# Introduction aux systèmes d'exploitation - M2 CCI

## Scripts Shell

Nicolas Louvet (Univ. Lyon 1)

d'après un cours pompé sur Laure Gonnord
qui avait pompé elle-même sur Julien Forget

1. Fichiers de commandes (scripts)
2. Passage de paramètres et code de retour
3. Chaînes de caractères et caractères spéciaux
4. Arithmétique et expressions booléennes
5. Exécution conditionnelle
6. Itérations
7. Fonctions
8. Divers

# Fichiers de commandes

Fichier de commandes :

- Fichier contenant une suite de commandes vues précédemment ;
- Unix (script shell) :
  - Par convention, extension `.sh` ;
  - Par convention, indiquer sur la première ligne le shell utilisé : `#!/bin/sh` ou `#!/bin/bash`
  - Pas besoin de compiler ! Il suffit de mettre le droit d'exécution (`chmod u+x`) au fichier puis d'exécuter directement le fichier.

# Fichier de commandes

Sous UNIX :

```
#!/bin/bash
pwd
cd ../TP
pwd
```

```
laure@meert:~/SystemeCCI/Cours$ ls *.sh
cdbin.sh  parameters.sh
laure@meert:~/SystemeCCI/Cours$ ./cdbin.sh
bash: ./cdbin.sh: Permission denied
laure@meert:~/SystemeCCI/Cours$ cdbin.sh
cdbin.sh: command not found
laure@meert:~/SystemeCCI/Cours$ chmod u+x cdbin.sh
laure@meert:~/SystemeCCI/Cours$ ./cdbin.sh
/home/laure/SystemeCCI/Cours
/home/laure/SystemeCCI/TP
```

Le script s'exécute dans son propre environnement.

# Langage de programmation ?

Le “script shell” n’est pas un langage de programmation.

Différences notoires :

- Pas de compilation ;
- Les expressions et variables sont toutes des *chaînes de caractères* ;
- Pas d’expressions arithmétiques $\Rightarrow$ commande `expr` ;
- Pas d’expressions booléennes $\Rightarrow$ commande `test`.
- Le shell fournit :
  - un mécanisme de passage de paramètres
  - les structures de contrôle (`if-then-else`, `for`, etc)
  - un mécanisme de gestion de fonctions

Ces commandes et le shell ont un `man` associé.

# Passage de paramètres

Exemple : `commande paramètre1 paramètre2 paramètre3`

$* donne la liste des paramètres
$# donne le nombre de paramètres
$0 donne le nom de la commande
$1 donne le premier paramètre, $2, $3 ...

# Passage de paramètres : exemple

```
#!/bin/sh
#Le fichier parameters.sh
echo "Le programme a $# arguments"
echo "Les deux premiers sont $1 puis $2"
echo "Tous les arguments : $*"
```

```
laure@meert:~$ ./parameters.sh premier deuxième troisième
Le programme a 3 arguments
Les deux premiers sont premier puis deuxième
Tous les arguments : premier deuxième troisième
```

# Code de retour

Toute commande UNIX renvoie un code de retour :

- 0 si tout est correct
- entier $> 0$ sinon
- ce code ne s'affiche pas, il est stocké dans la **variable** « ? »

```
laure@meert:~$ ls -la cookies.txt
-rw-rw-r-- 1 laure laure 164 août  14 08:57 cookies.txt
laure@meert:~$ echo $?
0
laure@meert:~$ ls -la test
ls: impossible d'accéder à test: Aucun fichier ou dossier de ce type
laure@meert:~$ echo $?
2
```

Un script renvoie aussi un code de retour :

- par défaut, c'est le code de retour de la dernière commande qu'il exécute,
- on peut utiliser `exit n` pour renvoyer le code `n`.

# Caractères spéciaux

- # introduit un commentaire
- \ préserve la valeur du caractère suivant (échappement)
- ' ' délimitent une chaîne de caractères. Les variables qu'elle contient ne sont pas évaluées
- " " délimitent une chaîne de caractères. Les variables qu'elle contient sont évaluées

# Caractères spéciaux : un exemple à retenir !

```
laure@Latitude-XT2:~$ a=essai
laure@Latitude-XT2:~$ echo "ceci est un $a"
ceci est un essai
laure@Latitude-XT2:~$ echo 'ceci est un $a'
ceci est un $a
laure@Latitude-XT2:~$ b="ceci est un $a"
laure@Latitude-XT2:~$ echo $b
ceci est un essai
laure@Latitude-XT2:~$ c='ceci est un $a'
laure@Latitude-XT2:~$ echo $c
ceci est un $a
```

- Dans une chaîne entre guillemets "", ce qui apparaît sous la forme de `$nom` est évalué.
- Dans une chaîne entre quote '', ce qui apparaît sous la forme de `$nom` n'est pas évalué.

# Récupérer le résultat d'une commande

le $() permet de récupérer le résultat d'une commande dans une chaîne.

```
laure@meert:~$ echo "la date est date"
la date est date
laure@meert:~$ echo "la date est $(date)"
la date est mardi 20 août 2013, 15:30:59 (UTC+0200)
```

On peut donc aussi mettre le résultat d'une commande dans une chaîne :

```
laure@meert:~$ a=$(date)
laure@meert:~$ echo "la date est $a"
la date est mardi 20 août 2013, 15:32:33 (UTC+0200)
```

# Arithmétique

`expr` :

- les arguments sont interprétés comme des expressions arithmétiques ;
- les différents termes de l'expression doivent être séparés par des espaces

```
laure@meert:~$ expr 5+6
5+6
laure@meert:~$ expr 5 + 6
11
laure@meert:~$ a=10
laure@meert:~$ a=$(expr $a + 1)
laure@meert:~$ echo $a
11
```

# Expressions booléennes : `test`

`test <expression>` ou `[␣<expression>␣]`

- L'expression est évaluée et si la valeur est vraie alors le code de retour de la commande est 0, sinon 1 ;
- Sur les chaînes de caractères :
  - `test -z <chaîne>` est vraie si la chaîne est vide
  - `test -n <chaîne>` est vraie si la chaîne n'est pas vide
  - `test <chaîne1> = <chaîne2>` (`!=` différent)

```
laure@meert:~/tmp$ test toto = titi
laure@meert:~/tmp$ echo $?
1
laure@meert:~/tmp$ test toto = toto
laure@meert:~/tmp$ echo $?
0
```

# test

- sur les chaînes numériques (une suite de chiffres)

  `test <chaîne1> -eq <chaîne2>`

  - `-eq` pour equal, `ne` pour not equal
  - `-le`, `-gt`, ...
- sur les fichiers
  - `test -f <fichier>` est vrai ssi `fichier` est un fichier ordinaire
  - `test -d <repert>` est vrai ssi `repert` est un répertoire
- `man test` pour tous les détails !

**Attention :** le placement des espaces est une grosse source de problèmes avec les tests (et dans les scripts shell en général...)

# test

Opérateurs sur les expressions logiques :

- négation : !
- conjonction : -a
- disjonction : -o

```
laure@meert:~/tmp$ ls e*
essai  essai~  exos_commandes.txt  exos_gis.txt  exos_gis.txt~
laure@meert:~/tmp$ test -f essai -o 6 -gt 5; echo $?
0
laure@meert:~/tmp$ [ -f essai -o 6 -gt 5 ]; echo $?
0
laure@meert:~/tmp$ u=2; v=5; test \( "$u" -eq 2 \) -a \( "$v" -eq 4 \)
laure@meert:~/tmp$ echo $?
1
```

# Conditionnelle

```
if <liste de commandes>
   then <liste de commandes>
   else <liste de commandes>
fi
```

cascade de if

```
if <liste de commandes>
   then <liste de commandes>
   else if <liste de commandes>
        then .....
        else if ....
             fi
        fi
fi
```

```
#! /bin/bash
if test -f $1
then
    echo $1 existe
else
    echo $1 n'existe pas
fi
```

# Aiguillage

```
case <chaine> in
    <motif>) <liste de commandes>;;
    <motif>) <liste de commandes>;;
     ........
     ........
esac
```

```
#! /bin/sh

case $# in
     0) pwd;;
     1) if [ -f $1 ]
          then echo $1
               existe
        fi;;
     2) cp $1 $2 ;;
     *) echo trop de
          paramètres;;
esac
```

# Itération bornée

```
for <variable> in <chaine>
    do <liste de commandes>
done
```

```
laure@meert:~$ for i in a b c; do echo $i; done
a
b
c
laure@meert:~$ for i in $(ls CM3*); do echo "** $i **"; done
** CM3_Prog.aux **
** CM3_Prog.log **
** CM3_Prog.nav **
** CM3_Prog.out **
** CM3_Prog.pdf **
** CM3_Prog.snm **
** CM3_Prog.tex **
** CM3_Prog.toc **
** CM3_Prog.vrb **
```

# Itération avec indices de boucle

Le mieux est de se débrouiller avec la commande `seq` :

`for i in $(seq first incr last); do commandes; done`

```
laure@meert:~$ for i in $(seq 3 2 8); do echo $i; done
3
5
7
```

# Itération non bornée

En sh

**while** <liste de commandes>
    **do** <liste de commandes>
**done**

**until** <liste de commandes>
    **do** <liste de commandes>
**done**

```
#! /bin/bash

echo -n 'repondre par oui ou par non '
read reponse
while test "$reponse" != "oui" -a "$reponse" != "non"
do
    echo -n 'il faut repondre par oui ou par non '
    read reponse
done
echo 'fini!'
```

# Fonctions

**Déclaration** :

```
ma_fonction()
{
   <liste de commandes>
}
```

**Appel** :

`ma_fonction arg1 arg2 arg3 ...`

- On ne déclare pas les paramètres de la fonction ;
- Pas de vérification par le shell sur le nombre ou le type des paramètres à l'appel ⇒ à vérifier vous-même !
- A l'intérieur de la fonction, les $1, $2, ... correspondent aux paramètres d'appel de la fonction ;
- Le { et le } doivent être en début de ligne.

C'est un peu comme si on écrivait un nouveau script dans le même fichier...

# Fonction

```
#!/bin/sh
#Le fichier lt.sh
lt()
{
    ls -lt $1 | more
}

for fic in $*; do
     lt $fic
done

laure@meert:~/tmp$ ./lt.sh  fonction.sh sys.h
-rwxrwxr-x 1 laure laure 51 Aug  19 15:54 fonction.sh
-rw-r--r-- 1 laure laure 921 Aug  19 15:56 sys.h
```

# Les paramètres

- décaler les paramètres de position
  - `shift` [*n*]
  - ex : `shift` sans paramètre, $2 devient $1, $3 devient $2, etc.

```
#!/bin/sh

if test $# -eq 0 ; then
  echo "Je n'ai pas de paramètre :("
else
  echo "J'ai des paramètres :D"
  echo "Les voici :"
  while test ! $# -eq 0 ; do
    echo "** $1 **"
    shift 1
  done
fi;
```

- @ est une variable qui donne la liste des paramètres :

```
#!/bin/sh

echo "Mes paramètres :"
for i in "$@"; do
  echo "** $i **"
done
```

# Fichiers d'initialisation

Exemple de configuration utilisateur `~/.bashrc`

```
#definition de quelques alias
alias la='ls -a'

#ajout de chemins perso pour la variable PATH
PATH=$PATH:/home/laure/bin:$HOME/piplib-1.4.0:$HOME/C2Fsm/bin
export PATH
```

On peut faire beaucoup plus que cela :

- redéfinir le "look" du prompt
- définir des fonctions

Prise en compte immédiate du fichier `.bashrc` :

```
laure@meert:~/$source .bashrc
```