
}
	`function` nom
	`(` liste de commandes `)`
		`function` nom
		`(`
		liste de commandes
		`)`

Note : Pour la déclaration en shell de base, il suffisait d'écrire : `nom()`

Exemple

```
function essai {
echo $1
return 0
}
```

L'appel d'une fonction obéit à la même syntaxe que celle d'une commande. Par, exemple, la fonction donnée en exemple peut être appelée par la ligne de commande : `essai bonjour`

1. Utiliser plutôt des parenthèses afin d'avoir une sémantique habituelle avec les variables ≪ locales ≫ à la fonction

4 Exercices

1. Écrire une commande permettant de supprimer tous les fichiers `core` présents dans l'arbre ayant pour racine le catalogue fourni en paramètre ou, par défaut, le catalogue courant.

2. Écrire une commande qui pose la question « Etes-vous satisfait ? (o/n) » tant que l'usager n'a pas frappé une chaîne de caractères commençant par `"o"` ou `"n"`. (Utiliser `test` ou `expr`).

3. Comment un utilisateur peut-il savoir si les fichiers de son répertoire privé racine ont été modifiés, accédés depuis sa précédente déconnexion ?

4. Écrire une commande

   `arbre [-d] [<nom de répertoire>]`

   qui liste de façon indentée tous les fichiers de l'arborescence du répertoire spécifié. Si l'option `-d` est spécifiée, seuls les répertoires sont listés. Si aucun répertoire n'est spécifié, le répertoire courant est pris comme racine par défaut.

5. Écrire une commande

   `profondeur [<nom de répertoire>]`

   qui calcule la profondeur de l'arbre de répertoires ayant pour racine le répertoire spécifié ou par défaut, le répertoire courant.

   Variante : écrire une commande

   `lepluslong [<nom de répertoire>]`

   qui liste le chemin d'accès le plus long.

6. Une commande comportait les options suivantes :

   `-v -o <nom de fichier> -T <valeur hexa> -S -I -e -t`

   Une nouvelle version du logiciel n'accepte plus les options `-T` et `-e`, et l'option `-t` a été rebaptisée `-x`. Ecrire une commande qui prend en paramètre l'ancienne commande, avec ses paramètres, et exécute la nouvelle commande, avec ses nouveaux paramètres.

7. Écrire une commande

   `inv <chaîne non vide>`

   qui inverse la chaîne de caractères fournie en paramètre et l'affiche. Si le paramètre est oublié, la commande affiche son mode d'emploi.

   Exemple :

   ```
   >inv esope
   >epose
   ```

8. Écrire une commande qui liste, pour tous les répertoires contenus dans le répertoire courant, les informations suivantes : nom du propriétaire, nom du répertoire, date de dernière modification.

9. Écrire une commande

   `dliste [-d <nom rép.>] <mois>+`

   qui liste les fichiers ou répertoires du répertoire spécifié en premier paramètre (ou par défaut le répertoire courant) dont le mois de dernière mise à jour est spécifié dans les paramètres suivants. La commande affiche son mode d'emploi en cas de paramètre erroné.

   Exemple :

   ```
   >dliste Sep Oct Jan
   >dliste -d /bin Feb
   ```

10. Écrire une commande

    `tuer <motif>`

    qui tue le processus dont le nom contient le motif donné en paramètre. Si plusieurs processus correspondent au motif fourni, l'ambiguïté est signalée, et la commande n'a pas d'effet.

11. Écrire une commande

    `compter <fichier> <chaîne>`

    qui compte le nombre d'occurrences du motif donné en second paramètre, dans le fichier fourni en premier paramètre.

    Remarque : le motif donné en second paramètre ne comporte pas d'espaces ; ce motif peut figurer plusieurs fois dans une même ligne du fichier fourni en premier paramètre.

12. Écrire une commande

    `voyelles <chaîne>`

    qui affiche les voyelles contenues dans la chaîne fournie en paramètre. Si une même voyelle figure plusieurs fois dans la chaîne fournie en paramètre, elle ne sera affichée qu'une fois.

13. Écrire une commande qui affiche la plus longue ligne lue depuis l'entrée standard.

14. Écrire une commande qui affiche les exécutables qui peuvent être atteints par plusieurs des chemins figurant dans la variable `PATH`. Variantes : afficher les chemins d'accès complets à ces exécutables ; distinguer les fichiers homonymes des fichiers réellement identiques.

15. Écrire une commande

    `doublons <fichier>`

    qui affiche les lignes du fichier fourni en paramètre qui comportent des mots identiques consécutifs (doublons). Attention, un mot en doublon peut se trouver à cheval sur deux lignes !