

Séance No 3

- Cours :
 - fonctions du shell
 - substitutions de noms de fichiers
 - structures de contrôles
 - for/in/do/done
 - expressions rationnelles, commande grep
 - caractères de protection
 - structures de contrôles
 - while/do/done
 - case/in/esac

fonctions du shell

- permet d'isoler un traitement particulier constitué de plusieurs commandes
- une fonction est identifiée par son nom
- elle doit être définie avant d'être utilisées. Une fois définie, une fonction est considérée comme une commande interne.
- une fonction s'utilise comme une commande:
« nom liste_de_paramètres »
- la définition d'une fonction n'entraîne l'exécution d'aucune commande. C'est chaque appel à cette fonction qui déclenche l'exécution des fonctions

définition d'une fonction

- syntaxe 1 (conseillée car la plus compatible):

```
mafonction () {  
    commande1  
    commande2  
    ...  
}
```
- syntaxe 2 (shells récents):

```
function maFonction {  
    commande1  
    commande2  
    ...  
}
```

Exemple

- considérons le script suivant:

```
#!/bin/sh  
echo "1ere commande"  
maPremiereFonction(){  
    echo "coucou depuis maPremiereFonction"  
}  
echo "2e commande"  
maPremiereFonction  
echo "après l'appel de maPremiereFonction"
```
- son exécution provoque l'affichage suivant:

```
1ere commande  
2e commande  
coucou depuis maPremiereFonction  
après l'appel de maPremiereFonction
```

Paramètres des fonctions

- comme avec toute commande, il est possible de fournir des paramètres à une fonction.
 - utilisation :
 - les paramètres sont ajoutés après le nom de la fonction, séparés par des espaces ou tout autre caractère séparateur
 - exemple: test premier second
 - définition de la fonction
 - les paramètres sont des variables locales à la fonction nommées: \$1, \$2, ..., \$9, \${10}, ...
 - les variables spéciales \$#, \$* et @\$ sont aussi disponibles
 - \$0 contient par contre le nom du script (et pas le nom de la fonction)

paramètres: exemple

- script: test.sh

```
#!/bin/bash  
f1 () {  
    echo "1er argument de f1: $1"  
    echo "deuxieme argument de f1: $2"  
    echo "nombre d'arguments de f1: $#"  
    echo "tous les arguments de f1: $*"  
    echo "nom du script: $0"  
}  
echo début du script  
echo "1er argument du script: $1"  
echo "deuxieme argument du script: $2"  
echo "nombre d'arguments du script: $#"  
echo "tous les arguments du script: $*"  
echo "nom du script: $0"  
f1 quand est-ce "qu'on" mange
```

paramètres: exemple

- execution du script: « ./test.sh salut les tepos »

```
début du script
1er argument du script: salut
deuxieme argument du script: les
nombre d'arguments du script: 3
tous les arguments du script: salut les tepos
nom du script: test
fi quand est-ce "qu'on" mange#! /bin/bash
1er argument de fi: quand
deuxieme argument de fi: est-ce
nombre d'arguments de fi: 4
tous les arguments de fi: quand est-ce qu'on mange
nom du script: ./test.sh
```

code de retour d'une fonction

- comme toute commande, une fonction a un code de retour
- commande « return n »: termine la fonction et retourne le code d'erreur « n ».
- Erreur à ne pas faire: *return* ne retourne pas le résultat de la fonction mais son code d'erreur

Exemple : recherche dans un annuaire

- on considère le fichier suivant:

```
$ cat annuaire
petit 0169478047
acces2400 0136642424
joe 0404056789
sophie 0164570101
Malik 0237463201
```

- script test.sh

```
#!/bin/bash
estDansAnnuaire (){
    if grep $1 annuaire
    then return 0
    else return 1
```

Exemple revu (1)

- script test.sh

```
#!/bin/bash
estDansAnnuaire (){
    return grep $1 annuaire
}
if estDansAnnuaire petit
then petit est présent dans annuaire
else pas de petit en vue
fi
```

- on utilise directement le code de retour de la commande grep.

Exemple revu (2)

- script test.sh

```
#!/bin/bash
estDansAnnuaire (){
    if [ $# -ne 1 ]
    then
        echo "estDansAnnuaire: utilisation sans
        argument"
        return 1
    fi
    return grep $1 annuaire
}
if estDansAnnuaire petit
then petit est présent dans annuaire
else pas de petit en vue
fi
```

- programmation défensive: vérification du nombre de paramètres.

Variables

- les variables utilisées dans un script sont globales.
- une exception: les variables locales définies par *typeset* (ne sont pas au programme de l'UEL).
- leur portée est dynamique: la variables est utilisable dès que le flot d'exécution l'a rencontrée.

variables locales

- r af: dans une version ult erieure de ce document et de cet enseignement.

fonction: retourner une valeur

- une fonction   3 modes de communication avec le monde ext erieur:
 - via le code de retour : ne pas faire sauf pour une valeur binaire vrai/faux «   la grep »
 - via la sortie standard: m ethode conseill ee car utilisable via redirection. On se contente d'afficher le r esultat sur la sortie standard
 - via une variable : la variable porte en g en eral le nom de la fonction.

Exemples

```
#!/bin/bash
double(){
  echo $1$1
}
triple (){
  triple=$1$1$1
}
d=$(double pa)
triple pa
echo le double de pa est $d
echo le triple de pa est $triple
```

- **ex ecution:**

```
le double de pa est papa
le triple de pa est papapa
```

substitutions de noms de fichiers

- les noms de fichiers param etres de commandes du shell peuvent  tre cit es exhaustivement ou d ecrits   l'aide d'expression g en erique (« caract eres joker »)
- caract eres:
 - *: une suite de caract eres
 - ?: un unique caract ere
 - [abcd]: l'un des caract ere entre crochets
 - [a-d]: un caract ere situ e entre a et d
 - [^liste]: cit e en premi ere position, ^ est un caract ere de n egation: tous les caract eres sauf ceux de *liste*.

Exemples

- *.sh: les fichiers dont le nom finit par .sh
- p*: les fichiers dont le nom commence par la lettre p
- [A-Z]*.sh : les fichiers dont le nom commence par une majuscule et finissant par .sh
- ??? : les fichiers dont le nom fait 3 lettres
- [^a-z]* : les fichiers dont le nom ne commence pas par une minuscule non accentu ee

structures de contr ole: for

- syntaxe:

```
for var in listeValeur
do
  commande1
  ...
done
```
- s emantique:
 - la variable *var* prend chaque valeur de la liste de valeur
 - les commandes situ ees entre do et done sont ex ecut ees pour chaque valeur de la variable *var*

Exemple 1: liste de valeurs citée

```
$for i in 1 2 3 4
do
  echo $i
done
1
2
3
4
```

Exemple 2: liste des valeurs contenues dans une variable:

```
$ varTest="a b c d"
$for i in $varTest
do
  echo $i
done
a
b
c
d
```

Exemple 3: liste des valeurs via substitution de commande

```
$for i in $(ls)
do
  echo $i
done
fichier1
fichier2
... (résultat de la commande ls, un fichier par ligne)
```

Exemple 4: liste des valeurs via génération de noms de fichiers

```
$for i in f*
do
  echo $i
done
fichier1
fichier2
... (les noms des fichiers commençant par f dans le dossier courant, un fichier par ligne)
```

structures de contrôle: case/in/esac

- syntaxe:

```
case mot in
  modele1) liste de commandes;;
  modele2|modele3|modele4) liste de commandes;;
  ...
esac
```
- les modèles sont des chaînes incluant éventuellement des caractères spéciaux * ? []^ (utilisés dans la substitution des noms de fichiers)

structures de contrôle: case/in/esac

- principe
 - la valeur du mot est comparée à chaque modèle
 - la liste de commandes du premier modèle correspondant est exécutée jusqu'au « ;; »
 - l'exécution se poursuit ensuite après le « esac ».

case/in/esac: exemples

```
$ cat codePostal.sh
#!/bin/bash

case "$1" in
  75[0-9][0-9][0-9])
    echo "code postal parisien"
    ;;
  7[78][0-9][0-9][0-9]|9[1-5][0-9][0-9][0-9])
    echo "code postal ile de France"
    ;;
  [0-9][0-9][0-9][0-9][0-9])
    echo "code postal de la france metropolitaine"
    ;;
  *)
    echo "code postal non reconnu"
    ;;
esac
```

structures de contrôles: while/do/done

- syntaxe
- ```
while commande1
do
 liste de commandes
done
```
- « commande1 » est exécutée,
    - si le code de retour vaut 0, la liste de commande est exécutée et on retourne exécuter « commande1 »
    - si le code de retour de « commande 1 » est différent de 0, la commande qui suit le « done » est exécutée.

## while/do/done: exemples

```
$ cat facto.sh
#!/bin/bash
n=5
x=1
while ((n > 0))
do
 ((x*=n))
 ((n-=1))
done
echo "12!=$x"

$./facto.sh
12!=120
```

## while/do/done: exemples

```
$ cat test.sh
#!/bin/bash
n=1
while (($# > 0))
do
 echo "parametre No $n: $1"
 ((n=n+1))
 shift
done

$./test.sh a g df "ze rt"
parametre No 1: a
parametre No 2: g
parametre No 3: df
parametre No 4: ze rt
```

## Expressions rationnelles

- utilisées par des nombreux outils pour décrire des chaînes de caractères
- Il existe deux familles d'expressions rationnelles :
  - expression régulières de base utilisées par
    - vi
    - grep
    - expr
    - sed
  - expressions régulière étendues utilisées par :
    - grep -E et egrep
    - awk

## •expressions rationnelles: éléments communs

- les caractères suivants sont communs aux expressions rationnelles de base et étendues:
  - ^: début de ligne
  - \$: fin de ligne
  - .: un caractère quelconque
  - [liste]: un caractère de la liste (idem [] du shell, ^: négation)
  - \*: 0 à n fois le caractère/expression précédente
  - |<, |>: chaîne vide de début et de fin de mot
  - \c: protège le caractère c de tout interprétation. ex.: \\$: caractère \$

## •Exemples:

- petit: la chaîne petit
- ^petit: chaîne démarrant par petit
- petit\$: chaîne finissant par petit
- DSCN[0-9]\*\JPG: DSCN suivi de 0 à N chiffres suivi de .JPG (noter la protection du caractère .)
- ^\$: chaîne vide (utile pour détecter les lignes vides)
- ^[^:]\*:[^:]\*:[^:]\*:1000 une chaîne contenant des champs séparés par le caractère : et dont le 4e champ vaut 1000. Appliqué sur un fichier /etc/passwd, sélectionne les lignes dont le groupe est 1000.

## caractères de protection

- \: protège le caractère suivant (sauf si c'est un saut de ligne)
- \ en fin de ligne:
- 'chaîne': protège tous les caractères de la chaîne
- "chaîne": protège tous les caractères sauf \$, ` et \ dans le cas où \ est devant \$, ", \ et en fin de ligne.
- caractères à protéger :  
| & ; < > ( ) \$ ` \ " ' <espace> <tab>  
<saut de ligne> \* ? [ # ~ = %

## •caractères de protection: exemples

```
$ echo 'un test $#\c\$('
un test $#\c\($
$ echo "un test $#\c\($"
un test 0\c$
$ echo un test \$
un test $
$ echo un test \$ \\
un test $ \
```