

Programmation Système

Cours 4: Programmation Shell

Khaddouja ZELLAMA

Khaddouja.zellama@dauphine.psl.eu

Licence L2 Mathématiques et Informatique
Département MIDO

Le langage shell

Le shell est le langage des commandes Unix

Il est conçu pour faciliter:

- L'utilisation de programmes externes e.g. ls, wc, head, grep
- La manipulation de fichiers et la redirection des E/S (via les opérateurs '<' et '>')
- Le chaînage de commandes externes (via l'opérateur '|')

Le langage shell

Le shell est le langage des commandes Unix

Il est conçu pour faciliter:

- L'utilisation de programmes externes e.g. ls, wc, head, grep
- La manipulation de fichiers et la redirection des E/S (via les opérateurs '<' et '>')
- Le chaînage de commandes externes (via l'opérateur '|')

C'est aussi un langage de programmation à part entière, avec:

- variables
- fonctions
- structures de contrôle if, while, for etc.

► Bien distinguer les mots clés du langage shell des commandes externes!

Bash

- Le shell est un langage interprété
- Bash est un interpréteur de commandes shell

Bash

- Le shell est un langage interprété
 - Bash est un interpréteur de commandes shell
-
- Lit et analyse les commandes écrites sur son entrée standard
 - Crée les processus pour exécuter les commandes
 - Re-dirige les E/S si nécessaire
 - Donne la main aux processus qu'il a créés
 - Reprend la main lorsque le processus a terminé son exécution

Bash (2)

- Exécuté au démarrage du terminal

Possibilité de changer le shell exécuté par le terminal

```
1 SHELL=/home/bnegreve/../../myshell xterm # exécute xterm avec mon propre shell
```

- S'exécute jusqu'à l'apparition de EOF¹, ou du mot clé exit
- Il a la possibilité de lancer d'autres programmes

Y compris lui même:

```
1 $ ps -H
2   PID TTY          TIME CMD
3  380589 pts/32    00:00:00 bash
4 $ bash
5 $ bash
6 $ ps -H
7   PID TTY          TIME CMD
8  380505 pts/31    00:00:00 bash
9  380619 pts/31    00:00:00  bash
10 380630 pts/31    00:00:00   bash
```


¹Caractère 'End Of File' qui signale la fin d'un fichier, obtenu avec Ctrl+D

Un premier script shell

Possibilité d'écrire des Scripts en shell

- Fichiers contenant des séries de commandes
- Destinés à être réutilisés plusieurs fois

```
1 $ cat script.sh
2 echo "Salut"
3 date
4 ps
5 $ bash < script.sh
6 Salut
7 Mon Feb 24 10:08:54 CET 2020
8      PID TTY          TIME CMD
9  380027 pts/24    00:00:00 bash
10 380329 pts/24    00:00:23 emacs
11 380893 pts/24    00:00:00 bash
12 380894 pts/24    00:00:00 ps
```

 Bash interprète le langage shell + certaines commandes spécifiques.
Pour être sûr qu'un script soit portable: utiliser l'interpréteur sh

Shebang (#!)

- Permet de spécifier le programme à utiliser pour interpréter le fichier
- Possibilité d'exécuter un script comme n'importe quel autre programme

```
1 $ cat script.sh
2 #! /bin/bash
3 echo "Salut"
4 date
5 ps
```

```
1 $ chmod +x script.sh
2 $ ./script.sh
3 Salut
4 Mon Feb 24 10:08:54 CET 2020
5      PID TTY          TIME CMD
6 380027 pts/24  00:00:00 bash
7 ...
```


Shebang (#!)

- Permet de spécifier le programme à utiliser pour interpréter le fichier
- Possibilité d'exécuter un script comme n'importe quel autre programme

```
1 $ cat script.sh
2 #! /bin/bash
3 echo "Salut"
4 date
5 ps
```

```
1 $ chmod +x script.sh
2 $ ./script.sh
3 Salut
4 Mon Feb 24 10:08:54 CET 2020
5     PID TTY          TIME CMD
6  380027 pts/24    00:00:00 bash
7  ...
```

Équivalent à

```
1 $ bash < script.sh
```

Note: marche aussi pour python, gnuplot etc.

Shebang (#!)

- Permet de spécifier le programme à utiliser pour interpréter le fichier
- Possibilité d'exécuter un script comme n'importe quel autre programme

```
1 $ cat script.sh
2 #! /bin/bash
3 echo "Salut"
4 date
5 ps
```

```
1 $ chmod +x script.sh
2 $ ./script.sh
3 Salut
4 Mon Feb 24 10:08:54 CET 2020
5      PID TTY          TIME CMD
6 380027 pts/24  00:00:00 bash
7 ...
```

Équivalent à

```
1 $ bash < script.sh
```

Note: marche aussi pour python, gnuplot etc.

Que faire si aucune commande ne répond à mon besoin?

► Composer avec les commandes existantes!

Composition de commandes

... via la redirection des E/S

(Rappel) Les processus UNIX ont

- une entrée standard (descripteur de fichier 0)
- une sortie standard (descripteur de fichier 1)
- une sortie erreur (descripteur de fichier 2)

Par défaut, tout est redirigé vers le “fichier” associé terminal
(e.g. /dev/pts/X)

Composition de commandes

... via la redirection des E/S

(Rappel) Les processus UNIX ont

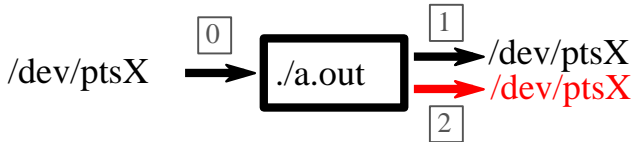
- une entrée standard (descripteur de fichier 0)
- une sortie standard (descripteur de fichier 1)
- une sortie erreur (descripteur de fichier 2)

Par défaut, tout est redirigé vers le “fichier” associé terminal
(e.g. /dev/pts/X)

- `scanf(..)` lit les caractères qui sont tapés dans le terminal
- `printf(..)` ou `dprintf(1, ..)` affiche les caractères dans le terminal
- `dprintf(2, ..)` affiche (aussi) les caractères dans le terminal

stdin, stdout, stderr (2)

```
1 bnegreve@neb:~$ ./a.out
2 PID 373015
3 ...
4 bnegreve@neb:~$ ls -l /proc/373015/fd/
5 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:39 0 -> /dev/pts/41
6 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:39 1 -> /dev/pts/41
7 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:39 2 -> /dev/pts/41
```



- `printf("salut");` écrit "salut" dans le terminal

Redirection de stdout en bash

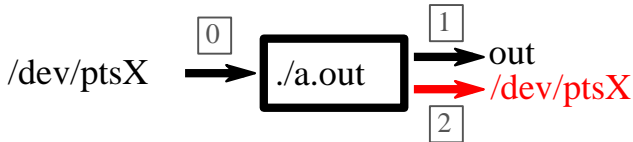
- Possibilité de rediriger stdout avec l'opérateur '>'

```
1 bnegreve@neb:~$ ./a.out > out
2 bnegreve@neb:~$ ps aux | grep a.out
3 bnegreve 373752 0.0 0.0 2304 500 pts/41 S 15:44 0:00 ./a.out
4 bnegreve@neb:~$ ls -l /proc/373752/fd
5 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:44 0 -> /dev/pts/41
6 l-wx----- 1 bnegreve bnegreve 64 Feb 8 15:44 1 -> /home/bnegreve/out
7 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:44 2 -> /dev/pts/41
```

Redirection de stdout en bash

- Possibilité de rediriger stdout avec l'opérateur '>'

```
1 bnegreve@neb:~$ ./a.out > out
2 bnegreve@neb:~$ ps aux | grep a.out
3 bnegreve 373752 0.0 0.0 2304 500 pts/41 S 15:44 0:00 ./a.out
4 bnegreve@neb:~$ ls -l /proc/373752/fd
5 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:44 0 -> /dev/pts/41
6 l-wx----- 1 bnegreve bnegreve 64 Feb 8 15:44 1 -> /home/bnegreve/out
7 lrwx----- 1 bnegreve bnegreve 64 Feb 8 15:44 2 -> /dev/pts/41
```



- `printf("salut");` écrit "salut" dans le fichier `out`

Redirection de stdout/stderr en bash

```
1 $ cat test.c
2 int main(int argc, char *argv[]){
3     dprintf(1, "Début de l exécution\n");
4     dprintf(1, "J écris sur stdout\n");
5     dprintf(2, "J écris sur stderr\n");
6     dprintf(1, "Fin de l exécution\n");
7     return 0;
8 }
9 $
```

```
1 $ gcc test.c
2 $ ./a.out
3 Début de l exécution
4 J écris sur stdout
5 J écris sur stderr
6 Fin de l exécution
7 $
```


Redirection de stdout/stderr en bash

```
1 $ cat test.c
2 int main(int argc, char *argv[]){
3     dprintf(1, "Début de l exécution\n");
4     dprintf(1, "J écris sur stdout\n");
5     dprintf(2, "J écris sur stderr\n");
6     dprintf(1, "Fin de l exécution\n");
7     return 0;
8 }
9 $
```

```
1 $ gcc test.c
2 $ ./a.out
3 Début de l exécution
4 J écris sur stdout
5 J écris sur stderr
6 Fin de l exécution
7 $
```

Rediriger stdout avec '1>' ou '>'

```
1 $ ./a.out > out.txt
2 J écris sur stderr
3 $ cat out.txt # contenu de out.txt:
4 Début de l exécution
5 J écris sur stdout
6 Fin de l exécution
7 $
```

Redirection de stdout/stderr en bash

```
1 $ cat test.c
2 int main(int argc, char *argv[]){
3     dprintf(1, "Début de l exécution\n");
4     dprintf(1, "J écris sur stdout\n");
5     dprintf(2, "J écris sur stderr\n");
6     dprintf(1, "Fin de l exécution\n");
7     return 0;
8 }
9 $
```

```
1 $ gcc test.c
2 $ ./a.out
3 Début de l exécution
4 J écris sur stdout
5 J écris sur stderr
6 Fin de l exécution
7 $
```

Rediriger stdout avec '1>' ou '>'

```
1 $ ./a.out > out.txt
2 J écris sur stderr
3 $ cat out.txt # contenu de out.txt:
4 Début de l exécution
5 J écris sur stdout
6 Fin de l exécution
7 $
```

Rediriger stderr avec '2>'

```
1 $ ./a.out 2> err.txt
2 Début de l exécution
3 J écris sur stdout
4 Fin de l exécution
5 $ cat err.txt # err.txt
6 J écris sur stderr
7 $
```

Redirection de stdin en bash

```
1 $ cat test.c
2 int main(int argc, char *argv[]){
3     char name[16];
4     printf("Quel est ton nom?\n");
5     scanf("%s", &name);
6     printf("Salut %s\n", name);
7     return 0;
8 }
9 $
```

```
1 $ gcc test.c
2 $ ./a.out
3 Quel est ton nom?
4 ben
5 Salut ben
6 $
```

► Lit "ben" depuis le terminal

Redirection de stdin en bash

```
1 $ cat test.c
2 int main(int argc, char *argv[]){
3     char name[16];
4     printf("Quel est ton nom?\n");
5     scanf("%s", &name);
6     printf("Salut %s\n", name);
7     return 0;
8 }
9 $
```

```
1 $ gcc test.c
2 $ ./a.out
3 Quel est ton nom?
4 ben
5 Salut ben
6 $
```

► Lit "ben" depuis le terminal

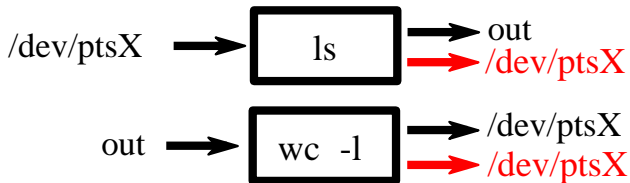
```
1 $ cat in.txt
2 ben
3 $
```

```
1 $ ./a.out < in.txt
2 Salut ben
3 $
```

► Lit "ben" depuis le fichier in.txt

Composition de commandes avec < >

```
1 $ ls > out
2 $ wc -l < out
3 35
4 $
```

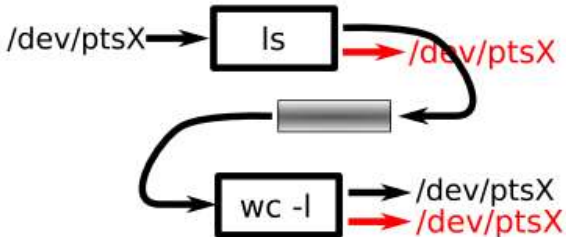


Composition de commandes avec |

```
1 $ ls | wc -l
2 35
3 $
```

► l'opérateur **pipe** (|) crée un fichier temporaire

- Réside uniquement en mémoire, de taille fixe
- Le stdout du processus de gauche est connectée a l'entrée du pipe.
- Le stdin du processus de droite est connectée a la sortie du pipe



Variables en bash

Affectation:

```
1 name=satya
```

Attention: pas d'espace entre le nom de la variable et '='

Utilisation

```
1 $ echo $name
2 satya
3 $ echo $name2
4 (rien)
5 $ echo ${name}2
6 satya2
```

Accolade pour delimitier l'identifiant de la variable si nécessaire

Portée des variables

```
1 $ name=ben
2 $ echo "Salut $name"!
3 Salut ben!
```

```
1 cat salut.sh
2 #! /bin/bash
3 echo Salut $name!
```

```
1 $ ./salut.sh
2 Salut ! # Ou est le nom ???
```


Portée des variables

```
1 $ name=ben
2 $ echo "Salut $name"!
3 Salut ben!
```

```
1 cat salut.sh
2 #! /bin/bash
3 echo Salut $name!
```

```
1 $ ./salut.sh
2 Salut ! # Ou est le nom ???
```

► Le script salut.sh ne s'exécute pas dans le même bash!

```
1 $ cat test.sh
2 #! /bin/bash
3 ps -H
```

```
1 bnegreve@neb:~$ ps -H
2      PID TTY          TIME CMD
3      16311 pts/11    00:00:00 bash
4      16416 pts/11    00:00:00  bash
5      16417 pts/11    00:00:00   ps
```

Portée des variables

```
1 $ name=ben
2 $ echo "Salut $name"!
3 Salut ben!
```

```
1 cat salut.sh
2 #! /bin/bash
3 echo Salut $name!
```

```
1 $ ./salut.sh
2 Salut ! # Ou est le nom ???
```

► Le script salut.sh ne s'exécute pas dans le même bash!

```
1 $ cat test.sh
2 #! /bin/bash
3 ps -H
```

```
1 bnegreve@neb:~$ ps -H
2      PID TTY          TIME CMD
3    16311 pts/11    00:00:00 bash
4    16416 pts/11    00:00:00  bash
5    16417 pts/11    00:00:00   ps
```

Les variables définies dans un bash ne sont plus disponibles dans les processus fils.

Mot clé export

Le mot clé export permet de propager la définition des variables dans les processus fils.

Sans export

```
1 $ name=ben
2 $ bash
3 $ ps -H # affiche la list des processus sous forme d'arbre
4   PID TTY          TIME CMD
5   24050 pts/3    00:00:00 bash
6   25411 pts/3    00:00:00  bash
7 $ echo $name # n'affiche rien, name n'est pas définie dans le processus fils
```

Avec export

```
1 $ export name=ben
2 $ bash
3 $ echo $name # affiche ben
4 ben
5 $ ./salut.sh
6 Salut ben! # ah :)
```

Passage de valeurs (1)

1^{ère} méthode: en utilisant les variables arguments

- \$0 : nom du script
- \$1 : premier argument
- \${N} : N-ième argument
- \$# : nombre d'arguments

print-name-2.sh:

```
1 #!/bin/bash
2
3 if [ $# -eq 1 ]; then
4     echo Votre nom est : $1.
5 else
6     echo Votre nom est : Anonyme.
7 fi
```

```
1 $ ./affiche-nom-2.sh ben
2 Votre nom est : ben.
3 $ ./affiche-nom-2.sh
4 Votre nom est : Anonyme.
```

Passage de valeurs (2)

2^{ème} méthode: par substitution

affiche-nom.sh:

```
1 #!/bin/bash
2
3 ${nom:=Anonyme}
4 echo Votre nom est : $nom.
```

```
1 $ ./affiche-nom.sh
2 Votre nom est : Anonyme.
3 $ export nom=ben
4 $ ./affiche-nom.sh
5 Votre nom est : ben.
```

Attention: sans le export, ça ne marche pas.

Substitution en bash

`${<varname>:-<default-value>}`

```
1 $ echo bonjour ${name:-anonyme}.  
2 Bonjour anonyme.  
3 name=ben  
4 $ echo ${name:-anonyme}.  
5 bonjour ben.
```

`${<varname>:=<default-value>}`

```
1 $ echo bonjour ${name:=anonyme}.  
2 bonjour anonyme.  
3 echo $name  
4 anonyme
```

`${<varname>:?<err-msg>}`

```
1 $ echo bonjour ${name:?variable inconnue}.  
2 bash: name: variable inconnue.
```

Note: interrompt l'exécution du script en cours.

Passage de valeurs (3)

3ème méthode: en utilisant la commande read

`read <var>` : Lit une ligne sur l'entrée standard, et affecte le resultat à la variable `var`.

`affiche-nom-3.sh`

```
1 #!/bin/bash
2 read nom # en attente d'une ligne
3 $ echo Votre nom est : $nom.
```

```
1 $ ./affiche-nom-3.sh
2 ben
3 Votre nom est : ben.
4 $ cat mon-nom.txt
5 ben
6 $ ./affiche-nom-3.sh < mon-nom.txt
7 Votre nom est : ben.
```

Variables spéciales

- `$USER`: nom de l'utilisateur en cours
- `$HOME`: chemin vers le répertoire *home* de l'utilisateur
- `$HOSTNAME`: nom de la machine en cours d'utilisation
- `$PWD`: répertoire courant
- `$RANDOM`: nombre aléatoire
- `$PATH`: ensemble de répertoires où se trouvent les exécutables

Variable \$PATH

Invocation d'un programme se fait de deux manières:

- En tapant le nom du binaire si la commande est dans un des répertoire de \$PATH
- En tapant un chemin dans le système de fichier si la commande n'est pas dans \$PATH

(e.g. /usr/bin/ls unix/TP/exo1 ./exo1)

```
1 $ echo $PATH
2 /usr/local/bin:/usr/bin:/bin
3 # liste de repertoires contenant des executable séparés par ':'
4 $ which ls # où se trouve la commande 'ls'
5 /bin/ls
6 $ PATH="" # a ne pas faire
7 $ ls
8 bash: ls: No such file or directory
```

Variable \$?

La variable \$? contient la valeur de retour du dernier programme qui s'est exécuté

Sémantique:

- 0 = Exécution réussie
- non-zero = Erreur

```
1 $ ls qsdqsd
2 ls: cannot access 'qsdqsd': No such file or directory
3 $ echo $?
4 2
```

```
1 $ ls test.sh
2 test.sh
3 $ echo $?
4 0
```

Variable \$?

La variable \$? contient la valeur de retour du dernier programme qui s'est exécuté

Sémantique:

- 0 = Exécution réussie
- non-zero = Erreur

```
1 $ ls qsdqsd
2 ls: cannot access 'qsdqsd': No such file or directory
3 $ echo $?
4 2
```

```
1 $ ls test.sh
2 test.sh
3 $ echo $?
4 0
```

Pourquoi?

► Permet de communiquer un message d'erreur en cas d'echec

Structure de contrôle if-then-else

```
1 if <commande-1>
2 then
3   <commande a executer si commande-1 a reussit>
4 else
5   <commande a executer sinon>
6 fi
```

Structure de contrôle if-then-else

```
1 if <commande-1>
2 then
3     <commande a executer si commande-1 a reussit>
4 else
5     <commande a executer sinon>
6 fi
```

trouver-mot.sh

```
1  #!/bin/bash
2  mot="poulet"
3  fichier="lesmiserables.txt"
4
5  if grep $mot $fichier > /dev/null
6  then
7      echo "Le mot $mot à été trouvé dans $fichier."
8  else
9      echo "Le mot $mot n'a pas été trouvé dans $fichier."
10 fi
```

Opération booléennes

```
1  if commande-1 && commande-2; then echo "ok"; else echo "ko"; fi
2  if commande-1 || commande-2; then echo "ok"; else echo "ko"; fi
```

Exemple

```
1  if grep mot1 fichier && grep mot2 fichier; then
2      echo $mot1 et $mot2 trouvés
3  fi
```

Opération booléennes

```
1 if commande-1 && commande-2; then echo "ok"; else echo "ko"; fi
2 if commande-1 || commande-2; then echo "ok"; else echo "ko"; fi
```

Exemple

```
1 if grep mot1 fichier && grep mot2 fichier; then
2     echo $mot1 et $mot2 trouvés
3 fi
```

```
1 $ grep '\bpoule\b' lesmiserables.txt && grep '\bvache\b' lesmiserables.txt
2 ...
3 $ echo $?
4 0 # <- reussit
```

```
1 $ grep '\bsmartphone\b' lesmiserables.txt && grep '\bvache\b' lesmiserables.txt
2 $ echo $?
3 1 # <- échoue
```

```
1 $ grep '\bsmartphone\b' lesmiserables.txt || grep '\bvache\b' lesmiserables.txt
2 ...
3 0 # <- reussit
```

&& et ||

&& Exécute la première commande, ET la deuxième si la première réussit

```
1 $ ls test.sh && echo "fichier trouvé"
2 test.sh
3 fichier trouvé
```

► La commande "echo" est exécutée puisque 'ls' s'est terminée sans erreur

```
1 $ ls qsdqsd && echo "fichier trouvé"
2 ls: cannot access 'qsdqsd': No such file or directory
```

► La commande "echo" ne s'est pas exécutée puisque ls a retourné une valeur non-nulle

&& et ||

&& Exécute la première commande, ET la deuxième si la première réussit

```
1 $ ls test.sh && echo "fichier trouvé"
2 test.sh
3 fichier trouvé
```

► La commande "echo" est exécutée puisque 'ls' s'est terminée sans erreur

```
1 $ ls qsdqsd && echo "fichier trouvé"
2 ls: cannot access 'qsdqsd': No such file or directory
```

► La commande "echo" ne s'est pas exécutée puisque ls a retourné une valeur non-nulle

|| Exécute la première commande, OU la deuxième si la première échoue

```
1 $ ls qslkdjqs || echo fichier introuvable
2 ls: cannot access 'qsdqsd': No such file or directory
3 fichier introuvable
```

La commande test

```
1 $ s=salut
2 $ test $s = "salut"
3 echo $?
4 0
```

► réussit lorsque le test est vrai, échoue sinon.

```
1 $ a=2
2 $ test $a -eq 2 && echo "a est bien égal à2"
3 a est bien égal à2
4 $ test $a -gt 2 && echo "a est supérieur ou ou égal à2"
5 a est supérieur ou ou égal à2
6 $ test $a -lt 2 || echo "a n'est pas plus petit que 2"
7 a n'est pas plus petit que 2
```

Equivalent

```
1 [ $a -eq 2 ] && echo "a est bien égal à2"
2 a est bien égal à2
```

Tester des valeurs de variables

Avec la structure if-then-else, et le programme test

```
1 if test $# -eq 1
2 then
3     echo "ok"
4 else
5     echo "ko"
6 fi
```

```
1 if [ $# -eq 1 ]
2 then
3     echo "ok"
4 else
5     echo "ko"
6 fi
```

► Affiche "ok" si la commande test réussit, "ko" sinon.

Exemple utile:

```
1 if [ $# -lt 2 ] || [ $# -gt 4 ]; then
2     echo "Erreur: nombre d'arguments invalide."
3 fi
```

Boucle while-done

```
1 while <commande1>
2 do
3   <commande2>
4   ...
5 done
```

Exemple:

```
1 num=$(echo $RANDOM % 100 | bc)
2 while read line
3   if [ $line -gt $num ]; then echo "plus petit"; fi
4   if [ $line -lt $num ]; then echo "plus grand"; fi
5   if [ $line -eq $num ]; then echo "bravo!"; exit 0; fi
6 done
```

Plus-petit-plus-grand

(en utilisant elif)

```
1 #!/bin/bash
2
3 # generate un nombre aléatoire entre 0 et 100
4 num=$(echo $RANDOM % 100 | bc)
5
6 while read line; do
7     if [ $line -gt $num ]
8     then
9         echo "plus petit"
10        elif [ $line -lt $num ]
11        then
12            echo "plus grand"
13        else
14            echo "bravo!"
15        exit 0
16    fi
17 done
```

Boucle for-done

```
1  for <var> in <liste valeurs>
2  do
3  <commande1>
4  done
```

mon-ls.sh

```
1  #!/bin/bash
2
3  for f in *
4  do
5  echo "fichier: $f"
6  done
```

renommer-date.sh

```
1  #!/bin/bash
2
3  date=$(date %F)
4  for i in $(ls *.pdf)
5  do
6    mv $i $date_$i
7  done
```

Boucle numérique

```
1 for i in {1..10}; do
2 echo $i
3 done
```

```
1 for i in $(seq 1 10); do
2 echo $i
3 done
```

nl.sh

```
1 #!/bin/bash
2
3 nlines=$(wc -l $1 | cut -d ' ' -f 1)
4
5 for i in $(seq $nlines); do
6     echo -n "$i: "
7     head -n $i $1 | tail -n 1
8 done
```

Substitution de commandes

`$(< commande >)`

► Exécute `< commande >`, et substitue `$(...)` par l'affichage produit par la commande

```
1 $ nlines=$(wc -l fichier.txt | cut -d ' ' -f 1)
2 $ echo $nlines
3 10
```

```
1 $ a=2
2 $ b=3
3 $ r=$(echo $a + $b | bc)
4 $ echo $r
5 5
```


Exécution séquentielle/parallèle

Exécution séquentielle

```
1 commande1  
2 commande2
```

► Attends la fin de commande1 pour exécuter commande2

```
1 commande1 &  
2 commande2
```

► Exécute commande1 et commande2 sans attendre la fin de commande1

Exécution séquentielle/parallèle

Exécution séquentielle

```
1 commande1  
2 commande2
```

► Attends la fin de commande1 pour exécuter commande2

```
1 commande1 &  
2 commande2
```

► Exécute commande1 et commande2 sans attendre la fin de commande1

affiche-pid.sh

```
1 $!/bin/bash  
2 while true  
3 do  
4     echo "Je suis le processus $!"  
5     sleep 1  
6 done
```

```
1 $ ./affiche-pid.sh &  
2 $ ./affiche-pid.sh  
3 Je suis le processus 89401  
4 Je suis le processus 89439  
5 Je suis le processus 89401  
6 Je suis le processus 89439  
7 ...
```