

Unix et Programmation Shell

Philippe Langevin

département d'informatique
UFR sciences et technique
université du sud Toulon Var

Automne 2013

brouillon en révision

- site du cours :
<http://langevin.univ-tln.fr/cours/UPS/upsh.html>
- localisation du fichier :
<http://langevin.univ-tln.fr/cours/UPS/doc/bash.pdf>

dernières modifications

upsh.tex	2024-01-28	21:05:32.653614398	+0100
tools.tex	2024-01-28	21:05:32.653614398	+0100
term.tex	2024-01-28	21:05:32.651614387	+0100
syntaxe.tex	2024-01-28	21:05:32.650614381	+0100
shell.tex	2024-01-28	21:05:32.649614376	+0100
prologue.tex	2024-01-28	21:05:32.648614371	+0100
proc.tex	2024-01-28	21:05:32.646614360	+0100
pipe.tex	2024-01-28	21:05:32.645614355	+0100
perm.tex	2024-01-28	21:05:32.645614355	+0100
man.tex	2024-01-28	21:05:32.643614344	+0100
part.tex	2024-01-28	21:05:32.644614349	+0100
file.tex	2024-01-28	21:05:32.642614338	+0100
direct.tex	2024-01-28	21:05:32.641614333	+0100
bash.tex	2024-01-28	21:05:32.639614322	+0100

9 - bash

- ligne de commande
- analyse
- quote
- expansion
- fonction et tableau
- commandes
- pipeline
- commandes séquentielles
- alternative
- test
- itérations
- choix multiple
- menus
- options

Edition

`Bash` utilise la bibliothèque `readline` qui inclut la *complétion automatique* des noms de commandes et de fichiers (tabulation) ainsi que la *navigation* dans l'historique des commandes.

- `set` permet de sélectionner un éditeur : `vi`, `emacs` (défaut).
- 8 Command Line Editing

<code>readline</code>	<code>emacs</code>	<code>vi</code>
beginning-of-line	CTRL-a	0
end-of-line	CTRL-e	\$
clear-screen	CTRL-l	
reverse-search-history	CTRL-r	
forward-search-history	CTRL-s	
kill-line	CTRL-k	d\$
backward-kill-line	CTRL-x Rubout	d0
unix-line-discard	CTRL-u	dd
unix-word-rubout	CTRL-w	dw
yank	CTRL-y	p
display-shell-version	CTRL-x C-v	

- `set -o vi`
- `set -o emacs`
- `bind, .inputrc`

readline

```
1 static char *ligne = (char *) NULL;
2 char *nligne (void) {
3     if (ligne) {
4         free (ligne);
5         ligne = (char *) NULL;
6     }
7     ligne = readline ("prompt:");
8     if (ligne && *ligne)
9         add_history (ligne);
10    return (ligne);
11 }
12 int main(void) {
13     while ( strcmp(nligne(), "quit"));
14     return 0;
15 }
```

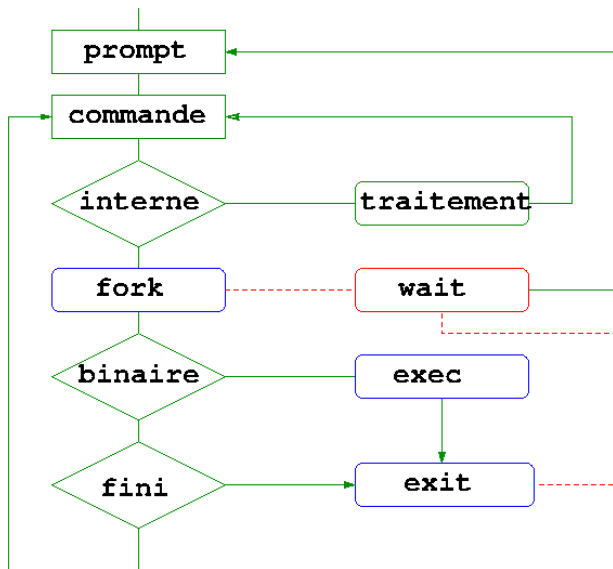
traitement des commandes

Le prompt du shell matérialise une ligne de commande invitant l'utilisateur à soumettre une instruction : une chaîne de caractères analysée par le shell pour lancer des commandes.

- interne : traitée dans le processus.
- externe : `clone` + `execve`.
- complexe : `clone`

```
11:47 pl@mic/script> echo $PS1  
\  
A pl@$MACHINE/\W
```

boucle du shell



Interprétation des lignes

`bash` segmente la ligne en mots (tokens), le premier mot est généralement une commande :

- 1 alias
 - `.bashrc`
 - `alias`
- 2 commande interne
 - `enable`
- 3 binaire
 - variable **PATH**
 - `which`

les autres arguments sont des options ou paramètres de la commande.

what's for ?

Un exemple proposé par Chet Ramey :

```
1 for for in for; do for=for; done; echo $for
```

what's for ?

Un exemple proposé par Chet Ramey :

```
1 for for in for; do for=for; done; echo $for
```

Nous ne tenterons pas de percer les mystères de la grammaire du shell `bash`, tout au plus quelques points extraits du manuel :

```
~> wget http://www.gnu.org/software/bash/manual/  
bash.html  
~> man bash
```

Ne sont pas vraiment abordés ici :

- les fonctions,
- les tableaux,
- les tableaux associatifs.

jeux de caractères

Méta-caractères (séparateurs) :

| & ; () < >

Opérateurs de contrôle :

| |& || && ; &

Opérateurs de redirection :

< << > >> >& <<<

Paramètres spéciaux :

* @ # ? \$! 0 _ ~

Mots réservés :

```
1 ! case do done elif else esac fi for function if
2 in select then until while { } time [[ ]]
```

apostrophe

Les protections permettent de forcer l'interpréteur à ignorer la signification spéciale de certains caractères ou mots. Les protections peuvent être utilisées pour empêcher le traitement des caractères spéciaux, éviter la reconnaissance des mots-réservés ou empêcher le développement des paramètres. Il y a trois mécanismes de protection :

- le caractère d'échappement (antislash)
- les apostrophes (quote)
- les guillemets (double-quote)

quoting

- \ protège le caractère qu'il précède.
- les apostrophes protègent une chaîne de caractères.

entre deux guillemets :

- \$ est toujours actif.
- \ affecte

```
$ \ " <nl>
```

cow quotes

```
↪ cowsay -TU 'how to quote ?'
```

```
-----  
< how to quote ? >
```

```

      \      ^  ^
      \      --  ^
      \      (oo)\ -----
      \      (--) \           )\ /\
      \      U  ||-----w  |
      \           ||           ||

```

cow explains how to quote

```
1 #!/bin/bash -v
2 x=quote; cowsay -T\\ / "how to '$x' ?"
```

```
#!/bin/bash -v
x=quote; cowsay -T\\ / "how to '$x' ?"
```

```
-----
< how to 'quote' ? >
```

```

  \      ^  ^
  \      --  ^
  \      (oo)\ -----
  \      (--) \          )\\ /
  \      \ /  ||-----w  |
  \      ||      ||      ||

```


quotes

```
↪ cowsay -f small "' $$" ' "$HISTSIZE" "\ $$"
```

```
-----  
< " $$" 512 $$ >
```

```

  \
   \
      ,---,
     (oo) -----
     (--)          )\
      ||--|| *

```

digression

```

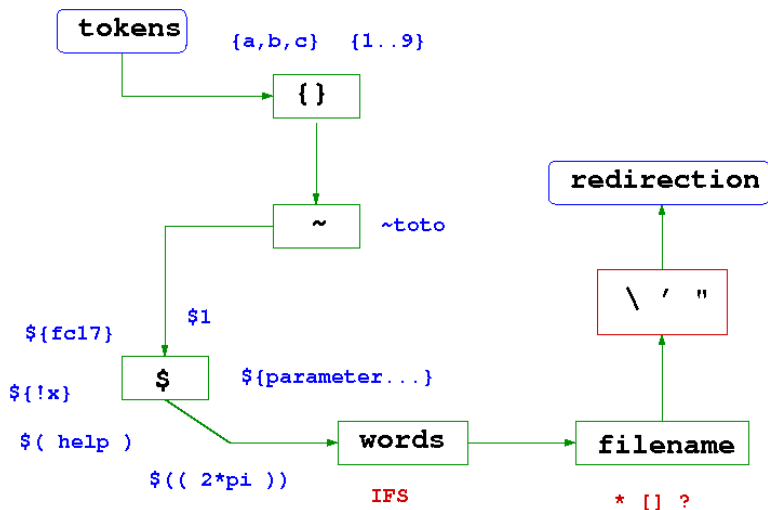
1 #!/bin/bash -v
2 z=hello
3 printf -v z "%s $USER" $z
4 echo $z
5 hello pl
6 man ascii | grep -o '[0-9][0-9 ]*A$'
7 101 65 41 A
8 x=A
9 printf "%3o %2x %3d %s" \"$x \"$x \"$x" $x
10 101 41 65 A z=65
11 printf -vx "0%3o" $z
12 echo -e "\\$x"
13 A
14 printf "\\$(printf "%03o" $z)
15 printf "%03o" $z)

```

Analyse des commandes

- 1 Lecture : fichier, chaîne, terminal
- 2 Analyse lexicale : application des quotes, suite de tokens (opérateur, mot) séparés par des métacaractères.
- 3 Traitement des alias.
- 4 Analyse syntaxique des commandes.
- 5 Expansion des tokens : noms de fichiers, commandes et arguments.
- 6 Traitement des redirections
- 7 Exécution des commandes.
- 8 Terminaison des fils, status.

expansion des tokens



{a,b,c,...} : accolade (brace)

```
1 #!/bin/bash
2 for x in {alice , eve, bob} .{X,Y}
3 do
4     touch /tmp/$x.txt
5 done
6 ls /tmp/[a-z]*.[XYZ]*.txt
```

```
/tmp/ alice .X. txt
/tmp/ alice .Y. txt
/tmp/ bob .X. txt
/tmp/ bob .Y. txt
/tmp/ eve .X. txt
/tmp/ eve .Y. txt
```

{1..n} : énumération

```
1 #!/bin/bash
2 if [ ! -f /tmp/host.txt ] ; then
3     for x in {1..254} ; do
4         host 193.49.96.$x
5     done | grep -v nat > /tmp/host.txt
6 fi
7 cut -d ' ' -f 1,5 < /tmp/host.txt | head -5
```

```
1.96.49.193.in-addr.arpa rhodes.univ-tln.fr.
2.96.49.193.in-addr.arpa mail.univ-tln.fr.
3.96.49.193.in-addr.arpa lorcanne.univ-tln.fr.
4.96.49.193.in-addr.arpa dpt-info.univ-tln.fr.
5.96.49.193.in-addr.arpa serecom.univ-tln.fr.
```

tilde

```
1 #!/bin/bash -v
2
3 echo -n ~{alice,pl , bob,guest}
4 ~alice /home/pl ~bob /home/guest
5 for gus in {alice , pl , bob,guest}
6 do
7     echo ~$gus
8 done
9 ~alice
10 ~pl
11 ~bob
12 ~guest
```

argument et paramètre de position

```
1 #!/bin/bash -v
2 echo $0 : $*
3 set a b c
4 echo $*
5 while [ $# -gt 0 ]; do
6     echo $1 : $*
7     shift
8 done
```


décalage des paramètres de position

```
1 #!/bin/bash -v
2 echo $0 : $*
3 ./shift .sho :
4 set a b c
5 echo $*
6 a b c
7 while [ $# -gt 0 ]; do
8     echo $1 : $*
9     shift
10 done
11 a : a b c
12 b : b c
13 c : c
```

tri des arguments

```
1 #!/bin/bash
2 if [ $# = 0 ]; then
3     set 2 1 4 3 6 5 8 7
4 fi
5 echo $* | tr ' ' '\n' \
6         | sort -n | tr '\n' ' ' \
```

paramètres spéciaux

```
1 #!/bin/bash -v
2 echo opt=$- pid=$$
3 opt=hvB pid=2615
4 bash -c 'echo child =$$' &
5 echo hello > /tmp/spec
6 echo $?
7 0
8 echo hello > /root/tmp
9 ./spec.sho: line 6: /root/tmp: Permission non accordée
10 echo $? $!
11 1 2617
12 child =2617
```

séparateur interne de champs

```
1 #!/bin/bash -v
2 set gnu is not unix
3
4 echo "$*"
5 gnu is not unix
6 echo "$@"
7 gnu is not unix
8
9 IFS=:
10
11 echo "$*"
12 gnu:is : not: unix
13 echo "$@"
14 gnu is not unix
```

danger !

Le caractère ! est utilisé :

- opérateur logique :!=
- **pid** dernier job \$!
- indirection \${!...}
- historique !



```
39 echo hello
40 touch this
41 echo coucou
42 cat file
43 history
~> !39
echo hello
~> !-5
touch this
~> !ec
echo hello
~> !? file
cat file
~> ^file^foo^
```

nom de fichier

Une expression contenant un des

[*] ?

donne la liste des noms de fichiers du répertoire courant qui sont en correspondance avec l'expression :

- * : chaîne arbitraire
- [] : ensemble
- ? : caractère
- Absence de correspondance : attention !

filename

```
1 #!/bin/bash -v
2 if [ ! -d tmp ] ; then mkdir tmp ;fi
3 cd tmp
4 touch abc
5 touch cba
6 echo a[abc]b
7 a[abc]b
8 echo a[abc]c
9 abc
10 shopt -s failglob
11 echo a[abc]b
12 ./filename.sho: line 9: Pas de correspondance : a[abc]b
```

`$(commande)`

```
1 #!/bin/bash -v
2
3 man -s 1 -f bc
4
5 res=$( bc <<< "2^(2^5) + 1" )
6
7 factor $res
```

Il existe une syntaxe synonyme à base d'anticotes.

\$((arithmétique))

```

1 #!/bin/bash -v
2 x=010
3 y=2
4 echo $(( x * y ))
5 16
6
7 let y++
8 echo $y
9 3
10
11 let z=x*y
12 echo $z
13 24

```

```

1 #!/bin/bash -v
2
3 echo $(( 2#1000000 ))
4 64
5 echo $(( 16#abcdef ))
6 11259375
7
8 x=$(( 16#abcdef ))
9 echo $( bc <<< "obase=16; $x"
10 )
11 bc <<< "obase=16; $x" )
12 ABCDEF

```

`${parameter : [-+= ?] }`

```
1 #!/bin/bash -v
2 echo ${foo:-hello world}
3 hello world
4 echo ${bar:=hello world}
5 hello world
6 echo foo=$foo bar=$bar
7 foo= bar=hello world
8 echo ${bar:+salut tout le monde}
9 salut tout le monde
10 echo ${bar:?}
11 hello world
12 echo ${foo:?}
13 ./para1.sho: line 7: foo : parametre vide ou non defini
```

chaîne, sous-chaîne, préfixe, suffixe

```
1 #!/bin/bash -v
2 str = 'Hello everybody out there using minix'
3 echo ${str:6}
4 everybody out there using minix
5 echo ${str:6:10}
6 everybody
7 # sharp prints suffixe
8 echo ${str#*y}
9 body out there using minix
10 echo ${str##*y}
11 out there using minix
12 # percent prints prefixe
13 echo ${str%%o*} ${str%o*}
14 Hell Hello everybody
```

`}${parameter/motif/remplace}`

```

1 #!/bin/bash -v
2 claim='I do not believe everybody should learn to code'
3 echo }${claim/d*t /really \ }
4 I really believe everybody should learn to code
5 echo }${claim//d*t/sometime d}
6 I sometime do code
7 echo \${str{^,^^,\,,\,\,} pat\}
8 }${str^pat} ${str^^pat} ${str,pat} ${str,,pat}
9 claim=${claim^^*}
10 echo }${claim/#D*T/}
11 I DO NOT BELIEVE EVERYBODY SHOULD LEARN TO
    CODE
12 echo }${claim/#I*Y/YOU} BASH!
13 YOU SHOULD LEARN TO CODE BASH!

```

`${!variable}` `${!prefix}` : déréréférencement !

```
~> foo=hello; ptr=foo ; echo ${!ptr}
hello
```

```
~> echo ${!H*}
HISTCMD HISTCONTROL HISTFILE HISTFILESIZE
HISTSIZE HOME HOSTNAME HOSTTYPE
```

```
~> env | grep HIST
HISTSIZE=512
HISTFILESIZE=2048
HISTCONTROL=ignoredups
HISTFILE=/home/pl/.compil_history
```

Bash :

`${!variable}` `${!prefix}` : déréréféréncement !

```
→ foo=hello; ptr=foo ; echo ${!ptr}
hello
```

```
→ echo ${!H*}
HISTCMD HISTCONTROL HISTFILE HISTFILESIZE
HISTSIZE HOME HOSTNAME HOSTTYPE
```

```
→ env | grep HIST
HISTSIZE=512
HISTFILESIZE=2048
HISTCONTROL=ignoredups
HISTFILE=/home/pl/.compil_history
```

Bash :

```
→ for x in ${!HIST*}; do echo $x=${!x}; done
```

fonction

```
1 #!/bin/bash
2 function chr {
3     printf \\$( printf '%03o' $1 )
4 }
5 function ord {
6     printf '%d' "'$1'"
7 }
8
9 for x in $* ; do
10     echo -ne \\t$x: ; ord $x
11 done
```

```
→ ./fonction.sho {A..E}
```

A:65

B:66

C:67

D:68

E:69

globale par défaut

```
1 #!/bin/bash -v
2 foo=globale
3
4 function myproc {
5 local foo=locale
6 echo $foo
7 echo $bar
8 }
9
10 bar=$foo
11
12 myproc
13 locale
14 globale
```


tableau

```
1 #!/bin/bash -v
2 week=(lundi mardi mercredi jeudi vendredi samedi dimanche)
3 echo x=$week bad=$week[1] good=${week[1]}
4 x=lundi bad=lundi[1] good=mardi
5 echo dim:{#week[*]} val:${week[*]}
6 dim:{#week[*]} val:lundi mardi mercredi jeudi vendredi
   samedi dimanche
7 echo ind:${!week[*]}
8 ind:0 1 2 3 4 5 6
9 for i in ${!week[*]} ; do
10   day=${week[i]}
11   echo -n ${day:0:1}
12 done
13 Immjvsd
```

tableau associatif

```

1 #!/bin/bash
2 declare -A count
3 for w in $( cat $0 | sed 's/[^a-z]/\n/g' ) ; do
4   let count[$w]++
5 done
6
7 for w in ${!count[@]} ; do
8   echo -n ' $w:${count[$w]}
9 done | fold -sw40

```

```

do:2 let:1 declare:1 bash:1 done:2
bin:1 fold:1 count:4 echo:1 cat:1 sed:1
a:1 in:2 g:1 n:2 s:1 for:2 w:5 z:1 sw:1

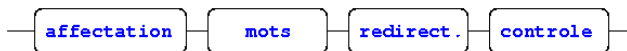
```

types de commandes

Une commande composée d'arguments i.e. `echo a b c` est une *commande simple* à partir desquelles on peut construire des commandes complexes :

- pipelines (tube)
- listes (batch)
- composée (contrôle)
 - Coprocesses Two-way communication between commands.
 - GNU Parallel Running commands in parallel.

commande simple



- En général, la commande est spécifiée par le premier mot, les autres sont les arguments de la commande.
- Un signal affecte la valeur de retour :

$128 + \text{signal}$

`[time [-p]] [!] CMD1 [[| ou |&] CMD2 ...]`

Un *pipeline* est une suite de *commandes simples* séparées par un opérateur de contrôle :

‘ | ’ ou ‘ |& ’.

- La sortie de `CMD1` est connectée par un tube vers l'entrée de `CMD2`.
- Autrement dit, `CMD2` lit dans la sortie de `CMD1`.
- La connexion est réalisée avant l'évaluation des redirections.
- Avec ‘ |& ’, le canal d'erreur de `CMD1` est dirigé vers l'entrée de `CMD2`.

Il s'agit d'un raccourci pour :

`CMD1 2>&1 | CMD2`

Cette redirection implicite est réalisée après l'évaluation des redirections.

exemple de pipeline

```
~> man bash | sed 's/[^A-Z]/\n/g' | \
  grep -E '[A-Z]{3}' | sort | uniq -c | sort -nr
  54 BASH
  42 SHELL
  29 BUILTIN
  27 COMMANDS
  21 PATH
  21 IFS
  18 COMP
  16 POSIX
  13 EXPANSION
  13 ENV
  11 NAME
```

Liste de commandes

Une liste de commandes est un enchaînement de pipelines séparés par un des opérateurs :

`;` `&` `&&` `||`

qui se termine éventuellement par `;` ou `&`, ou `' NL '`.

- Les opérateurs `&&` et `||` ont la même précedence, et sont prioritaires aux deux autres `;` et `&`.
- Une commande qui se termine par `&` est exécutée en arrière plan (exécution asynchrone) dans un sous-shell. Le shell n'attend pas la terminaison de la commande, le code est 0 (true).
- L'entrée standard est par défaut `/dev/null`.
- Les commandes séparés par des `;` sont exécutées séquentiellement. Le shell attend la terminaison de chacune des commandes à tour de rôle.
- La valeur de retour correspond au status de la dernière

liste conditionnelle

Il s'agit des séquences séparées par des '&&' ou bien des '||'.

Dans une conjonction :

$$\text{CMD}_1 \ \&\& \ \text{CMD}_2$$

CMD_2 est exécutée si et seulement si le retour de CMD_1 est OK.

Dans une disjonction :

$$\text{CMD}_1 \ \text{---} \ \text{CMD}_2$$

CMD_2 est exécutée si et seulement si le retour de CMD_1 est KO.

Dans les deux cas, la valeur de retour sera bien celle de la dernière commande exécutée.

exemple

```
1 #!/bin/bash -v
2 help time | sed -nE '/[pP]osix/s/[[[: space:]]+ / /p'
3 -p print the timing summary in the portable Posix format
4 time -p (sleep 1 && sleep 2 && sleep 3)
5 real 6.00
6 user 0.00
7 sys 0.00
8 time -p (sleep 1 || sleep 2 || sleep 3)
9 real 1.00
10 user 0.00
11 sys 0.00
12 time -p (sleep 1 & sleep 2 & sleep 3)
13 real 3.00
14 user 0.00
15 sys 0.00
```

(liste)

exécutée dans un sous-shell sans effet sur l'environnement.

```
1 #!/bin/bash -v
2 x=HELLO
3 echo x=$x s=$BASH_SUBSHELL
4 x=HELLO s=0
5 ( x=${x,,*}; echo x=$x s=$BASH_SUBSHELL; )
6 x=hello s=1
7 echo x=$x s=$BASH_SUBSHELL
8 x=HELLO s=0
```

```
~> ( cd /tmp; pwd ) ; pwd
/tmp
/home/pl
```

{liste ;}

exécutée avec l'environnement du shell en cours !

```
1 #!/bin/bash -v
2 x=hello
3 echo x=$x s=$BASH_SUBSHELL
4 x=hello s=0
5 { x=${x^^*}; echo x=$x s=$BASH_SUBSHELL; }
6 x=HELLO s=0
7 echo x=$x s=$BASH_SUBSHELL
8 x=HELLO s=0
```

```
1 :: { echo hello; echo world } >> message
```

```
if test-cmd ; then CMD0 ;[else CMD1 ;]fi
```

Si le code de retour de `test-cmd` est nul alors `CMD0` est exécutée sinon c'est `CMD1`.

```
1 if [ -n HOST ] ; then
2   if ping -c1 $HOST.euphoria.fr ; then
3     ssh pl@$HOST.euphoria.fr who
4   else
5     echo $HOST is down
6   fi
7 fi
```

Les commandes `[...]` et `[[...]]` sont utilisées pour construire des tests.

- `[]` test simple, détaillé ici.
- `[[]]` voir le manuel.
- `elif` construction est possible.

help test

-d file	répertoire
-f file	fichier régulier
-p file	pipe, fifo
-x file	exécutable
-N file	modifié depuis dernier accès
-o optname	option du shell activée
-v varname	variable assignée
-z string	chaîne vide
-n string	chaîne non vide

test unaire

```
1 #!/bin/bash -v
2 # 0 : SUCCES
3 # 1 : FAILURE
4 unset s
5 [ -z "$s" ]; echo $?
6 0
7 [ -n "$s" ]; echo $?
8 1
9 s=initialisation
10 [ ! -z "$s" ] && echo non vide
11 non vide
12 [ -n "$s" ] && echo init
13 init
```

test binaire

file1 -nt file2	file1 est plus récent que file2
file1 -ot file2	file1 est plus ancien que file2
str1 = str2	égalité des chaînes
str1 != str2	chaînes différentes.
str1 < str2	comparaison lexicographique
str =~ pat	correspondance
arg1 ♣ arg2	comparaison arithmétique
	♣ : '-eq', '-ne', '-lt', '-le', '-gt', or '-ge'.

```

$ [ "x" = "x" ]; echo $?      —> 0
$ [ "x" < "y" ]; echo $?     —> 1
bash: y: Aucun fichier ou dossier de ce type
$ [[ "x" < "y" ]]; echo $?   —> 0
$ [[ azerty =~ a.*y ]]; echo $? —> 0

```

test binaire

```
1 #!/bin/bash -v
2 x='1234'
3 y=' 1234'
4 if [ "$x" = "$y" ] ; then
5     echo egalite
6 else
7     echo difference
8 fi
9 difference
10 if [ $x = $y ] ; then
11     echo egalite
12 else
13     echo difference
14 fi
15 egalite
```


expression logique

! expr	négation de expr
(expr)	
expr1 -a expr2	et logique
expr1 -o expr2	ou logique

expression logique

```
1 #!/bin/bash -v
2 function transfert {
3 ls -lt $2 $1/$2
4 if [ -d $1 -a -f $2 -a "$2" -nt "$1/$2" ] ; then
5     cp -f $2 $1/$2
6 fi
7 }
8 echo hello > /tmp/h
9 sleep 60
10 echo Hello > h
11 transfert /tmp h
12 -rw-rw-r-- 1 pl pl 6 20 juin 08:24 h
13 -rw-rw-r-- 1 pl pl 6 20 juin 08:23 /tmp/h
14 transfert /tmp h
15 -rw-rw-r-- 1 pl pl 6 20 juin 08:24 /tmp/h
```

while test-cmds ; do seq-cmds ; done

Exécute la suite de commandes tant que le code de retour de test-commands vaut zéro. Le code de retour est celui de la dernière commande exécutée.

```
1 while ping -c1 -W1 192.168.0.254; do
2     echo alive
3     sleep 60
4 done
```

- les " ;" optionnel deviennent obligatoires sur une ligne !

L'inévitable factorielle !

```
1 declare -i n=1
2 r=1 ; z=1
3 # comparaison des chaines
4 while [ $r == $z ]
5 do
6     # n est entier
7     n=n+1
8     # r non entier
9     let r=r*n
10    z=$( bc <<< "$z*$n" )
11 done
12 echo n=$n $r $z
```

L'inévitable factorielle !

```
1 declare -i n=1
2 r=1 ; z=1
3 # comparaison des chaines
4 while [ $r == $z ]
5 do
6     # n est entier
7     n=n+1
8     # r non entier
9     let r=r*n
10    z=$( bc <<< "$z*$n" )
11 done
12 echo n=$n $r $z
```

```
1 n=21 -4249290049419214848 51090942171709440000
```

L'inévitable factorielle !

```
1 #!/bin/bash
2 z=$( \
3 bc << -EOJ
4     r=1
5     for ( n=1 ; n < 50; n++ )
6         r=r*n
7     print r
8 -EOJ
9 )
10 echo $z
```

while et read

La commande `read` utilise le premier caractère de la variable `IFS` pour découper les lignes d'un fichier :

```
1 #!/bin/bash
2 IFS=:
3 while read user x uid z
4 do
5     if [ $uid -ge 1000 ]
6     then
7         echo $user
8     fi
9 done < /etc/passwd
```

mais une commande sait faire cela, laquelle ?

while et read

La commande `read` utilise le premier caractère de la variable `IFS` pour découper les lignes d'un fichier :

```
1 #!/bin/bash
2 IFS=:
3 while read user x uid z
4 do
5     if [ $uid -ge 1000 ]
6     then
7         echo $user
8     fi
9 done < /etc/passwd
```

mais une commande sait faire cela, laquelle ?

```
[$] gawk -F: '{ if ($3 >=1000) print $1}' /etc/passwd
```


for name [[in [words...]]];] do cmds; done

- expansion de words
- exécution des commandes

```
1 #!/bin/bash
2 set a b c
3 for x      ; do echo -n $x; done
4 for x in * ; do echo -n $x; done
5 for x in $( cat $0 ); do echo -n $x; done
6 for x in alpha beta gamma ;
7   do
8     echo -n $x;
9   done
```

- pas de in words est spécial.

for ((expr1 ; expr2 ; expr3)) ; do cmds ; done

Avec un comportement similaire à la boucle `for` du langage C : l'expression arithmétique `expr1` est évaluée, puis celle de `expr3` tant que `expr2` n'est pas nulle.

```
1 deb=$1
2 lim=$2
3 pas=1
4 if [ $# = 3 ] then
5     pas=$3
6 fi
7
8 for (( x=deb; x<lim ; x+=pas )) ;
9     do
10         echo $x ;
11     done
```

Chronométrage

```
1 deb=$1 ; lim=$2 ; pas=$3 ; cmd=$4
2 if [ $# != 4 ] ; then
3     echo usage $0 : deb lim pas cmd
4     exit 1
5 fi
6 if [ -f time.log ] ; then
7     echo effacer time.log
8     exit 2
9 fi
10 for (( x=deb; x<lim ; x+=pas )) ; do
11     echo -n .$x
12     /usr/bin/time -a -o time.log --format="$x %e" $cmd $x
13 done
```

■ gnuplot...

case wd in [[(] x [| x]...) cmd-list ;;]...esac

case exécute la liste de commandes correspondant au premier motif correspondant.

- | sépare les motifs
-) termine les motifs

Chaque clause se termine par :

‘;;’ ‘;&’ ‘;;&’.

```
1 echo -n "Enter the name of an animal: "
2 read ANIMAL
3 echo -n "The $ANIMAL has "
4 case $ANIMAL in
5   horse | dog | cat) echo -n "four";;
6   man | kangaroo ) echo -n "two";;
7   *) echo -n "an unknown number of";;
8 esac
```

select name [in words...]; do cmds; done

Une liste d'items est contruite à partir de la suite words. Elle est placée sur la sortie standard. L'utilisateur est invité à faire un choix passé dans la variable `REPLY`.

```
1 mkdir /tmp/dir
2 rm -rf /tmp/dir/*
3 touch /tmp/dir/{a..h}
4 select fname in /tmp/dir/*;
5 do
6     index=$REPLY
7     break;
8 done
9 echo -e "\nselection: $fname ($index)"
10 echo ----
```

```
1) /tmp/dir/a    3) /tmp/dir/c
5) /tmp/dir/e    7) /tmp/dir/g
2) /tmp/dir/b    4) /tmp/dir/d
6) /tmp/dir/f    8) /tmp/dir/h
#?
```

```
selection: /tmp/dir/g (7)
```

```
select.out : select.sh
echo 7 | ./select.sh \
        |& fold -s -w30 > select.out

sed -n '/select.out/,/clean/p' \
        makefile >> select.out

clean:
```

getopts optstring name [args]

La commande `getopts` permet de gérer des options passées par la ligne de commande :

- `optstring` les options à reconnaître, avec argument ou pas.
- `name` option courante.
- `OPTIND` indice de l'argument suivant.
- `OPTARG` argument de l'option.

Traitement des erreurs

- le drapeau `OPTERR` contrôle les messages d'erreurs.
- idem `' :` préfixe `optstring`.
- `name` vaut `' ?'` pour une option inconnue.
- `name` vaut `' :'` pour un argument attendu.

utilisation de getopt

```
1 #!/bin/bash
2 while getopt ":hp:r:" opt; do
3     case $opt in
4         p) parm=$OPTARG;;
5         \?) echo "option invalide $OPTARG"; exit 1 ;;
6         :) echo "argument requis $OPTARG"; exit 2 ;;
7         h) echo "usage $0 [ -p nombre ] [ fichier ]"
8             exit 3 ;;
9     esac
10 done
11 echo $*
12 shift $((OPTIND-1))
13 [[ $parm =~ $reg ]] && echo matching
```