Programmation Shell -- Ecriture de fichiers de commandes

christophe.cerin@iutv.univ-paris13.fr

(version au 18/02/2011)

Introduction. La séquence de boot

BIOS	EFI
• POST • Read bootable media • Load Master Boot Record • Execute MBR	• POST • Read bootable media • Load the GPT table • Mount the EFI system-partition • Run EFI-specific code

GRUB(v1)	GRUB(v2)	(E)LILO
• Stage 1 loaded into MBR/EFI and gets executed by BIOS/EFI • Stage 1.5 loaded by Stage 1, including critical drivers • Stage 2, in the boot filesystem, executes • Stage 2 loads the kernel	• Stage 1 loaded into MBR/EFI and gets executed by BIOS/EFI • Load first sector of core.img • Continues loading core.img • Loads GRUB config • Loads the kernel	• Stage1 loaded into MBR (or EFI by ELILO) and executed by BIOS/EFI • Stage2 is leaded by Stage 1, executes • Loads LILO information. • Loads the kernel

Kernel Load
• The kernel uncompresses into memory • If configured, the kernel mounts the Initial Ramdisk, which contains needed modules to load the rest of the OS • Mounts the root filesystem, loading any needed modules from initrd • Swaps / from initrd to the actual root filesystem • Executes the specified init process

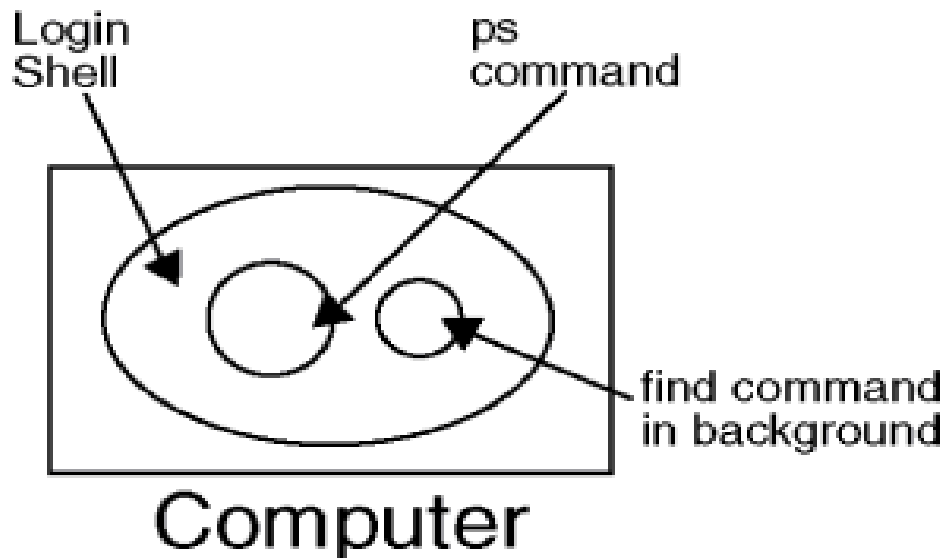
Initd	Systemd	Upstart	Launchd

• Is launched by the kernel as PID 1
• Checks /etc/inittab for loading procedures
• Runs scripts specified by inittab
  ◦ Mounts needed filesystems
  ◦ Loads needed modules
  ◦ Starts needed services based on runlevel
  ◦ Finishes setting up userspace
	• Is launched by the kernel as PID 1
	• Reads /etc/system.conf
	• Mounts needed filesystems
	• Loads needed modules
	• Starts services as needed
		• Is launched by the kernel as PID 1
		• Runs startup events listed in /etc/events.d based on runlevel.
		• Loads needed modules
		• Mounts needed filesystems
		• Starts needed services
			• Is launched by the kernel as PID 1
			• Reads /etc/launchd.conf for config details
			• Reads /etc/launchd.plist for per-driver/service details

- On passe la main à des scripts shell situés dans /etc afin de lancer différents services ;
- L'idée : activer les services les uns après les autres (car un service peut avoir besoin d'un autre service pour se lancer).
- Rappels : BIOS (Basic Input Output System : fonctions de base stockées dans la ROM de la carte mère. Ex: lecture s'un secteur sur le disque) -- EFI (Extensible Firmware Interface : appelé à remplacer le BIOS, il possède des fonctions pour gérer la carte réseau, l'installation de plusieurs OS, une interface graphique...). Voir aussi : http://www.intel.com/technology/efi/

Introduction

Connexion à un ordinateur : passer la main à un interpréteur de commandes.



- Les commandes unix vues jusqu'à ce jour vont être insérées dans des fichiers que l'on va exécuter ;
- L'idée : régler un problème en le traitant par une succession de commandes « élémentaires » : c'est de la programmation !

- La nature des commandes insérées dans le fichier : a) des instructions de l'interpréteur de commande b) et/ou des commandes de /usr/bin (cf prochain chapitre) ou accessibles depuis PATH
- Rq : il y a beaucoup de shell (sh, bash, csh, tcsh, zsh, ksh). On étudie bash.

- Un interpréteur de commandes (shell) est une interface entre le niveau utilisateur et les ressources (disks, mem, network)
- Pour systématiser les traitements, tous les shell permettent de faire de la mise en séquence d'actions, du choix, de l'itération : ce sont des langages de programmation (rudimentaires par rapport a C++, Java, C#)

Eléments introductifs

- Un fichier shell s'édite avec n'importe quel éditeur : c'est un fichier texte !
- On peut l'exécuter après un chmod +x ou par `source mon_fichier_shell param1 param2` ou par : `/bin/bash mon_fichier_shell param1 param2`

4 2 1 4 2 1 4 2 1
drwxrwxr x
Directory Indicator
Owner
Group
Ohter

- Dans mon_fichier_shell, $0 donne le nom du fichier shell, $1 donne le 1er paramètre etc
- $# donne le nb de paramètres du fichier shell
- $* est la liste (sous format de chaine) des paramètres, séparés par un blanc
- $$: numéro du processus shell correspondant à la commande

Fichier titi.sh:

```
#!/bin/bash
# remarquez aussi la place des "
echo "Nombre de parametres :" $#
echo "premier parametre : $1"
```

Execution:

```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/bash titi.sh
Nombre de parametres : 0
premier parametre :
```

Execution:

```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/bash titi.sh AAA
Nombre de parametres : 1
premier parametre : AAA
```

Remarque

- L’exemple précédent ressemble à ce que l’on peut faire avec argv, argc dans un programme C

```
main(int argc, char *argv[]){

  printf("Nb parametre : %d\n",argc);
  printf("Premier parametre : %s\n",argv[1]);

}
```

```
Ordinateur-de-Christophe-Cerin:~ cerin$ gcc titi.c
Ordinateur-de-Christophe-Cerin:~ cerin$ ./a.out AAA
Nb parametre : 2
Premier parametre : AAA
```

Variables & expressions arithmétiques

- Notation 1 : entre ((et))
- Notation 2 : entre [et]

```
> var1=3
> var2=5
> echo $(($var1 + $var2 * 2))
> 13
> echo $[$var1 + $var2 * 2]
> 13
> echo [$var1 + $var2 * 2]
> [3 + 5 #toto.sh# 164expertsGT03.doc 2]
```

Variables & expressions arithmétiques

- Notation 3 : entre { }

> ${VARIABLE} : retourne le contenu de la variable. Les {} : délimiteurs

${VARIABLE:-DEFAULT} : retourne la valeur par défaut si la variable n'est pas initialisée sinon la valeur de la variable.

${VARIABLE:=DEFAULT} : retourne la valeur par défaut si la variable n'est pas initialisée et l'initialise avec DEFAULT sinon retourne la valeur de la variable.

${VARIABLE:+VALUE} : retourne VALUE si la variable est positionnée sinon une chaine vide est retournée.

${#VARIABLE} : retourne la longueur de la valeur de la variable sauf si VARIABLE est * ou @. Dans ces cas, on retourne $@ (liste des paramètres du script).

${VARIABLE:?MESSAGE} : retoune MESSAGE si VARIABLE n'a pas de valeur

Variables

```
port-cerin:~ christophecerin$ Myvar=123
port-cerin:~ christophecerin$ echo $Myvar
123
port-cerin:~ christophecerin$ echo ${Myvar}
123
port-cerin:~ christophecerin$ echo ${Myvar1:-321}
321
port-cerin:~ christophecerin$ echo ${Myvar:-321}
123
port-cerin:~ christophecerin$ echo ${Myvar1:=321}
321
port-cerin:~ christophecerin$ echo ${Myvar1}
321
port-cerin:~ christophecerin$ echo ${Myvar:+11111}
11111
port-cerin:~ christophecerin$ echo ${Myvar2:+11111}

port-cerin:~ christophecerin$ echo ${Myvar2:?"Bonjour"}
-bash: Myvar2: Bonjour
```

Opérateur && et || en shell

- Dans **`liste_commandes1 && liste_commandes2`** l'opérateur **`&&`** est un séparateur de commandes provoquant l'exécution de la liste de commandes **`liste_commandes2`** si la liste de commandes **`liste_commandes1`** renvoie un code de retour ayant la valeur **`0`** (qui équivaut à **`VRAI`** en shell).
- Dans **`liste_commandes1 || liste_commandes2`** l'opérateur **`||`** est un séparateur de commandes provoquant l'exécution de la liste de commandes **`liste_commandes2`** si la liste de commandes **`liste_commandes1`** renvoie un code de retour ayant une valeur différente de **`0`** (qui équivaut à **`FAUX`** en shell).
- Remarque : Les commandes UNIX retournent un code nul lorsque tout s'est bien passé, et des valeurs non nulles dans les différents cas d'erreurs. De façon similaire, dans un programme C, le code de retour est égal à **`0`** lorsque tout s'est bien passé, et à des valeurs non nulles dans les différents cas d'erreurs. En shell, la convention concernant les valeurs **`VRAI`** et **`FAUX`** est l'inverse de celle du langage C, c'est-à-dire que **`VRAI`** correspond à **`0`**, alors que **`FAUX`** correspond à n'importe quelle valeur non nulle.

La commande test (c'est une builtin command de bash !)

- Permet de comparer des « objets », retourne un code mais pas sur la sortie standard
- C'est un if then else un peu particulier

La commande utilitaire test

Syntaxes et description :

- **test comparaison** OU **[comparaison]**

(il faut vraiment taper les crochets et respecter les espaces, dans ce dernier cas). La comparaison **comparaison** peut être la combinaison booléenne de plusieurs comparaisons élémentaires : **comparaison1 -a comparaison2** pour le **ET** logique. **comparaison1 -o comparaison2** pour le **OU** logique. **!comparaison3** pour le **NON** logique.

Il faut bien prendre garde à ne pas confondre **-a** avec **&&**, ni **-o** avec ||

La commande utilitaire test

Des parenthèses précédées de caractères `\` (pour protéger les parenthèses, qui sont des méta-caractères du shell) peuvent être nécessaires en cas de combinaisons un peu plus compliquées, pour préciser les priorités, comme dans l'exemple suivant :

```
! \( comparaison1 -o comparaison2 \)
```

Chaque comparaison élémentaire doit être écrite avec la syntaxe suivante : **`expr1 comparateur expr2`** où **`expr1`** et **`expr2`** désignent des "expressions" du shell courant

La commande utilitaire test

Le comparateur **comparateur** ne s'écrit pas comme en C : si les valeurs de **expr1** et **expr2** sont des chaînes de caractères constituées de chiffres : **-eq** désigne l'égalité (abréviation de **eq**ual). **-ne** désigne la non égalité (abréviation de **n**ot **e**qual). **-gt** signifie "strictement supérieur" (abréviation de **g**reater **t**han). **-ge** signifie "supérieur ou égal" (abréviation de **g**reater or **e**qual). **-lt** signifie "strictement inférieur" (abréviation de **l**ess **t**han). **-le** signifie "inférieur ou égal" (abréviation de **l**ess or **e**qual). Si les valeurs de **expr1** et **expr2** sont des chaînes de caractères quelconques : **=** désigne l'égalité. **!=** désigne la non égalité.

La commande utilitaire test

Attention : Il faut bien prendre garde à ne pas confondre les comparateurs **`=`** et **`-eq`**. Alors que **`test 2 = 02`** retourne **`FAUX`**, que **`test 2 -eq 02`** retourne **`VRAI`** et que **`test baba = bobo`** retourne **`FAUX`**, ce qui semble tout à fait cohérent, on sera surpris de constater que **`test baba -eq bobo`** retourne **`VRAI`** ! Il ne faut donc pas utiliser le comparateur **`-eq`** pour comparer des chaînes de caractères

- **`test $chaine`** Dans cette écriture, la commande **`test`** teste si la variable **`chaine`** a comme valeur la chaîne vide. Si ce n'est pas le cas, la commande **`test`** retourne **`0`** (c'est-à-dire **`VRAI`**).

La commande utilitaire test

Exemples :

- **`test $1 -gt 0 -a $# -eq 2`** Cette commande teste si le premier paramètre positionnel a une valeur entière strictement supérieure à **`0`**, et si le nombre de paramètres positionnels du shell courant ayant reçu une valeur est égal à **`2`**
- **`test -d /users/linfg/linfg0/TD`**Cette commande teste si **`/users/linfg/linfg0/TD`** est le nom d'un répertoire.
- **`test -f nom -a -r nom`** Cette commande teste si **`nom`** est le nom d'un fichier accessible en lecture. Elle est équivalente à : **`test -f nom && test -r nom`**

Usages type de la commande test

```
$ test \( 3 -eq 3 -a 2 -gt 1 \) && echo "oui"
oui
$ test \( 4 -eq 3 -a 2 -gt 1 \) || echo "oui"
oui

$ test \( 4 -eq 3 -a 2 -gt 1 \) || ( echo -n "oui";echo "oui")
ouioui
```

Remarquez le groupement de plusieurs commandes dans la partie droite du dernier exemple.

```
$ while [`test 3 -lt 4` ]; do echo "okay"; done
$
```

ATTENTION : test ne retourne pas un objet Booléen mais il positionne $?

```
$ test 3 -lt 4;jj=$?
$ while [ $jj ]; do echo "Okay"; test 3 -lt 4;   jj
=$?; done
Okay
Okay
Okay
Okay
Okay
```

Supplément

- Le paramètre **`$?`** a pour valeur le code de retour de la dernière commande exécutée dans le shell.

```
#!/bin/bash
ls -lAFd Documents quux
test -a Documents ; echo $?
test -a quux      ; echo $?
test ! -a quux    ; echo $?
```

Exemple difficile : beaucoup d'options

```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/bash toto.sh
ls: quux: No such file or directory
drwx------  51 cerin  cerin  1734  3 Oct 01:12 Documents/
0
1
0
```

On contrôle l'exécution

La commande if-then-else

Syntaxe :

```
if condition1
then liste_commandes1
elif condition2
then liste_commandes2
else liste_commandes3
fi
```

- Les conditions `condition1` et `condition2` doivent être des commandes. La commande la plus souvent utilisée est la commande **`test`**. C'est la valeur du code de retour de cette commande qui est utilisée comme booléen (avec la convention inverse de celle du langage C).

La commande if-then-else

```
#!/bin/bash
if [ $# -eq 1 ]
then
   echo "Le répertoire $1 a le contenu suivant : `ls $1` "
else
   echo 'Mauvais nombre de paramètres'
fi
```

Attention à la présentation : if, then, else, fi sont sur des lignes différentes. C’est important !

Choisissez une bonne fois pour toute votre style. Pas de mélange ! C'est une question de forme.

Bash Conditional Expressions

- On peut aussi construire des expressions booléennes au moyen de la construction [expression]
- expression est (en gros) un terme de la forme 'flag var' comme dans :

```
#!/bin/bash
if [ -z "$PS1" ]; then
        echo This shell is not interactive
else
        echo This shell is interactive
fi
```

Bash Conditional Expressions

- `-a` *`file`* True if file exists.
- `-b` *`file`* True if file exists and is a block special file.
- `-c` *`file`* True if file exists and is a character special file.
- `-d` *`file`* True if file exists and is a directory.
- `-e` *`file`* True if file exists.
- `-f` *`file`* True if file exists and is a regular file.
- `-g` *`file`* True if file exists and its set-group-id bit is set.
- `-h` *`file`* True if file exists and is a symbolic link.
- `-k` *`file`* True if file exists and its "sticky" bit is set.

Expressions booléennes (suite)

`-p` *file* True if file exists and is a named pipe (FIFO).
`-r` *file* True if file exists and is readable.
`-s` *file* True if file exists and has a size greater than zero.
`-t` *fd* True if file descriptor fd is open and refers to a terminal.
`-u` *file* True if file exists and its set-user-id bit is set.
`-w` *file* True if file exists and is writable.
`-x` *file* True if file exists and is executable.
`-O` *file* True if file exists and is owned by the effective user id.
`-G` *file* True if file exists and is owned by the effective group id.
`-L` *file* True if file exists and is a symbolic link.

Expressions booléennes (suite)

`-S` *`file`* True if file exists and is a socket.

`-N` *`file`* True if file exists and has been modified since it was last read.

`file1` `-nt` *`file2`* True if file1 is newer (according to modification date) than file2, or if file1 exists and file2 does not.

`file1` `-ot` *`file2`* True if file1 is older than file2, or if file2 exists and file1 does not.

`file1` `-ef` *`file2`* True if file1 and file2 refer to the same device and inode numbers.

`-o` *`optname`* True if shell option optname is enabled. The list of options appears in the description of the `` `-o' `` option to the `set` builtin.

`-z` *`string`* True if the length of string is zero.

Expressions bolléennes

-n **string**

string True if the length of **string** is non-zero.

string1 == **string2** True if the strings are equal.
`=' may be used in place of `==' for strict POSIX compliance.

string1 != **string2** True if the strings are not equal.

string1 < **string2** True if **string1** sorts before **string2** lexicographically in the current locale.

string1 > **string2** True if **string1** sorts after **string2** lexicographically in the current locale.

arg1 OP **arg2** OP is one of `-eq', `-ne', `-lt', `-le', `-gt', or `-ge'. These arithmetic binary operators return true if **arg1** is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to **arg2**, respectively. **Arg1** and **arg2** may be positive or negative integers.

Le choix avec case

- Similaire au switch de C…mais plus élaboré
- Faire un man bash pour la syntaxe complète

```
#!/bin/bash
case $1 in (titi | toto) echo "found titi or toto";;
           (tata | tutu) echo "found tata or tutu";;
esac
```

```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/sh toto.sh tutu
found tata or tutu
```

Le choix avec case

```
#!/bin/bash
j=`ls -i toto.sh | cut -b 1,2`; echo $j
case `ls -i toto.sh | cut -b 1,2`
    in (16) echo "found first 2 car of inode";;
esac
```

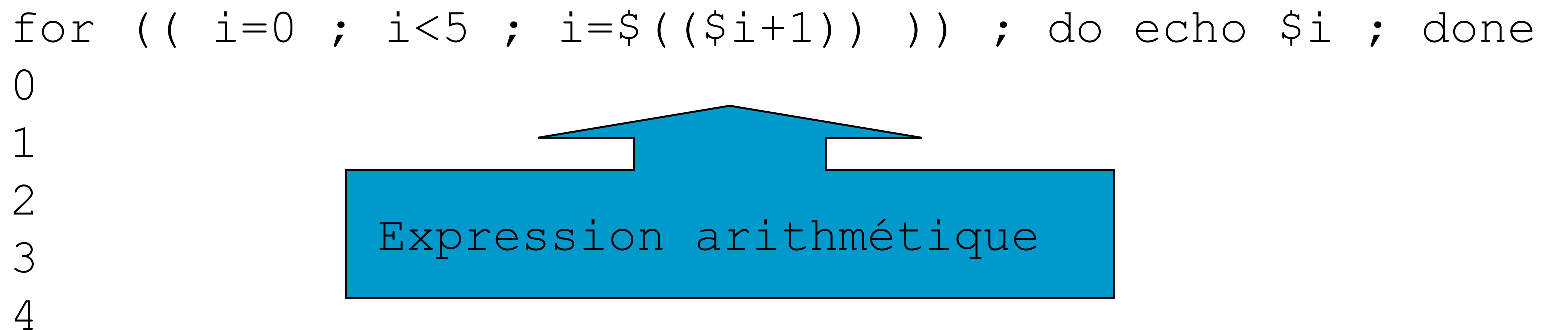
```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/sh toto.sh
16
found first 2 car of inode
```

L’itération avec for

- Première interprétation : pour tous les objets dans la liste faire un traitement
- C’est un équivalent de l’instruction foreach que l’on trouve dans des langages comme Perl et Python par ex
- Faire un man bash pour tous les détails syntaxiques
- `for i in `ls -al`; do echo $i; done`

L’itération avec for

- Deuxième interprétation : celle du for du langage C
- C’est à dire que l’usage est « pour i de 1 a N faire »



L'itération avec while ou until (1)

```
i=0;while (( $i<5 )); do echo $i; i=$(( $i + 1 )); done
0
1
2
3
4
```

Arithmétique sur les entiers

```
i=0;while (( "$i" < "5" )); do echo $i; i=$(( $i + 1 )); done
0
1
2
3
4
```

Comparaison sur les chaînes

$i est une chaîne ou un entier ?

```
i=0;until [ $i -gt 4 ]; do echo $i; i=$(( $i + 1 )); done
0
1
2
3
4
```

Bonne programmation homogène : à la fois la conditionnelle et l'incrémentation sont exprimées avec de l'arithmétique sur les entiers.

Il n'y a pas de mélange des genres !

L'itération avec while ou until (2)

```
while list; do list; done
until list; do list; done
```

The while command continuously executes the do list as long as the last command in list returns an exit status of zero. The until command is identical to the while command, except that the test is negated; the do list is executed as long as the last command in list returns a non-zero exit status. The exit status of the while and until commands is the exit status of the last do list command executed, or zero if none was executed.

```
$ while `ls toto.sh`; do echo 12 ; done
```

Ça boucle !

Pour ne pas boucler il faut que

La re-execution de toto.sh conduise a un non-zero exit status ou alors on fait :

```
$ while (exit -1 ; `ls toto.sh`); do echo 12 ; done
```

L'itération avec while ou until (2)

```
#!/bin/bash
# fichier ee.sh
echo "bonjour"
exit -1

> while `/bin/bash ee.sh`; do echo 12; done
bonjour
>
```

Si on commente le exit -1 : boucle à l'infinie :

```
bonjour
12
bonjour
12
^C
```

Itérer sur une liste de n'importe quoi

```
#!/bin/bash
set "bonjour" 12 "titi" 32.34
while [ $# -gt 0 ]
do
    echo "-----($#): $1 -----"
    shift 1
done
>
-----(4): bonjour -----
-----(3): 12 -----
-----(2): titi -----
-----(1): 32.34 -----
```

Les variables du shell

- Comme dans tous les langages de programmation nous en avons besoin !
- Différentes méthodes pour affecter une variable selon le shell (bash, tcsh…) : EXPORT, SETENV…
- Le point délicat : l'évaluation des variables… qui peut être compliquée voire très compliquée !

Les variables

- Ce sont des objets « non typés » : elles peuvent contenir des chaînes, le résultat d’une commande, la valeur d’un paramètre positionnel…
- Dans l’exemple qui suit, remarquer que le j est collé au symbole = (règle pour l’affectation) et que j n’a pas de type et donc peut recevoir tout ce que l’on veut !
- Remarquer ensuite le $ pour obtenir le contenu et les deux ` qui permettent l’évaluation d’une commande

Exemple d’utilisation

```
#!/bin/bash
j="Bonjour" ; echo $j
j=$1 ; echo $j
j=`ls -al`; echo $j
```

```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/bash toto.sh CERIN
Bonjour
CERIN
total 1778384 drwxr-xr-x 197 cerin cerin 6698 3 Oct
04:19 . drwxrwxr-t 6 root admin 204 20 Mar 2004 .. -rw-
r--r-- 1 cerin cerin 4 20 Mar 2004 .CFUserTextEncoding
-rwxr--r-- 1 cerin cerin 15364 3 Oct 03:00 .DS_Store
-rw------- 1 cerin cerin 4583 7 Apr 19:22
```

- Le \ sert à ne pas interpréter le caractère qui suit (méta-caractère)
- Le double quote sert à « fabriquer des chaînes » et à garder la valeur littérale à l'exception de `$, `...`, \ ou newline`
- Le ` X` sert à évaluer « ce qu'il y a au milieu (X) ». Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed as follows: `$(`*`command`*`)` or `` ` ``*`command`*`` ` ``
- Le quote sert à empécher une évaluation : on garde la valeur littérale de l'objet entre quote

```
> var1=3
> $var1 + '2'
> bash: 3: command not found
> echo '$var1 + 2'
$var1 + 2
> echo "$var1 + 2"
3 + 2
> echo `$var1 + 2`
-bash: 3: command not found
> echo $(ls $var1)
ls: 3: No such file or directory
```

- Very difficult notion !
- Expansion : certaines expressions génèrent plusieurs objets -- ou comment des petites expressions génèrent de gros objets…
- Plusieurs sortes d’expansion : {}, ``…

Expansion

```
#!/bin/bash
j="a{d,c,b}e"; echo $j
j='a{d,c,b}e'; echo $j
for i in `ls -a{b,c,d}e`; do echo $i; done
for i in `ls -a{l,i,k}n`; do echo $i; done
```

- Les {} first

```
Ordinateur-de-Christophe-Cerin:~ cerin$ /bin/sh toto.sh
a{d,c,b}e
a{d,c,b}e
ls: option invalide -- e
Pour en savoir davantage, faites: « ls --help ».
total
91540
9519125
drwxr-xr-x
98
3074
1032
20
2007-09-26
18:49
.
9107
drwxr-xr-x
3
0
0
0
```

```
Exemple type d'utilisation :

mkdir /usr/local/src/bash/{old,new,dist,bugs}
```

Expansion… more difficult

```
${parameter:=word}
       Assign  Default  Values.   If  parameter  is  unset or null, the
       expansion of word is assigned to parameter.  The value of param-
       eter  is  then  substituted.   Positional parameters and special
       parameters may not be assigned to in this way.
${parameter:?word}
       Display Error if Null or Unset.  If parameter is null or  unset,
       the  expansion  of  word (or a message to that effect if word is
       not present) is written to the standard error and the shell,  if
       it is not interactive, exits.  Otherwise, the value of parameter
       is substituted.
${parameter:+word}
       Use Alternate Value.  If parameter is null or unset, nothing  is
       substituted, otherwise the expansion of word is substituted.
${parameter:offset}
${parameter:offset:length}
        Substring  Expansion.   Expands  to  up  to length characters of
        parameter starting at the character  specified  by  offset.   If
        length  is omitted, expands to the substring of parameter start-
        ing at the character specified by offset.  length and offset are
        arithmetic   expressions   (see  ARITHMETIC  EVALUATION  below).
        length must evaluate to a number greater than or equal to  zero.
        If  offset  evaluates  to  a number less than zero, the value is
```

- Lors d’une ouverture de session bash le fichier ~/.bashrc est exécuté : il contient des commandes
- Exemple de fichier ~/.bashrc :
  `alias ll='ls -al’` ce qui permet de construire un synonyme
- Vous pouvez y configurer votre prompt, vous déplacer dans un répertoire, etc

Les procédures / fonctions

- Procédure : retour de valeurs via les paramètres ;
- Fonction : le nom retourne une unique valeur ;
- Usages en informatique : favoriser la modularité = interface bien définie qui cache une décision d'implémentation c'est à dire faire de l'abstraction.

Les procédures : exemple introductif

```
# !/bin/bash
untar () {
      cp $1 /store
      tar -xvf $1
       echo 'Done'
}
```

Les procédures : exemple avec paramètres et shift

```
#!/bin/sh
# A simple script with a function...

add_a_user()
{
  USER=$1
  PASSWORD=$2
  shift; shift;
  # Having shifted twice, the rest is now comments ...
  COMMENTS=$@
  echo "Adding user $USER ..."
  echo useradd -c "$COMMENTS" $USER
  echo passwd $USER $PASSWORD
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}
```

Les procédures : variables locales

```
#!/bin/bash
# Global and local variables inside a function.

func ()
{
  local loc_var=23        # Declared as local variable.
  echo                    # Uses the 'local' builtin.
  echo "\"loc_var\" in function = $loc_var"
  global_var=999          # Not declared as local.
                          # Defaults to global.
  echo "\"global_var\" in function = $global_var"
}

func

# Now, to see if local variable "loc_var" exists outside function.

echo
echo "\"loc_var\" outside function = $loc_var"
                                      # $loc_var outside function =
                                      # No, $loc_var not visible globally.
echo "\"global_var\" outside function = $global_var"
                                      # $global_var outside function = 999
                                      # $global_var is visible globally.
echo

exit 0
```

```
#!/bin/bash
# fichier test3bis.sh
testX () {
    testY () {
        echo "TestY"
    }
    echo "TestX"
    testY   # appel a
}
$ source testX.sh
$ testY
-bash: testY: command not found
$ testX
TestX
TestY
$ testY
TestY
```

- C'est toujours la valeur de la dernière instruction exécutée dans la procédure
- Pas besoin de faire explicitement un return « à la C »
- Attention aux effets de bord ! (voir l'exemple qui suit)

Valeur de retour

```
#!/bin/bash
# fichier func4.sh
test4 () {
    for element in $@
      do
          # ignore all elements except "--x"
          if test "$1" = "--x"
          then
               logfile=$2
          fi
          shift
    done
    echo "logfile=$logfile"
    echo "3+4" | bc -l # NEW
}

$ source func4.sh
new-host-2:SE christophecerin$ test4
logfile=
7
```

- On veut écrire un programme qui prend en paramètre une chaîne de caractères et qui la renverse :
  - Utilisation des itérations sur les chaînes
- On veut écrire un programme qui prend en paramètre une chaîne contenant des caractères digit et qui fait la somme des digits
  - Composition d'un test sur les chaînes
  - Utilisation des itérations sur les chaînes

Programme inverse

```
#!bin/bash

##################
# Inverse
##################

if [ $# -ne 1 ]; then
  echo "passer une chaîne en paramètre"
  exit 0
fi

echo "${#1} est la longueur chaîne: $1"

for (( i=${#1};i>0;i=$((i-1)) ))↰
do
   echo -n "${1:$((i-1)):1}"
done

echo
```

Programme somme_digit

```
##################
# Somme des digits
##################

for (( i=0;i<${#1};i=$((i+1)) ))
do
   car=${1:$i:1}
   if [ "$car" \< ":" -a "$car" \> "/" ]
   then
         if [ $i -eq $((${#1}-1)) ];
         then
               echo -n "$car"
         else
               echo -n "$car+"
         fi
         somme=$((somme+$((car))))
   else
         echo "$car: pas un digit"
         exit 0
   fi
done

echo "=$somme"
```

Nouveautés en bash4.0 et points non couverts

- Variable de type tableau à une dimension : c'est possible en bash. Dans Bash 4.0 il y a aussi les dictionnaires (tableaux associatifs)
- … et certainement plein d'autres petites choses du genre :

```
$ a=3
$ ((a++))
$ echo $a
4
```

```
$ j='foo(){ printf "\x2a"; }'
$ eval $j
$ foo
*
```

Je crée une fonction
à la volée
Et je l'exécute

- Ils sont à une dimension
- Ils peuvent contenir n'importe quels types d'objets...
- Mais il faut faire attention à la notation pour les utiliser

Exemple d'utilisation des tableaux

```
#!/bin/bash

# Les tableaux à une dimension : exemples
# de base.
# Pour aller plus loin, voir le site :
# http://tldp.org/LDP/abs/html/arrays.html#EX66

# Declaration explicite (sans la taille) :
declare -a MonTableau
# ou aussi
declare -a MonTableauBis[10]

MonTableau[0]='12'
MonTableau[1]='bonjour'
echo "$((MonTableau[0])) -- $((MonTableau[1]))"
echo "${MonTableau[0]} -- ${MonTableau[1]}"

MonTableauBis[0]=12
v="bonjour"
MonTableauBis[1]=${v}
echo "$((MonTableauBis[0]))--${MonTableauBis[1]}"
```

Exemple d'utilisation des tableaux

```
cerin@ubuntu:~$ /bin/bash arrays.sh
12 -- 0
12 -- bonjour
12 - bonjour
```

Il faut donc utiliser les {} pour ne pas avoir de problème

Autres particularismes avec les tableaux

- Initialisation peut se faire aussi comme suit : `name=(value1 ... valuen)`
- Any element of an array may be referenced using the syntax `${name[subscript]}`.
- Extrait de la documentation Bash :

```
If the subscript is '@' or '*', the word expands to
    all members of the array name. These subscripts
    differ only when the word appears within double
    quotes. If the word is double-quoted, ${name[*]}
    expands to a single word with the value of each
    array member separated by the first character of
    the IFS variable, and ${name[@]} expands each
    element of name to a separate word. $
    {#name[subscript]} expands to the length of $
    {name[subscript]}. If subscript is '@' or '*', the
    expansion is the number of elements in the array.
    Referencing an array variable without a subscript
    is equivalent to referencing with a subscript of 0.
```

- L'exemple qui suit concerne une fonction qui prend en paramètre un tableau
- On veut accéder aux éléments du tableau, obtenir le nombre d'éléments du tableau
- Les notations sont pas simples, il faut truander – certaines notations ne sont pas possible !

The Good, the Bad and the Ugly

```
#!/bin/bash
# fonctionne en Bash 4 aussi

function print_array {
    # Setting the shell's Internal Field Separator to null
    OLD_IFS=$IFS
    IFS=''

    # On ne peut pas faire simple :
    # longueur=${#$1[*]}
    # ni
    # longueur=${#1[*]}

    # Create a string containing "colors[*]"
    local array_string="$1[*]"

    # assign loc_array value to ${colors[*]} using indirect variable reference
    local loc_array=(${!array_string})

    # Resetting IFS to default
    IFS=$OLD_IFS

    # Checking the second element "Light Gray" (the one with a space)
    echo ${loc_array[1]}

    # Print the length of loc_array
    echo ${#loc_array[*]}

}

# create an array and display contents
colors=('Pink' 'Light Gray' 'Green')
echo ${colors[*]}

# call function with positional parameter $1 set to array's name
print_array colors

# checking whether the function "local" loc_array is visible here
echo "Does the local array exist here? -->${loc_array[*]}<--"
exit 0
```

The Good, the Bad and the Ugly

```
$ /usr/local/bin/bash try1.sh
Pink Light Gray Green
Light Gray
3
Does the local array exist here? --><--
$ /usr/local/bin/bash --version
GNU bash, version 4.1.0(3)-release (i386-apple-darwin8.11.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.


$ /bin/bash try1.sh
Pink Light Gray Green

1
Does the local array exist here? --><--
$ /bin/bash --version
GNU bash, version 2.05b.0(1)-release (powerpc-apple-darwin8.0)
Copyright (C) 2002 Free Software Foundation, Inc.
```

Les dictionnaires

- On les appelle aussi les tableaux associatifs ;
- On associe une valeur à une clef (on associe une définition à un mot dans le Robert) ;
- Avantage : cout pour rechercher une valeur est en O(1) c.à.d que ça ne coûte rien (temps constant) alors que pour rechercher une valeur dans un tableau de taille N ça coûte N-1 comparaisons dans le pire cas !

Exemple d'utilisation dictionnaire

```
#!/bin/bash
# Declaration d'un tableau associatif
# c.a.d d'un dictionnaire
# La clef peut être de n'importe
# quel type

declare -A ASSOC

ASSOC[First]="first element"
ASSOC[Hello]="second element"
ASSOC[Peter Pan]="A weird guy"

# Utilisation
echo "${ASSOC[Hello]}"

exit

cerin@ubuntu:~$ /bin/bash arrays.sh
second element
```

- Mettre en place un schéma de mise en séquence d'actions (`; && ||`)
- Mettre en place une expression booléenne puis un choix (`test, if, case`)
- Mettre en place une itération (for, while, until) simple et l'itérateur `for i in ... done`
- Mettre en place une structuration des programmes (fonctions) et des données (tableaux, dictionnaires)
- Repérer les particularismes : expansions, différents rôles de {}, (()), [], et certaines règles syntaxiques (place des blancs, mots clés), 0=true, 1=false...