Cours de Programmation Unix n°2

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Janvier 2013

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

1 - Développement des noms de fichiers

Motif = argument comportant des jokers

Jokers : `*` `?` `[...]`

Bash substitue chaque motif par la liste des fichiers

- correspondant au motif, et
- existant effectivement,

séparés par des blancs, dans l'ordre lexicographique.

Correspondances :

`*`	toute chaîne, y compris vide
`?`	1 caractère, n'importe lequel
`[`$c_1 c_2 \cdots c_n$`]`	1 caractère dans cet ensemble
`[`c_1`-`c_2`]`	1 caractère dans cet intervalle

Exemples

```
$ ls
bu2.c  bu2.h  ga.c  ga.h  ga.o meu1.c zo5.h

$ echo *.c
bu2.c ga.c meu1.c

$ echo *.[ch]
bu2.c bu2.h ga.c ga.h meu1.c zo5.h

$ echo ??[0-9].[ch]
bu2.c bu2.h zo5.h
```

Fonctionne aussi avec des noms de répertoires :

```
$ ls ../t[dp][1-5]/*.[ch]
```

Fichiers cachés

Les fichiers et répertoires commençant par '.' sont *cachés* y compris les répertoires '.' et '..'

```
$ ls
essai.txt prog1.c resume.txt
$ ls -a
. .. .ancien.txt essai.txt prog1.c resume.txt
```

Les motifs commençant par * ou ? ne correspondent pas aux fichiers cachés.

```
$ ls *.txt
essai.txt resume.txt
$ ls .*.txt
.ancien.txt
```

Caractères exclus

- `[^...]` ou `[!...]` correspond à 1 caractère $\notin$ `[...]`

```
$ ls
essai.txt prog1.c resume.txt
$ ls *[^0-9].*
essai.txt resume.txt
```

- Syntaxe :
  - On peut mélanger éléments et intervalles :
    `[A-Za-z0-9_. ]` `[^A-Za-z0-9_. ]`
  - Inclure ou exclure `']'` : le mettre en premier
    `[]abc]` `[^]abc]`
  - Inclure ou exclure `'-'` : le mettre en premier ou dernier
    `[-abc]` `[abc-]` `[^-abc]` `[^abc-]`
  - Inclure ou exclure `'^'` ou `'!'` : ne pas le mettre en premier
    `[abc^!]` `[^abc^!]`

Aucune correspondance

Si le motif ne correspond à aucun fichier ou répertoire existant, le motif n'est pas développé.

```
$ ls
essai.txt prog1.c resume.txt
$ ls *.h
ls: impossible d'accéder à *.h: Aucun fichier
ou dossier de ce type
$ echo *.h
*.h
```

2 - Développement des accolades

Permet la création de chaînes indépendamment de l'existence des fichiers.

Syntaxe : `{mot1,...,motn}` *sans blanc*

→ bash substitue la chaîne initiale en faisant le produit cartésien des listes

```
$ echo {ga,bu,meu}
ga bu meu
$ echo a{ga,bu,meu}z
agaz abuz ameuz
$ echo {ga,bu,meu}-{eggs,ham}
ga-eggs ga-ham bu-eggs bu-ham meu-eggs meu-ham
```

Imbrication

- On peut imbriquer les {}

```
$ echo ga{zo,pti{foo,bar}lo}bu
                    ↓
$ echo ga{zo,ptifoolo,ptibarlo}bu
                    ↓
gazobu gaptifoolobu gaptibarlobu
```

Protection

- On peut protéger {},␣ avec \ ou '' ou "":

```
$ echo {\{hop\},ploum,a\,b\␣c}
{hop} ploum a,b␣c
$ echo {"{hop}",ploum,'a,b␣c'}
{hop} ploum a,b␣c
```

- Il faut un moins une virgule :

```
$ echo {ga}
{ga}
```
→ pas de développement
```
$ echo {ga,}
ga
```

Mélange motif et accolades

Lorsqu'un motif contient des accolades :

- le développement des {} est fait en premier, les motifs sont dupliqués littéralement ;
- puis le développement des motifs est fait sur les fichiers existants.

```
$ ls /home/{thiel,guest}/tp[1-7]/.c
                        ↓
$ ls /home/thiel/tp[1-7]/.c /home/guest/tp[1-7]/.c
                        ↓
$ ls /home/thiel/tp2/ga.c /home/thiel/tp3/bu.c
     /home/guest/tp1/zo.c
```

3 - Branchements multiples avec case

Syntaxe :

```
case mot in
    motif) instructions ;;
    ...
esac
```

Le `mot` est développé (substitution des variable, etc), puis mis en correspondance avec chaque motif.

Le premier motif qui correspond $\rightarrow$ exécution des instructions puis saut après `esac`.

Les motifs dans case

Règles pour les motifs :

- idem motifs de fichiers : `* ? []`
- sans correspondance à des fichiers effectifs
- pas de développement des `{}`
- + liste motifs avec `|`

```
case "$i" in
    "un")         echo "1" ;;
    deux|trois)   echo "2 ou 3" ;;
    q*)           echo "commence par q" ;;
    *)            echo "rien" ;;
esac
```

`*)` en fin de `case` = cas par défaut

4 - Divers

Commandes `true`, `false` :

Rappel : n'affiche rien, réussit / échoue immédiatement

Essayer : `type true ; help true ; which true`

Usage :

```
x=true
if $x ; then ... ; else ... ; fi

continuer=true
while $continuer ; do ... ; done

if ... ; then true ; else ... ; fi
```

Séparateurs entre deux commandes

`com1 ; com2`	séquentiel : attend la fin de `com1` avant de lancer `com2`
`com1 & com2`	parallèle : lance `com1` en tâche de fond puis lance tout de suite `com2`
`com1 && com2`	séquentiel conditionnel : si `com1` réussi, alors lance `com2`
`com1 || com2`	séquentiel conditionnel : si `com1` échoue, alors lance `com2`

→ Le résultat est celui de la dernière commande exécutée

Attention : `com1 && com2 || com3` n'est pas équivalent à
`if com1 ; then com2 ; else com3 ; fi`

```
$ if true ; then false ; else echo "perdu" ; fi
$ true && false || echo "perdu"
perdu
```

Contrôle de boucles

Dans les boucles `for ... ; do ... ; done`
`while ... ; do ... ; done`

`break [n]` arrêt boucle
`continue [n]` itération suivante

`[n]` : pour les boucles imbriquées,
arrêt ou itération suivante de `n` niveaux

`read` à plusieurs variables

`read` $v_1\ v_2 \cdots v_n$

- lit une ligne sur l'entrée standard,
- découpe en mots (séparateur blanc),
- affecte les variables :

$$\begin{aligned} \text{mot}_1 &\rightarrow v_1 \\ \text{mot}_2 &\rightarrow v_2 \\ &\vdots \\ \text{mot}_{n-1} &\rightarrow v_{n-1} \\ \text{reste ligne} &\rightarrow v_n \end{aligned}$$

$\rightarrow$ `read` v_1 met toute la ligne dans v_1 sans la découper

5 - Fonctions

- Syntaxe : `nom_fonction () { instructions ;}`
  - tapée dans un terminal,
  - ou placé n'importe où dans un script, avant l'appel
  - `()` toujours accolées : signifie déclaration fonction

- Appel : `nom_fonction arguments`

→ Une fonction s'utilise comme un script

```
$ cat titi
#! /bin/bash
echo "Reçu $1"
exit 0

$ ./titi foo
Reçu foo
```

```
$ toto () {
  echo "Reçu $1"
  return 0
}

$ toto foo
Reçu foo
```

Masquage des arguments

Les arguments courants sont masqués pendant l'appel :

```
$ set foo
$ echo "$1"
foo
$ hop () { echo "$1" ;}
$ hop bar
bar
$ echo "$1"
foo
```

Sortie de fonction

Il ne faut pas confondre :

`return [x]` : sortie immédiate de la fonction
`exit [x]` : sortie immédiate du script (ou du shell)

- `x` est le code de terminaison $\rightarrow$ `$?`
- par défaut : code de terminaison de la dernière commande

Comme les scripts, les fonctions réussissent ou échouent mais ne renvoient pas de résultat.

Usage typique :

```
foo () { ..... ; return 0 ;}
if foo ga bu ; then ... ; fi
while foo zo meu ; do ... ; done
```

Portée des variables

Les variables sont globales par défaut :

```
$ hop () { a="foo" ;}
$ a="bar" ; hop ; echo "$a"
foo
```

On peut déclarer des variables locales à une fonction :

```
local a b c=valeur
```

```
$ hop () { local a="foo" ;}
$ a="bar" ; hop ; echo "$a"
bar
```

→ Indispensable pour éviter les effets de bord !

Redirections

- On peut rediriger les E/S d'une fonction :

```
hop () { ls ;}

hop > toto
hop | sort
```

- Bloc entre accolades : permet aussi de rediriger des instructions

```
{ echo "ga" ; echo "bu" ;} > toto
```

6 - Débogage

- Afficher la liste des fonctions :

  ```
  declare -F
  ```

- Afficher le corps d'une fonction :

  ```
  declare -f nom_fonction
  ```

  réussit si la fonction existe.

- Supprimer une fonction :

  ```
  unset -f nom_fonction
  ```

  on peut à tout moment redéfinir une fonction,
  `unset` non obligatoire.

Informations sur les variables

- Afficher la liste des variables :

  ```
  declare -p
  ```

- Afficher plus d'informations sur une variable :

  ```
  declare -p nom_variable
  ```

  réussit si la variable existe.

- Supprimer une variable :

  ```
  unset -v nom_variable
  ```

Conversion automatique

```
declare -l nom_variable
declare -u nom_variable
```

La variable sera convertie en minuscules ou majuscules à l'affectation.

```
$ declare -u ga
$ read ga
bonJOUr
$ echo "$ga"
BONJOUR
```

Désactiver : `declare +l|+u` ou `unset`

Trace automatique

Pour demander à bash d'afficher toutes les commandes telles qu'elles sont exécutées : `set -x` désactiver : `set +x`

```
$ ga="home"; bu="thiel"
$ set -x
$ if [ "$ga" = "$bu" ]; then echo "ok" ; fi
+ '[' home = thiel ']'
$ toto(){ local a="/bin"; cd "$1/$a";}
$ toto "/$ga/$bu"
+ toto /home/thiel
+ local a=/bin
+ cd /home/thiel//bin
$ ls *.sh
+ ls -F numsauv.sh sudoku.sh
numsauv.sh sudoku.sh
$ set +x
```

Trace automatique (2)

- Activer/désactiver dans la console ou dans un script :

`set -x ... set +x`

- Configurer le script pour qu'il s'affiche en mode trace :

```
#! /bin/bash -x
```

- Exécuter le script en mode trace sans le modifier :

```
$ bash -x ./monscript
```

- Voir le chapitre 32 de l'ABSG.