



UNIVERSITÉ DE  
MONTPELLIER



# Introduction aux systèmes d'exploitation et à leur fonctionnement

Programmation shell, scripts

Fabien Laguillaumie ([fabien.laguillaumie@umontpellier.fr](mailto:fabien.laguillaumie@umontpellier.fr))

# OBJECTIFS DE CETTE PARTIE

Introduire la "programmation" du shell avec les scripts

- ▶ Comprendre l'interprétation des scripts
- ▶ Connaître les commandes de contrôle
- ▶ Connaître le passage des arguments

# Prélude : grep et sed

# UN ÉDITEUR-FILTRE UTILE : GREP

## Syntaxe

```
grep [OPTION...] PATTERNS [FILE...]
```

## Description

- ▶ Recherche les motifs PATTERNS dans les fichiers FILE et affiche les lignes qui les contiennent
- ▶ Le motif peut être donné sous forme d'une **expression régulière**
- ▶ S'il n'y a pas de fichier précisé, l'entrée standard est analysée

# GREP : EXPRESSIONS RÉGULIÈRES

## Méta-caractères :

- ▶ `.` : un caractère quelconque
- ▶ `*` : 0 ou plusieurs fois le caractère précédant
- ▶ `?` : 0 ou 1 fois
- ▶ `+` : 1 ou plusieurs fois
- ▶ `( )` : délimiter une chaîne
- ▶ `{ }` : définition d'une multiplicité
- ▶ `[ ]` : définition d'un seul caractère (une position)
- ▶ `[^ ... ]` : tout, sauf les caractères donnés
- ▶ `^ ... $` : début ... fin de la ligne

`b.b`

`b*a` ou `.*c`

`b[ai]b` ou `[a-z][0-9]*`

Attention : `b[0-28]b` correspond à `b0b`, `b1b`, `b2b` et `b8b`

▶ Pour dés-spécialiser un méta-caractère, on utilise le `\`



## GREP : QUELQUES OPTIONS

- ▶ -v : affiche les lignes ne contenant pas la chaîne
- ▶ -c : compte le nombre de lignes contenant la chaîne
- ▶ -n : chaque ligne contenant la chaîne est numérotée
- ▶ -x : ligne correspondant exactement à la chaîne
- ▶ -l : affiche le nom des fichiers qui contiennent la chaîne
- ▶ -r : recherche récursive (dans un répertoire)
- ▶ -i : recherche sans tenir compte de la casse

## Syntaxe

```
sed [OPTION]... script-only-if-no-other-script [input-file]...
```

## Description

- ▶ Éditeur de flux qui filtre et modifie le texte
- ▶ Il lit les lignes d'un fichier une à une, applique des commandes d'édition et renvoie les lignes résultantes sur la sortie standard.
- ▶ Il supporte les expressions régulières

très puissant

RTFM

## SED : SUBSTITUTION

s permet de changer la 1ère ou toutes les occurrences d'une chaîne par une autre.

- ▶ **login@machine:~\$ sed "s/toto/baba/" fichier**  
change la 1ère occurrence de la chaîne toto par baba
- ▶ **login@machine:~\$ sed "s/toto/baba/3" fichier**  
change la 3ème occurrence de la chaîne toto par baba
- ▶ **login@machine:~\$ sed "s/toto/baba/g" fichier**  
change toutes les occurrences de la chaîne toto par baba
- ▶ **login@machine:~\$ sed "s/toto/baba/p" fichier**  
en cas de remplacement, affiche les lignes correspondantes
- ▶ **login@machine:~\$ sed "s/toto/baba/w resultat" fichier**  
en cas de remplacement, la ligne est écrite dans resultat
- ▶ **login@machine:~\$ sed "s/[Tt]oto/baba/g"**

on peut bien sûr utiliser des expressions régulières



Fin du prélude.

# EXEMPLE INTRODUCTIF

- ▶ Une séquence de commandes est exécutée dans le shell
  - ▶ **login@machine:~\$ rep=Archil**
  - ▶ **login@machine:~\$ pwd**
  - ▶ **login@machine:~\$ cd \$rep**
  - ▶ **login@machine:~\$ ls**
  - ▶ **login@machine:~\$ ps**
- ▶ La même séquence peut être enregistrée dans un fichier texte après avoir indiqué **#!/bin/bash**
- ▶ Le droit d'exécution doit être accordé à ce fichier
- ▶ Le fichier (script) peut alors être exécuté

Dans les scripts :

- ▶ séquences de commandes
- ▶ des variables
  - ▶ comment utiliser des variables numériques ?
  - ▶ comment passer des arguments, des valeurs à un script ?
- ▶ structures de contrôle
  - ▶ comment organiser des boucles ?
  - ▶ comment réaliser des branchements conditionnels ?
  - ▶ comment faire des fonctions ?

## Les scripts bash :

- ▶ Ce sont des fichiers textes qui regroupent un ensemble d'instructions qui pourraient être tapées en ligne de commandes.
- ▶ Ils sont rédigés dans le langage bash et interprétés par bash.
- ▶ Ils peuvent admettre des paramètres.
- ▶ Ils peuvent être lancés depuis la ligne de commande.

## Le nom des scripts

- ▶ On donne un nom *expressif*.
- ▶ On peut donner l'extension (facultative) `.sh`.

## La structure :

# ! = « sha-bang »

- ▶ La première ligne est composée de `#!/bin/bash`, suivie d'une ligne vide
- ▶ Chaque ligne correspond à une instruction
- ▶ Sur un ligne, tout texte placé après `#` est considéré comme un commentaire et ne sera pas interprété

```
#!/bin/bash

rep=Archi1                # définition de la variable rep
pwd                      # affichage de répertoire courant
cd $rep                  # déplacement dans le répertoire défini par rep
ls
ps
```

- Rappel : pour rendre le script exécutable

```
login@machine:~$ chmod u+x monshell
```

- Pour l'exécuter, on tape son chemin

```
login@machine:~$ ./monshell
```

```
login@machine:~$ mes_scripts/monshell
```

## SCRIPTS : ARGUMENTS

- ▶ Le shell peut passer des arguments au script
- ▶ Ce sont des chaînes de caractères tapées dans la ligne de commande
- ▶ Ces informations sont passées au script comme variables qui peuvent être utilisés dans les instructions

Nom de la variable	Valeur de la variable
\$0	Le nom du script
\$1 à \$9	Les arguments passés à l'appel du script
\$#	Le nombre de paramètres
\$*	La liste des arguments à partir de \$1

```
login@machine:~$ ./monscript -h 123 fichier.pgm
```

./ indique le “chemin complet” au bash, qui ne vérifie pas \$PATH

# SCRIPTS : ARGUMENTS

```
#!/bin/bash

echo "nom du script :" $0
echo "nombre de paramètres :" $#
echo "premier paramètre :" $1
echo "deuxième paramètre :" $2
```

Son exécution :

A terminal window titled 'laguillaumie@fabuntu: /tmp' with standard window controls. The prompt is 'laguillaumie@fabuntu:/tmp\$'. The command './testpar -h toto.tex' has been entered and executed. The output shows: 'nom du script : ./testpar', 'nombre de paramètres : 2', 'premier paramètre : -h', and 'deuxième paramètre : toto.tex'. The prompt is now 'laguillaumie@fabuntu:/tmp\$' with a cursor.

```
laguillaumie@fabuntu:/tmp$ ./testpar -h toto.tex
nom du script : ./testpar
nombre de paramètres : 2
premier paramètre : -h
deuxième paramètre : toto.tex
laguillaumie@fabuntu:/tmp$
```



# ÉCHAPPEMENTS

Pour un échappement

- ▶ **ponctuel** : \  
Le caractère qui suit le \ échappe à l'interprétation
- ▶ **complet** : guillemets simples ' '  
Tous les caractères entre guillemets simples échappent à l'interprétation
- ▶ **partiel** : guillemets doubles " "  
Tous les caractères compris entre guillemets doubles échappent à l'interprétation, sauf le caractère \$.  
⇒ noms de variables interprétés, et donc remplacés par leur valeur

# LES ENTIERS

- ▶ stockés dans des variables
- ▶ pas de type
- ▶ `a=2; b=3; c=5`
- ▶ Les opérations sur les entiers se font en utilisant les symboles :

`$(( et ))`

- ▶ `d = $(( $a**$b+$c ))`  
`echo $(( $d - 1 ))`

- ▶ On peut utiliser la commande `expr`  $\rightsquigarrow$  `man expr`

# TESTS

- ▶ Il est possible d'effectuer des *tests* sur les entiers ou les chaînes de caractères à l'aide de la commande **test**.
- ▶ De façon équivalente, on peut utiliser **[ et ]**.  
**Dans ce dernier cas, il ne faut pas oublier les espaces.**
- ▶ **[ \$A = 1 ], [ \$i -le 4 ], test \$i -le 4,...**

Remarques :

```
login@machine:~$ type [  
login@machine:~$ which [  
login@machine:~$ ls -l /usr/bin/
```

```
login@machine:~$ [ 4 == 4 ]  
login@machine:~$ echo $?  
login@machine:~$ [ 4 == 5 ]  
login@machine:~$ echo $?
```

# TESTS

## Tests pour les entiers

- ▶  $x == y$  : les deux valeurs sont égales
- ▶  $x \neq y$  : les deux valeurs ne sont pas égales
- ▶  $x > y$  :  $x$  est supérieur à  $y$
- ▶  $x \geq y$  :  $x$  est supérieur ou égal à  $y$
- ▶  $x < y$  :  $x$  est inférieur à  $y$
- ▶  $x \leq y$  :  $x$  est inférieur ou égal à  $y$

**Remarque :** Ces expressions peuvent être enchaînées par des opérateurs logiques :

- ▶  $\&$  : opérateur "et" (and)
- ▶  $\|$  : opérateur "ou" (or)

! : complément

# TESTS

## Tests pour les chaînes de caractères

- ▶ `-n mot` : vrai si la longueur de `mot` n'est pas égale à 0
- ▶ `-z mot` : vrai si la longueur de `mot` est égale à 0
- ▶ `mot1 = mot2` : les mots sont identiques
- ▶ `mot1 != mot2` : les mots sont différents

```
laguillaumie@fabuntu: ~  
laguillaumie@fabuntu:~$ if [ -n toto ]  
> then  
> echo oui  
> fi  
oui  
laguillaumie@fabuntu:~$ exemple=bonjour  
laguillaumie@fabuntu:~$ if [ $exemple = bonjour ]  
> then  
> echo en effet  
> fi  
en effet  
laguillaumie@fabuntu:~$
```

## Tests pour les fichiers et répertoires

- ▶ `-f fichier` : le fichier existe et il est ordinaire
- ▶ `-d fichier` : le fichier existe et il est un répertoire
- ▶ `-x fichier` : le fichier existe et il est exécutable
- ▶ `-r fichier` : le fichier existe et il est accessible en lecture
- ▶ `-r fichier` : le fichier existe et il est accessible en écriture
- ▶ `-e fichier` : le fichier existe

# SCRIPTS : BOUCLE FOR

- ▶ **parcourir une énumération :**

```
for var in mot1 mot2  
do  
    commandes  
done
```

- ▶ **« comme en C » :**

```
for (( initialisation de var ; condition ; modification ))  
do  
    commandes  
done
```

# SCRIPTS : BOUCLE FOR

## ► parcourir une énumération :

```
for var in blabla en cours
do
    echo $var
done
```

```
for var in `ls`
do
    ls -l | grep pdf
done
```

```
for var in *
do
    echo $var
done
```



# SCRIPTS : BOUCLE FOR

► « comme en C » :

```
for (( var=0 ; var < 3 ; var++ ))  
do  
    echo $var  
done
```

# SCRIPTS : BOUCLE WHILE

- ▶ Une séquence de commandes peut être exécutée tant qu'une condition est vraie

```
while commande1  
do  
    commande2  
done
```

- ▶ A chaque itération, l'exécution de commande2 est conditionnée par le code de retour de commande1 :
  - ▶ S'il est égal à 0, alors le corps est exécuté
  - ▶ S'il est différent de 0, on quitte la boucle

## INTERLUDE : CODE DE RETOUR D'UNE COMMANDE

- ▶ Quand on exécute une commande ou un script, on reçoit un **code de retour**
  - ▶ **login@machine:~\$ echo \$?**
- ▶ C'est un nombre compris entre 0 et 255
- ▶ Exit status 0 : la dernière commande (ou script) s'est exécutée avec succès
- ▶ Exit status 2 : problème de permission ou erreur de syntaxe



```
laguillaumie@fabuntu: ~  
laguillaumie@fabuntu:~$ for i in *  
> echo $i  
bash: syntax error near unexpected token `echo'  
laguillaumie@fabuntu:~$ echo $?  
2  
laguillaumie@fabuntu:~$
```

- ▶ Exit status 126 : problème de permission d'exécution pour un script
- ▶ Exit status 127 : la commande n'existe pas, ou n'est pas dans le PATH

## INTERLUDE : CODE DE RETOUR D'UNE COMMANDE

- ▶ Un code de retour égal à 1 renvoyé par `ls` n'a pas la même signification qu'un code de retour égal à 1 renvoyé par `grep`.
- ▶ Les valeurs et significations du code de retour d'une commande unix ou du shell sont documentées dans les pages correspondantes du manuel (ex : `man grep`).
- ▶ Dans un script, on peut renvoyer un code de retour avec la commande `exit`.
- ▶ Lorsque une commande est exécutée en *arrière-plan* (exécution asynchrone), son code de retour *n'est pas* mémorisé dans le paramètre spécial « ? ».

# SCRIPTS : BOUCLE WHILE

```
while cat sujet.txt  
do  
    echo bonjour  
done
```

```
n=1  
while (( $n <= 5 ))  
do  
    echo "Valeur : $n"  
    n=$((n+1))  
done
```

Remarque : `let "expression"` correspond à `((expression))`

```
login@machine:~$ let "n=n+1"
```

# SCRIPTS : BOUCLE UNTIL

- Organisation différente par rapport à while

```
until commande1
do
    commande2
done
```

- A chaque itération, l'exécution de commande2 est conditionnée par le code de retour de commande1 :
  - S'il est différent de 0, alors le corps est exécuté
  - S'il est égal à 0, on quitte la boucle

```
until cat sujet.txt
do
    echo bonjour
done
```

# SCRIPTS : INSTRUCTIONS CONDITIONNELLES

## Commande if

```
if command01
then
    command11
fi
```

```
if command01
then
    command11
else
    command12
fi
```

```
if command01
then
    command11
elif
    command02
then
    command12
else
    command13
fi
```

# SCRIPTS : INSTRUCTIONS CONDITIONNELLES

## Branchement case

- ▶ Quand le mot correspond à un premier motif dans la liste, les actions associées sont exécutées
- ▶ La liste des actions est terminée par `;;`
- ▶ Plusieurs motifs peuvent être séparés par des `|` (ou)
- ▶ Le cas `*`) se traduit : autrement

```
case mot in
motif1) command
:
command ;;
motif2 | motif3 ) commande
command ;;
*) commande
: ;;
esac
```



Exemple.