Unix : concepts et outils avancés
TD no 2 : scripts shell

Olivier Aumage
LIP, bureau 302
email : Olivier.Aumage@ens-lyon.fr

4 octobre 2001

1 Introduction

L'objectif du TD d'aujourd'hui est de se familiariser progressivement avec l'écriture de *scripts shell*. Dans le cadre du cours et des TD, le shell `sh` sera utilisé comme référence, mais il en existe beaucoup d'autres :
- dérivés directs de `sh` tels que `bash` ;
- shells C et dérivés directs (`csh`, `tcsh`), `tcsh` est le shell par défaut sur les machines de l'ENS ;
- autres (zsh, ksh, etc.)

Les différences portent essentiellement sur la syntaxe des constructions du langage et sur la richesse fonctionnelle. Les fonctionnalités abordées dans le cadre du cours et des TD sont proposées d'une manière ou d'une autre par tous les shells.

Les *scripts* shell sont des programmes courts (et parfois un peu moins courts) interprétés, constitués d'appels de commande et de constructions du langage proposé par le shell correspondant. Ils permettent notamment de contrôler les informations transmises aux programmes et d'analyser de manière rudimentaire les sorties générées par de tels programmes.

Pour faire des traitements plus complexes, tout en gardant la souplesse d'utilisation des scripts, on peut faire appel à l'un des nombreux langages de scripts tels que PERL, Python, TCL ou encore Ruby par exemple (même CAML peut dans une certaine mesure servir de langage de script). Ces langages ne seront pas abordés dans le cours d'Unix, mais il peut être utile de connaître les rudiments de l'un d'eux, à titre personnel.

2 Bases

Écriture et exécution d'un script Un *script shell* est un fichier texte contenant des instructions, des commandes et des symboles spéciaux. On exécute le script en passant le nom du fichier en paramètre au shell. Exemple :

```
% cat exemple.sh
echo hello
% sh exemple.sh
hello
%
```

Le premier symbole spécial est le `#`. Tout mot commençant par un `#` est ignoré et le reste de la ligne également. Le caractère `#` introduit donc un *commentaire*.

Il existe un commentaire spécial de la forme `#!programme` que l'on peut mettre sur la première ligne du script. Dans ce cas, si le fichier contenant le script a le droit d'exécution, il peut être exécuté comme un programme normal.

```
% cat exemple
#!/bin/sh
echo hello
% chmod +x exemple
% ./exemple
```

```
hello
%
```

Variables Comme dans la plupart des langages, il est possible d'utiliser des variables dans un script. Les variables du shell contiennent des chaînes de caractères. Elles n'ont pas besoin d'être déclarées. Elles prennent vie lors de la première affectation (avant la première affectation, elles ont pour valeur la chaîne vide). Une affectation est de la forme suivante :

```
var=chaine
```

Il ne doit pas y avoir d'espace autour du signe =. Note : vous pouvez essayer ces instructions en lançant `sh` (ou `bash` pour avoir un environnement de travail plus convivial) puis en les entrant directement au clavier.

On récupère le contenu de la variable en la préfixant du signe `$`. Essayez les exemples suivants (après avoir exécuté `var=chaine`) :

```
% echo $var
% echo ma$variable
% echo ma${var}iable
```

Expliquez les résultats des deux dernières commandes `echo`.

Testez également les lignes suivantes :

```
% var=ls
% echo '$var'
% echo "$var"
% echo `$var`
```

Dans la terminologie anglaise, utilisée traditionnellement, la première version utilise des *quotes* (ou *single-quotes*), la seconde version utilise des *double-quotes* et la dernière utilise des *backquotes*. Les trois versions sont des chaînes de caractères et peuvent donc être affectées à une variable.

Essayez également les lignes suivantes :

```
% echo '\$var'
% echo "\$var"
```

Quelle(s) déduction(s) faites-vous ?

Premiers scripts Écrivez le script shell `hello` qui produit le résultat :

```
% ./hello.sh
Hello world.
```

Paramètres Tapez le script `my_hello.sh` suivant :

```
% cat my_hello.sh
#!/bin/sh
echo $1
```

Puis essayez les commandes suivante :

```
% ./my_hello.sh
% ./my_hello.sh my hello world
```

Faites les mêmes tests en remplaçant `$1` par `$2` puis par `$3`. Que déduisez vous sur le rôle des variables `$1`, `$2`, etc. ?

Faites les mêmes tests en remplaçant `$1` par `$*` puis par `$#` et déduisez-en le rôle des variables `$*` et `$#`.

Quotes Écrivez un script `brut.sh` qui produit le résultat :

```
% ./brut.sh
Le prix du baril de brut se situe autour de $1 US.
```

Modifiez le script précédent pour que le prix corresponde au premier argument du script :

```
% ./brut2.sh 3
Le prix du baril de brut se situe autour de $3 US.
```

Modifiez le script précédent pour que le prix corresponde au contenu d'un fichier `tarif` :

```
% cat tarif
10
% ./brut3.sh
Le prix du baril de brut se situe autour de $10 US.
```

Modifiez le script précédent pour que le nom du fichier contenant le tarif corresponde à l'argument numéro 1.

3 Constructions

Tests Les tests conditionnels utilisent la construction `if` :

```
if <commande>
then
  <instructions1>
else
  <instructions2>
fi
```

La commande est évaluée et son *code de retour*, aussi appelé *code d'erreur* (il s'agit de la valeur de retour de la fonction `main` ou de la valeur passée en paramètre à `exit` dans un programme **C**), est testée. Si elle est égale à 0, elle est considérée comme *vraie*. Dans ce cas, les `instructions1` sont exécutées. Dans le cas contraire (valeur de retour *non nulle*), les `instructions2` sont exécutées. La construction `if` est très souvent utilisée conjointement avec la commande `test` (`man test` pour les détails).

A quoi correspond `$0` ? Modifiez le script `brut2.sh` pour qu'il affiche un message d'erreur dans le cas où l'argument n'est pas spécifié :

```
% ./brut2.sh
usage: brut2.sh <tarif>
```

Modifiez le script `brut3.sh` pour qu'il vérifie l'existence du fichier passé en paramètre et affiche un message d'erreur si le fichier n'existe pas :

```
% ./brut3.sh bidon
fichier non trouve
```

Expressions et boucles Les boucles sont traitées par les constructions `for` et `while`. La construction `for` a la forme suivante :

```
for <var> in <liste>
do
  <instructions>
done
```

La section `liste` est une liste d'un ou plusieurs mots séparés par des espaces (plus précisément par le ou les caractères séparateurs contenus dans la variable spéciale `IFS`, voir plus loin). La boucle itère sur les mots de la liste en les affectant tour-à-tour à la variable `var`. À chaque itération, les `instructions` sont exécutées. Exemple :

```
% for i in a b c
> do
>   echo $i
> done
a
b
c
%
```

La liste peut être construite en utilisant les *jokers* de noms de fichiers. Le joker `?` remplace (strictement) *une* lettre quelconque, le joker `*` remplace *zéro ou plus* lettres quelconques consécutives. Ainsi, dans l'exemple suivant, la boucle affiche tous les fichiers du répertoire courant dont la seconde lettre est un `e` :

```
for i in ?e*
do
  echo $i
done
```

La construction `while` à la forme suivante :

```
while <commande>
do
  <instructions>
done
```

La `commande` est évaluée et traitée comme pour le cas du `if`, et la section `instructions` est exécutée tant que le code de retour de `commande` reste égal à 0. Comme pour le `if`, la commande du while est souvent un appel à `test`.

Modifiez le script `brut3.sh` pour qu'il affiche les tarifs du brut pour divers pays, chaque tarif étant stocké dans un fichier `<pays>.tarif` (ex : `koweit.tarif`, `venezuela.tarif`, etc.)

```
% ./brut3.sh
Le prix du baril de brut se situe autour de $10 US.
Le prix du baril de brut se situe autour de $15 US.
```

Pour augmenter l'intérêt du script `brut3.sh`, mettez à profit la commande `basename` (`man basename` pour les infos) pour indiquer le nom du pays correspondant sur chaque ligne.

```
% ./brut3.sh
Le prix du baril de brut au koweit se situe autour de $10 US.
Le prix du baril de brut au venezuela se situe autour de $15 US.
```

En utilisant la commande `expr` (`man expr`), écrivez un script `fact` qui affiche la factorielle de l'argument passé en ligne de commande :

```
% ./fact.sh 4
24
```

4 Paramètres

Commande shift Seuls les paramètres `$1` à `$9` sont accessibles directement. Pour utiliser les suivants, il est nécessaire d'utiliser la commande `shift`. Cette commande décale tous les arguments d'une position *vers la gauche* : `$2` est renommé en `$1`, `$3` en `$2`, ... L'argument `$1` est perdu et la variable `$#` est décrémentée. Si la commande `shift` est suivie d'un argument entier positif, elle décale les arguments d'autant.

- Écrivez le script shell `param.sh` qui affiche ses paramètres.
- Écrivez le script shell `param2.sh` qui affiche ses paramètres dans l'ordre inverse.

Commande set La commande `set` remplace la liste d'arguments du programme par une nouvelle liste. Rajoutez `set une liste de parametres` au début du script `param2.sh` (sur la ligne suivant `#!/bin/sh`) et constatez le résultat.

Utilisez la commande `set` pour modifier une dernière fois `brut3.sh`. Les noms des fichiers contenant les tarifs sont maintenant *contenus* dans un fichier dont le nom est passé en argument numéro 1.

Variable IFS La variable `IFS` contient le caractère *séparateur* utilisé par le shell pour *découper* l'entrée de `set` en arguments distincts. Utilisez cette propriété pour afficher le contenu de votre variable `PATH`, en affichant un seul répertoire par ligne.

Quelques exercices supplémentaires

Parcours À l'aide des commandes `set` et `shift` et de la variable spéciale `$*` (chaîne constituée de tous les arguments séparés par le caractère contenu dans `IFS`), écrivez un script `largeur.sh` qui réalise un parcours en largeur d'une arborescence à partir du répertoire spécifié en argument numéro 1 et affiche chacun des répertoires rencontrés durant son parcours.

Quels sont les modifications à faire pour transformer `largeur.sh` en un script `profondeur.sh` qui réalise de manière similaire un parcours en profondeur de l'arborescence.

— ❀ —