

# Les interpréteurs de commandes, le shell Bash

Luigi Santocanale

Laboratoire d'Informatique Fondamentale,  
Centre de Mathématiques et Informatique,  
39, rue Joliot-Curie - F-13453 Marseille

10 novembre 2004

# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - Tests
  - Boucles

# Plan

## 1 Introduction aux interpréteurs

- Généralités
- Les mécanismes de substitution et d'expansion
- Lancement des processus, redirection, communication

## 2 Programmation avec les scripts

- Introduction
- Variables spéciales
- Tests
- Boucles

# Objectifs d'un interpréteur (shell)

Première interface entre l'utilisateur et le système :

- soumettre de commandes, de tâches,
- faciliter le travail de l'utilisateur : mécanismes de
  - complétion,
  - historique,
  - expansion,
  - contrôle des tâches (« job control »), etc.

# Objectifs d'un interpréteur (shell)

Première interface entre l'utilisateur et le système :

- soumettre de commandes, de tâches,
- faciliter le travail de l'utilisateur : mécanismes de
  - complétion,
  - historique,
  - expansion,
  - contrôle des tâches (« job control »), etc.

# Les commandes externes

L'interpréteur génère un nouveau processus  
qui se charge d'exécuter la commande.

Exemple :

```
[lsantoca@localhost lecture6]$ sleep 1 & ps
  PID TTY          TIME CMD
 3331 pts/2    00:00:00 bash
 4648 pts/2    00:00:00 sleep
 4649 pts/2    00:00:00 ps
```

# Les commandes externes

L'interpréteur génère un nouveau processus qui se charge d'exécuter la commande.

Exemple :

```
[lsantoca@localhost lecture6]$ sleep 1 & ps
```

PID	TTY	TIME	CMD
3331	pts/2	00:00:00	bash
4648	pts/2	00:00:00	sleep
4649	pts/2	00:00:00	ps

# Les commandes internes

Résidents à l'intérieur du shell,  
ils n'engendrent pas de nouveaux processus.

Leurs but :

- améliorer la performance d'un commande externe,
- organiser les commandes en unités logiques.

Exemple :

```
[lsantoca@localhost lecture6]$ which jobs
which: no jobs in ( ... )
[lsantoca@localhost lecture6]$ jobs
[3]+  Running                  xpdf lecture6.pdf &
```

Un interpréteur est un langage de commandes.

On peut s'en servir pour écrire des commandes complexes.



# Les commandes internes

Résidents à l'intérieur du shell,  
ils n'engendrent pas de nouveaux processus.

Leurs but :

- améliorer la performance d'un commande externe,
- organiser les commandes en unités logiques.

Exemple :

```
[lsantoca@localhost lecture6]$ which jobs
which: no jobs in ( ... )
[lsantoca@localhost lecture6]$ jobs
[3]+  Running                  xpdf lecture6.pdf &
```

Un interpréteur est un langage de commandes.

On peut s'en servir pour écrire des commandes complexes.

# Les commandes internes

Résidents à l'intérieur du shell,  
ils n'engendrent pas de nouveaux processus.

Leurs but :

- améliorer la performance d'un commande externe,
- organiser les commandes en unités logiques.

Exemple :

```
[lsantoca@localhost lecture6]$ which jobs
which: no jobs in ( ... )
[lsantoca@localhost lecture6]$ jobs
[3]+  Running                  xpdf lecture6.pdf &
```

Un interpréteur est un langage de commandes.

On peut s'en servir pour écrire des commandes complexes.

# Les commandes internes

Résidents à l'intérieur du shell,  
ils n'engendrent pas de nouveaux processus.

Leurs but :

- améliorer la performance d'un commande externe,
- organiser les commandes en unités logiques.

Exemple :

```
[lsantoca@localhost lecture6]$ which jobs
which: no jobs in ( ... )
[lsantoca@localhost lecture6]$ jobs
[3]+  Running                  xpdf lecture6.pdf &
```

Un interpréteur est un langage de commandes.

On peut s'en servir pour écrire des commandes complexes.

# Programmation avec les « scripts »

Un script :

- suite de commandes contenus dans un fichier.  
Dans ce fichier, les commandes sont séparées par le caractère ' ; ' ou bien une nouvelle ligne.

Programmer avec les scripts :

- administration systèmes,
- prototypes des applications.

Scripts déconseillés si :

- utilisation intensive des ressources,
- application fiables, etc.

# Programmation avec les « scripts »

Un script :

- suite de commandes contenus dans un fichier.  
Dans ce fichier, les commandes sont séparées par le caractère ' ; ' ou bien une nouvelle ligne.

Programmer avec les scripts :

- administration systèmes,
- prototypes des applications.

Scripts déconseillés si :

- utilisation intensive des ressources,
- application fiables, etc.

# Programmation avec les « scripts »

Un script :

- suite de commandes contenus dans un fichier.  
Dans ce fichier, les commandes sont séparées par le caractère ' ; ' ou bien une nouvelle ligne.

Programmer avec les scripts :

- administration systèmes,
- prototypes des applications.

Scripts déconseillés si :

- utilisation intensive des ressources,
- application fiables, etc.

# Familles de shell

- Famille Bourne-Shell, dérivé du langage de commandes originaire de UNIX :
  - `ksh` : Korn-shell,
  - `bash` : Bourne-Again shell.
- Famille du C-Shell, dérivé langage de commandes originaire de la distribution BSD de UNIX :
  - `cs` : le C-shell,
  - `tcsh` : Turbo-C-shell.

Ici, on prende comme référence le Bourne-Again shell.

# Familles de shell

- Famille Bourne-Shell, dérivé du langage de commandes originaire de UNIX :
  - `ksh` : Korn-shell,
  - `bash` : Bourne-Again shell.
- Famille du C-Shell, dérivé langage de commandes originaire de la distribution BSD de UNIX :
  - `csh` : le C-shell,
  - `tcsch` : Turbo-C-shell.

Ici, on prende comme référence le Bourne-Again shell.



# Familles de shell

- Famille Bourne-Shell, dérivé du langage de commandes originaire de UNIX :
  - `ksh` : Korn-shell,
  - `bash` : Bourne-Again shell.
- Famille du C-Shell, dérivé langage de commandes originaire de la distribution BSD de UNIX :
  - `cs` : le C-shell,
  - `tcsh` : Turbo-C-shell.

Ici, on prende comme référence le Bourne-Again shell.

# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - Tests
  - Boucles

# Historique

```
[lsantoca@localhost lecture6]$ PS1="<\u : \!>$ "  
<lsantoca : 1019>$ echo abc  
abc  
<lsantoca : 1020>$ !1019 > fichier.txt  
echo abc > fichier.txt  
<lsantoca : 1032>$ cat fichier.txt  
abc
```

## Session : Le contrôle des tâches

```
[lsantoca@localhost lecture6]$ jobs
[3]-  Running                  xpdf _region_.pdf &
[4]+  Running                  xpdf lecture6.pdf &
[lsantoca@localhost lecture6]$ kill %3
[lsantoca@localhost lecture6]$ sleep 20 & sleep 20 &
[5] 4589
[6] 4590
[3]  Terminated              xpdf _region_.pdf
[lsantoca@localhost lecture6]$ echo aaa | wc -l | sleep 20 &
[7] 4593
[5]  Done                     sleep 20
[6]  Done                     sleep 20

[lsantoca@localhost lecture6]$ (sleep 20 ; ps) &
[7] 4625
[lsantoca@localhost lecture6]$ fg %7
( sleep 20; ps )
```

## Les alias, le caractère ~

```
[lsantoca@localhost lecture6]$ alias
alias gccW='gcc -Wall -pedantic'
alias mv='mv -i'
alias rm='rm -i'
[lsantoca@localhost lecture6]$ gccW interbloquage.c
```

```
[lsantoca@localhost lecture6]$ ls -l ~lsantoca/radiofrance.txt
-rw-r--r--    1 lsantoca lsantoca      171 oct 23  2003 /home/lsantoca/
[lsantoca@localhost lecture6]$ ls -l ~/radiofrance.txt
-rw-r--r--    1 lsantoca lsantoca      171 oct 23  2003 /home/lsantoca/
```

# Les variables

```
[lsantoca@localhost lecture6]$ CETTEVAR=coucou
[lsantoca@localhost lecture6]$ echo ${CETTEVAR}
coucou
[lsantoca@localhost lecture6]$ echo $CETTEVAR
coucou
[lsantoca@localhost lecture6]$ $CETTEVAR
bash: coucou: command not found
[lsantoca@localhost lecture6]$ env | grep CETTEVAR
[lsantoca@localhost lecture6]$ export CETTEVAR=coucou
[lsantoca@localhost lecture6]$ env | grep CETTEVAR
CETTEVAR=coucou
```

# Génération des noms de fichiers

Caractère	Interprétation
?	caractère quelconque
*	chaîne de caractères quelconque
[	début d'un ensemble
[!	début complément d'un ensemble
]	fin définition d'ensemble
-	marque intervalle dans un ensemble

## Session : noms des fichiers

```
[lsantoca@localhost lecture6]$ touch ab ac
[lsantoca@localhost lecture6]$ ls -l a?
-rw-r--r--    1 lsantoca lsantoca      0 nov  5 11:54 ab
-rw-r--r--    1 lsantoca lsantoca      0 nov  5 11:54 ac
[lsantoca@localhost lecture6]$ ls -d a*
ab ac alias.session a.out* auto/
[lsantoca@localhost lecture6]$ touch bc
[lsantoca@localhost lecture6]$ ls [a-c]c
ac bc
[lsantoca@localhost lecture6]$ ls [!b-z]c
ac
[lsantoca@localhost lecture6]$ ls [!a-c]c
ls: [!a-c]c: No such file or directory
```



## Le « quoting », les délimiteurs de chaînes

Opération	Interprétation
<code>\c</code>	le caractère c (à la place de son expansion)
<code>'...'</code>	la chaîne est protégée,
<code>"..."</code>	une seule chaîne, la substitution des variables est réalisée
<code>'...'</code>	exécution de la chaîne, affectation du résultat affiché par la commande sur le stdout

Remarque :

- en C, la barre `\` déclenche la substitution,
- ici, la barre `\` empêche la substitution.

## Le « quoting », les délimiteurs de chaînes

Opération	Interprétation
<code>\c</code>	le caractère c (à la place de son expansion)
<code>'...'</code>	la chaîne est protégée,
<code>"..."</code>	une seule chaîne, la substitution des variables est réalisée
<code>'...'</code>	exécution de la chaîne, affectation du résultat affiché par la commande sur le stdout

Remarque :

- en C, la barre `\` déclenche la substitution,
- ici, la barre `\` empêche la substitution.

## Session : délimiteurs de chaînes

```
[lsantoca@localhost lecture6]$ CETTEVAR=coucou
[lsantoca@localhost lecture6]$ echo 'CETTEVAR est egal à $CETTEVAR'
CETTEVAR est egal à $CETTEVAR
[lsantoca@localhost lecture6]$ echo "CETTEVAR est egal à $CETTEVAR"
CETTEVAR est egal à coucou
[lsantoca@localhost lecture6]$ CETAUTREVAR="voici \"$CETTEVAR\""
[lsantoca@localhost lecture6]$ echo $CETAUTREVAR
voici "coucou"
[lsantoca@localhost lecture6]$ CETAUTREVAR="Beaucoup d' espaces"
[lsantoca@localhost lecture6]$ echo $CETAUTREVAR
Beaucoup d' espaces
[lsantoca@localhost lecture6]$ echo "$CETAUTREVAR"
Beaucoup d' espaces
[lsantoca@localhost lecture6]$ CETTEVAR='date'
[lsantoca@localhost lecture6]$ echo "CETTEVAR est egal à $CETTEVAR"
CETTEVAR est egal à ven nov 5 12:08:17 CET 2004
```

# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - Tests
  - Boucles

# Lancement des processus

`commande` : commande simple.

`(commande)` : commande exécutée par un sous-shell.

`commande1 & commande2 & ... & commanden [&]` :  
exécution de `commande1` en arrière plan ( $i < n$ ),  
`commanden` en arrière plan si suivi par `&`.

`commande1 ; commande2 ; ... ; commanden` :  
exécution séquentielle des commandes.

# Lancement des processus

`commande` : commande simple.

`(commande)` : commande exécutée par un sous-shell.

`commande1 & commande2 & ... & commanden [&]` :  
exécution de `commandei` en arrière plan ( $i < n$ ),  
`commanden` en arrière plan si suivi par `&`.

`commande1 ; commande2 ; ... ; commanden` :  
exécution séquentielle des commandes.

# Lancement des processus

`commande` : commande simple.

`(commande)` : commande exécutée par un sous-shell.

`commande1 & commande2 & ... & commanden [&]` :  
exécution de `commandei` en arrière plan ( $i < n$ ),  
`commanden` en arrière plan si suivi par `&`.

`commande1 ; commande2 ; ... ; commanden` :  
exécution séquentielle des commandes.

# Lancement des processus

`commande` : commande simple.

`(commande)` : commande exécutée par un sous-shell.

`commande1 & commande2 & ... & commanden [&]` :  
exécution de `commandei` en arrière plan ( $i < n$ ),  
`commanden` en arrière plan si suivi par `&`.

`commande1 ; commande2 ; ... ; commanden` :  
exécution séquentielle des commandes.



## Session : construction des commandes

```
[lsantoca@localhost lecture6]$ (ps;exit 0)
```

PID	TTY	TIME	CMD
3319	pts/1	00:00:00	bash
3838	pts/1	00:00:00	bash
3839	pts/1	00:00:00	ps

```
[lsantoca@localhost lecture6]$ (ps)
```

PID	TTY	TIME	CMD
3319	pts/1	00:00:00	bash
3844	pts/1	00:00:00	ps

```
[lsantoca@localhost lecture6]$ sleep 1; ps
```

PID	TTY	TIME	CMD
3205	pts/1	00:00:00	bash
3379	pts/1	00:00:00	ps

```
[lsantoca@localhost lecture6]$ sleep 1 & ps
```

PID	TTY	TIME	CMD
3205	pts/1	00:00:00	bash
3385	pts/1	00:00:00	sleep
3386	pts/1	00:00:00	ps

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1|commande2|...|commanden :`  
exécution de commande1 avec stdout redirigé sur le stdin de commande1+1 (tube).

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1|commande2|...|commanden :`  
exécution de commande1 avec stdout redirigé sur le stdin de commande1+1 (tube).

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1|commande2|...|commanden :`  
exécution de commande1 avec stdout redirigé sur le stdin de commande1+1 (tube).

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1 | commande2 | ... | commanden :`  
exécution de commande1 avec stdout redirigé sur le stdin de commande1+1 (tube).

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1 | commande2 | ... | commanden :`  
exécution de commande1 avec stdout redirigé sur le stdin de commande1+1 (tube).

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1 | commande2 | ... | commanden :`  
exécution de commande1 avec stdout redirigé sur le stdin de commande1+1 (tube).

# Redirection, communication

`commande < chemin :`  
exécution commande avec stdin obtenu du fichier chemin.

`commande <&desc :`  
exécution commande avec stdin obtenu du descripteur desc.

`commande > chemin :`  
exécution de la commande avec stdout redirigé sur chemin (écrasement).

`commande desc> chemin :`  
exécution de la commande avec descripteur ouvert desc redirigé sur chemin.

`commande >& desc :`  
exécution de la commande avec stdout redirigé le fichier ouvert desc.

`commande >> chemin :`  
exécution commande avec stdout redirigé sur chemin (ajout).

`commande1|commande2|...|commanden :`  
exécution de commande<sub>i</sub> avec stdout redirigé sur le stdin de commande<sub>i+1</sub> (tube).



## Session : Redirection des E/S

```
[lsantoca@localhost lecture6]$ echo -e "ab\nbc" > tmpfile.txt
[lsantoca@localhost lecture6]$ cat tmpfile.txt
ab
bc
[lsantoca@localhost lecture6]$ grep a < tmpfile.txt
ab
[lsantoca@localhost lecture6]$ echo -e "ab\nbc" | grep a
ab
[lsantoca@localhost lecture6]$ echo cd >> tmpfile.txt;cat tmpfile.txt
ab
bc
cd
[lsantoca@localhost lecture6]$ echo erreur >&2
erreur
[lsantoca@localhost lecture6]$ ls ficnonexistant 2> fichier.txt
[lsantoca@localhost lecture6]$ cat fichier.txt
ls: ficnonexistant: No such file or directory
```

# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - Tests
  - Boucles

# Script

Suite de commandes contenue dans un fichier texte  
(suffixe usuel : `.sh`).

Un interpréteur exécute les commandes dans ce fichier.

Deux façons pour exécuter un script:

- en tapant :

```
$ sh monscript.sh
```

- si la première ligne est de la forme

```
#!/chemin_au_shell
```

par exemple :

```
#!/bin/bash
```

et le fichier a les droits en exécution, on peut taper :

```
$ monscript.sh
```

# Script

Suite de commandes contenue dans un fichier texte  
(suffixe usuel : `.sh`).

Un interpréteur exécute les commandes dans ce fichier.

Deux façons pour exécuter un script:

- en tapant :

```
$ sh monscript.sh
```

- si la première ligne est de la forme

```
#!/chemin_au_shell
```

par exemple :

```
#!/bin/bash
```

et le fichier a les droits en exécution, on peut taper :

```
$ monscript.sh
```

# Script

Suite de commandes contenue dans un fichier texte  
(suffixe usuel : `.sh`).

Un interpréteur exécute les commandes dans ce fichier.

Deux façons pour exécuter un script:

- en tapant :

```
$ sh monscript.sh
```

- si la première ligne est de la forme

```
#!/chemin_au_shell
```

par exemple :

```
#!/bin/bash
```

et le fichier a les droits en exécution, on peut taper :

```
$ monscript.sh
```

## Session : scripts

```
[lsantoca@localhost lecture6]$ echo echo bonjour > monscript1.sh
[lsantoca@localhost lecture6]$ cat monscript1.sh
echo bonjour
[lsantoca@localhost lecture6]$ sh monscript1.sh
bonjour
[lsantoca@localhost lecture6]$ echo '#!' 'which sh' > monscript2.sh
[lsantoca@localhost lecture6]$ cat monscript1.sh >> monscript2.sh
[lsantoca@localhost lecture6]$ cat monscript2.sh
#!/bin/sh
echo bonjour
[lsantoca@localhost lecture6]$ chmod 711 monscript2.sh
[lsantoca@localhost lecture6]$ ./monscript2.sh
bonjour
```

# Remarques

- # : ce qui suit ce caractère est un commentaire,
- ; : ce caractère est un séparateur de commandes, non un terminateur de commandes.

Exemple :

```
...  
# la prochaine ligne contient deux commandes et un seul caractère ;  
echo bonjour; echo le monde  
...
```

# Remarques

- # : ce qui suit ce caractère est un commentaire,
- ; : ce caractère est un séparateur de commandes, non un terminateur de commandes.

Exemple :

```
...  
# la prochaine ligne contient deux commandes et un seul caractère ;  
echo bonjour; echo le monde  
...
```



# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - Tests
  - Boucles

# Les paramètres et la valeur de retour

`$0, $1, ... , $9, ${10}, ... :`  
les paramètres passés aux script.

`$*, $@ :` tous les paramètres,

`$# :` le nombre de paramètres.

`$? :`

la valeur de retour (produite par `exit`)  
de la dernière commande.

La commande `shift` :

`poubelle <- $1 <- $2, $2 <- $3 ...`

# Les paramètres et la valeur de retour

`$0, $1, ... , $9, ${10}, ... :`  
les paramètres passés aux script.

`$*`, `$@` : tous les paramètres,

`$#` : le nombre de paramètres.

`$?` :

la valeur de retour (produite par `exit`)  
de la dernière commande.

La commande `shift` :

`poubelle <- $1 <- $2, $2 <- $3 ...`

# Les paramètres et la valeur de retour

`$0, $1, ... , $9, ${10}, ... :`  
les paramètres passés aux script.

`$*`, `$@` : tous les paramètres,

`$#` : le nombre de paramètres.

`$?` :

la valeur de retour (produite par `exit`)  
de la dernière commande.

La commande `shift` :

```
poubelle <- $1 <- $2, $2 <- $3 ...
```

# Les paramètres et la valeur de retour

- `$0, $1, ... , $9, ${10}, ... :`  
les paramètres passés aux script.
- `$*`, `$@` : tous les paramètres,
- `$#` : le nombre de paramètres.
- `$?` :  
la valeur de retour (produite par `exit`)  
de la dernière commande.

La commande `shift` :

```
poubelle <- $1 <- $2, $2 <- $3 ...
```

# Les paramètres et la valeur de retour

`$0, $1, ... , $9, ${10}, ... :`  
les paramètres passés aux script.

`$*, $@ :` tous les paramètres,

`$# :` le nombre de paramètres.

`$? :`  
la valeur de retour (produite par `exit`)  
de la dernière commande.

La commande `shift` :

```
poubelle <- $1 <- $2, $2 <- $3 ...
```

# Programme : parametres.sh

```
1 : #! /bin/bash
2 :
3 : echo Nombre paramètres : $#
4 : echo Les paramètres : $*
5 : echo De même : $@
6 :
7 : echo -n "Une autre fois : "
8 : while [ "$1x" != "x" ]
9 : do
10 :     echo -n "$1 "
11 :     shift
12 : done
13 :
14 : echo
15 : exit 0
```

## Session : parematres

```
[lsantoca@localhost lecture6]$ parametres.sh 1 2 3 4 5 6  
Nombre paramètres : 6  
Les paramètres : 1 2 3 4 5 6  
De même : 1 2 3 4 5 6  
Une autre fois : 1 2 3 4 5 6  
[lsantoca@localhost lecture6]$ echo $?  
0
```



# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - **Tests**
  - Boucles

# La commande if

Teste si la valeur de retour d'un commande est « succès »:

```
if cmd
then
    ...
[ else
    ... ]
fi
```

Par exemple :

```
if grep bash fichier > /dev/null
then
    echo trouvé
else
    echo non trouvé
fi
```

# La commande if

Teste si la valeur de retour d'une commande est « succès »:

```
if cmd
then
    ...
[ else
    ... ]
fi
```

Par exemple :

```
if grep bash fichier > /dev/null
then
    echo trouvé
else
    echo non trouvé
fi
```

# La commande if

Teste si la valeur de retour d'une commande est « succès »:

```
if cmd
then
    ...
[ else
    ... ]
fi
```

Par exemple :

```
if grep bash fichier > /dev/null
then
    echo trouvé
else
    echo non trouvé
fi
```

## Les commandes test et [...]

Il s'agit d'une commande interne et externe.  
Retourne 0 si le test est vrai.

```
$ test -f lecture6.tex  
$ echo $?  
0
```

La commande suivante est une synonyme.

```
$ [ -f lecture6.tex ]  
$ echo $?  
0
```

## Les commandes test et [...]

Il s'agit d'une commande interne et externe.  
Retourne 0 si le test est vrai.

```
$ test -f lecture6.tex  
$ echo $?  
0
```

La commande suivante est une synonyme.

```
$ [ -f lecture6.tex ]  
$ echo $?  
0
```

## Les commandes test et [...]

Il s'agit d'une commande interne et externe.  
Retourne 0 si le test est vrai.

```
$ test -f lecture6.tex  
$ echo $?  
0
```

La commande suivante est une synonyme.

```
$ [ -f lecture6.tex ]  
$ echo $?  
0
```

## Session : if + test

```
$ if [ -f lecture6.tex ]; then \  
> echo lecture6.tex est regulier; \  
> else echo lecture6.tex n'est pas regulier; fi  
lecture6.tex est regulier  
$
```



# Tests sur les fichier

**-e** : fichier existe,

**-s** : fichier a taille non-nulle,

**-f, -d, -b, -c, -p, -L, -S** :

fichier est régulier, répertoire, périphérique bloque,  
périphérique caractère, tube, lien symbolique, socket,

**-r, -w, -x** :

le script a permission de lecture, écriture, exécution sur le  
fichier.

etc., voir `man test`.

# Tests sur les fichier

**-e** : fichier existe,

**-s** : fichier a taille non-nulle,

**-f, -d, -b, -c, -p, -L, -S** :

fichier est régulier, répertoire, périphérique bloque,  
périphérique caractère, tube, lien symbolique, socket,

**-r, -w, -x** :

le script a permission de lecture, écriture, exécution sur le  
fichier.

etc., voir `man test`.

# Tests sur les fichiers

`-e` : fichier existe,

`-s` : fichier a taille non-nulle,

`-f`, `-d`, `-b`, `-c`, `-p`, `-L`, `-S` :

fichier est régulier, répertoire, périphérique bloque,  
périphérique caractère, tube, lien symbolique, socket,

`-r`, `-w`, `-x` :

le script a permission de lecture, écriture, exécution sur le  
fichier.

etc., voir `man test`.

# Tests sur les fichier

`-e` : fichier existe,

`-s` : fichier a taille non-nulle,

`-f`, `-d`, `-b`, `-c`, `-p`, `-L`, `-S` :

fichier est régulier, répertoire, périphérique bloque,  
périphérique caractère, tube, lien symbolique, socket,

`-r`, `-w`, `-x` :

le script a permission de lecture, écriture, exécution sur le  
fichier.

etc., voir `man test`.

# Opérateurs (binaires) de test

Comparaison sur les entiers :

`-eq`, `-ne` : égalité, inégalité,

`-gt`, `-ge`, `-lt` `-le` :

(strictement) plus grand, (strictement) plus petit.

Opérateurs sur les chaînes de caractères :

`=`, `==`, `!=` : égalité, inégalité,

`<`, `>` : strictement plus petit/grand,

`-z`, `-n` : longueur nulle/non-nulle,

etc., voir `man test`.

# Opérateurs (binaires) de test

Comparaison sur les entiers :

`-eq`, `-ne` : égalité, inégalité,

`-gt`, `-ge`, `-lt` `-le` :

(strictement) plus grand, (strictement) plus petit.

Opérateurs sur les chaînes de caractères :

`=`, `==`, `!=` : égalité, inégalité,

`<`, `>` : strictement plus petit/grand,

`-z`, `-n` : longueur nulle/non-nulle,

etc., voir `man test`.

# Opérateurs (binaires) de test

Comparaison sur les entiers :

`-eq`, `-ne` : égalité, inégalité,

`-gt`, `-ge`, `-lt` `-le` :

(strictement) plus grand, (strictement) plus petit.

Opérateurs sur les chaînes de caractères :

`=`, `==`, `!=` : égalité, inégalité,

`<`, `>` : strictement plus petit/grand,

`-z`, `-n` : longueur nulle/non-nulle,

etc., voir `man test`.

# Opérateurs (binaires) de test

Comparaison sur les entiers :

`-eq`, `-ne` : égalité, inégalité,

`-gt`, `-ge`, `-lt` `-le` :

(strictement) plus grand, (strictement) plus petit.

Opérateurs sur les chaînes de caractères :

`=`, `==`, `!=` : égalité, inégalité,

`<`, `>` : strictement plus petit/grand,

`-z`, `-n` : longueur nulle/non-nulle,

etc., voir `man test`.



# Opérateurs (binaires) de test

Comparaison sur les entiers :

`-eq`, `-ne` : égalité, inégalité,

`-gt`, `-ge`, `-lt` `-le` :

(strictement) plus grand, (strictement) plus petit.

Opérateurs sur les chaînes de caractères :

`=`, `==`, `!=` : égalité, inégalité,

`<`, `>` : strictement plus petit/grand,

`-z`, `-n` : longueur nulle/non-nulle,

etc., voir `man test`.

# Session : exemples

```
$ a="1"; b="2"
$ [ "$a" -le "$b" ]; echo $?
0
$ b="a"
$ [ "$a" -le "$b" ]; echo $?
bash: [: a: integer expression expected
2
$ [ "$a" \< "$b" ]; echo $?
0
$ a="c"
$ [ "$a" \< "$b" ]; echo $?
1
$ [ "$a" < "$b" ]; echo $?
bash: a: No such file or directory
1
$ [[ "$a" < "$b" ]]; echo $?
1
$ [ -z "" ]; echo -n $?; [ -z ]; echo $?
00
$ [ ! -z "" ]; echo -n $?; [ ! 2 -le 1 ]; echo $?
10
```

# Opérateurs logiques (dans un test)

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

Remarque :

à partir de Bash 2.02 on peut utiliser `[[ ... ]]` à la place de `[...]` avec syntaxe type C  
(entre autres : les opérateurs `&&` et `||` permis).

# Opérateurs logiques (dans un test)

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

Remarque :

à partir de Bash 2.02 on peut utiliser `[[ ... ]]` à la place de `[...]` avec syntaxe type C  
(entre autres : les opérateurs `&&` et `||` permis).

# Opérateurs logiques hors test

`&&`, `||`, `!` : sémantique semblable à celle du C.

Expression	son équivalent
<code>cmd1 &amp;&amp; cmd2</code>	<code>if cmd1; then cmd2; fi</code>
<code>if [ test1 ] &amp;&amp; [ test2 ]</code>	<code>if [ test1 -a test2 ]</code>

Pas possible	Correcte
<code>if [ test1 &amp;&amp; test2 ]</code>	<code>if [[ test1 &amp;&amp; test2 ]]</code>

# Opérateurs logiques hors test

`&&`, `||`, `!` : sémantique semblable à celle du C.

Expression	son équivalent
<code>cmd1 &amp;&amp; cmd2</code> <code>if [ test1 ] &amp;&amp; [ test2 ]</code>	<code>if cmd1; then cmd2; fi</code> <code>if [ test1 -a test2 ]</code>

Pas possible	Correcte
<code>if [ test1 &amp;&amp; test2 ]</code>	<code>if [[ test1 &amp;&amp; test2 ]]</code>

# Tests arithmétiques avec ((...))

La commande

(( expr ))

retourne 1, si `expr` est évaluée à 0, et 0 sinon.

```
$ (( 1 )); echo -n "$? " ; (( 0 )); echo $?  
0 1  
$ (( 1 + 33 < 35 )); echo $?  
0  
$ (( -1 )); echo -n "$? " ; (( 33 )); echo $?  
0 0  
$ (( abs )); echo -n "$? "; [ abs ] ; echo $?  
1 0
```

Opérateurs arithmétiques :

`+`, `-`, `*`, `/`, `**`, `%` : comme d'habitude.

# Tests arithmétiques avec ((...))

La commande

```
(( expr ))
```

retourne 1, si `expr` est évaluée à 0, et 0 sinon.

```
$ (( 1 )); echo -n "$? " ; (( 0 )); echo $?  
0 1  
$ (( 1 + 33 < 35 )); echo $?  
0  
$ (( -1 )); echo -n "$? " ; (( 33 )); echo $?  
0 0  
$ (( abs )); echo -n "$? "; [ abs ] ; echo $?  
1 0
```

Opérateurs arithmétiques :

`+`, `-`, `*`, `/`, `**`, `%` : comme d'habitude.



# Tests arithmétiques avec ((...))

La commande

```
(( expr ))
```

retourne 1, si `expr` est évaluée à 0, et 0 sinon.

```
$ (( 1 )); echo -n "$? " ; (( 0 )); echo $?  
0 1  
$ (( 1 + 33 < 35 )); echo $?  
0  
$ (( -1 )); echo -n "$? " ; (( 33 )); echo $?  
0 0  
$ (( abs )); echo -n "$? "; [ abs ] ; echo $?  
1 0
```

Opérateurs arithmétiques :

`+`, `-`, `*`, `/`, `**`, `%` : comme d'habitude.

# Aguillage

```
case expr in
    motif1)
        suitecoms
    ;;

    motif2)
        suitecoms
    ;;

    ...

    *)

esac
```

# Programme : /etc/init.d/adsl (I)

```
1 : #!/bin/sh
2 : #
3 : # adsl                      This script starts or stops an ADSL connection
4 : #
5 : # chkconfig: 2345 99 01
6 : # description: Connects to ADSL provider
7 : #
8 : # LIC: GPL
9 : #
10 : # Copyright (C) 2000 Roaring Penguin Software Inc.  This software may
11 : # be distributed under the terms of the GNU General Public License, version
12 : # 2 or any later version.
13 :
14 : # Source function library if it exists
15 : test -r /etc/rc.d/init.d/functions && . /etc/rc.d/init.d/functions
16 :
17 : # From AUTOCONF
18 : prefix=/usr
19 : exec_prefix=/usr
20 :
21 : # Paths to programs
22 : START=/usr/sbin/adsl-start
23 : STOP=/usr/sbin/adsl-stop
24 : STATUS=/usr/sbin/adsl-status
```

# Programme : /etc/init.d/adsl (II)

```
25 : case "$1" in
26 :     start)
27 :         gprintf "Bringing up ADSL link"
28 :
29 :         $START
30 :         if [ $? = 0 ] ; then
31 :             touch /var/lock/subsys/adsl
32 :             echo_success
33 :         else
34 :             echo_failure
35 :         fi
36 :         echo ""
37 :         ;;
38 :
39 :     stop)
40 :         gprintf "Shutting down ADSL link"
41 :
42 :         $STOP > /dev/null 2>&1
43 :         if [ $? = 0 ] ; then
44 :             rm -f /var/lock/subsys/adsl
45 :             echo_success
46 :         else
47 :             echo_failure
48 :         fi
49 :         echo ""
50 :         ;;
```

## Programme : /etc/init.d/adsl (III)

```
51 :  
52 :     restart|reload)  
53 :         $0 stop  
54 :         $0 start  
55 :         ;;  
56 :  
57 :     status)  
58 :         $STATUS  
59 :         ;;  
60 :  
61 :     *)  
62 :         gprintf "Usage: adsl {start|stop|restart|reload|status}\n"  
63 :         exit 1  
64 : esac  
65 :  
66 : exit 0
```

# Plan

- 1 Introduction aux interpréteurs
  - Généralités
  - Les mécanismes de substitution et d'expansion
  - Lancement des processus, redirection, communication
- 2 Programmation avec les scripts
  - Introduction
  - Variables spéciales
  - Tests
  - Boucles

# La boucle for

```
for arg in list
do
    commandes
done
```

Exemple :

```
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo $i
done
echo
exit 0
```

# La boucle for

```
for arg in list
do
    commandes
done
```

Exemple :

```
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo $i
done
echo
exit 0
```



# La boucle while

```
while condition
do
    commandes
done
```

Exemple :

```
#!/bin/bash
i=1
MAX=10
while [ "$i" -le "$MAX" ]
do
    echo -n "$i "
    i='expr i + 1' # on peut aussi utiliser i=$(( $i + 1 ))
done
echo
exit 0
```

# La boucle while

```
while condition
do
    commandes
done
```

Exemple :

```
#!/bin/bash
i=1
MAX=10
while [ "$i" -le "$MAX" ]
do
    echo -n "$i "
    i='expr i + 1' # on peut aussi utiliser i=$(( $i + 1 ))
done
echo
exit 0
```

# Programme : liensymboliques.sh

```
1 : #! /bin/bash
2 : # liensymboliques.sh : liste les liens symboliques
3 : #+ dans un répertoire
4 :
5 : if [ $# -lt 1 ]; then ## La même chose : REPERTOIRE={1-'pwd'}
6 :     REPERTOIRE='pwd'
7 : else
8 :     REPERTOIRE=$1
9 : fi
10 :
11 : # Expliquer la raison de ! dans la commande suivante
12 : [ -d $REPERTOIRE ] || ! echo $REPERTOIRE : pas un répertoire || exit -1
13 :
14 : echo -e "Liens symboliques dans le répertoire\n\t\"$REPERTOIRE\" : \n"
15 : for fichier in `ls ${REPERTOIRE}/*`; do
16 :     if [ -L "$fichier" ]; then
17 :         echo "$fichier"
18 :     fi
19 : done | sort
20 :
21 : echo
22 : exit 0
```

## Session : liens symboliques

```
[lsantoca@localhost lecture6]$ chmod 755 liensymboliques.sh
```

```
[lsantoca@localhost lecture6]$ touch a b; ln -s a c ; ln -s b d
```

```
[lsantoca@localhost lecture6]$ liensymboliques.sh
```

Liens symboliques dans le répertoire

```
"/home/lsantoca/courses/2004/systemes/tex/slides/lecture6" :
```

```
/home/lsantoca/courses/2004/systemes/tex/slides/lecture6/c
```

```
/home/lsantoca/courses/2004/systemes/tex/slides/lecture6/d
```

```
[lsantoca@localhost lecture6]$ liensymboliques.sh .
```

Liens symboliques dans le répertoire

```
". " :
```

```
./c
```

```
./d
```

```
[lsantoca@localhost lecture6]$ liensymboliques.sh coucou
```

coucou : pas un répertoire