

2 - Programmation shell script

mickael.hoerdt@hesge.ch

Contenu

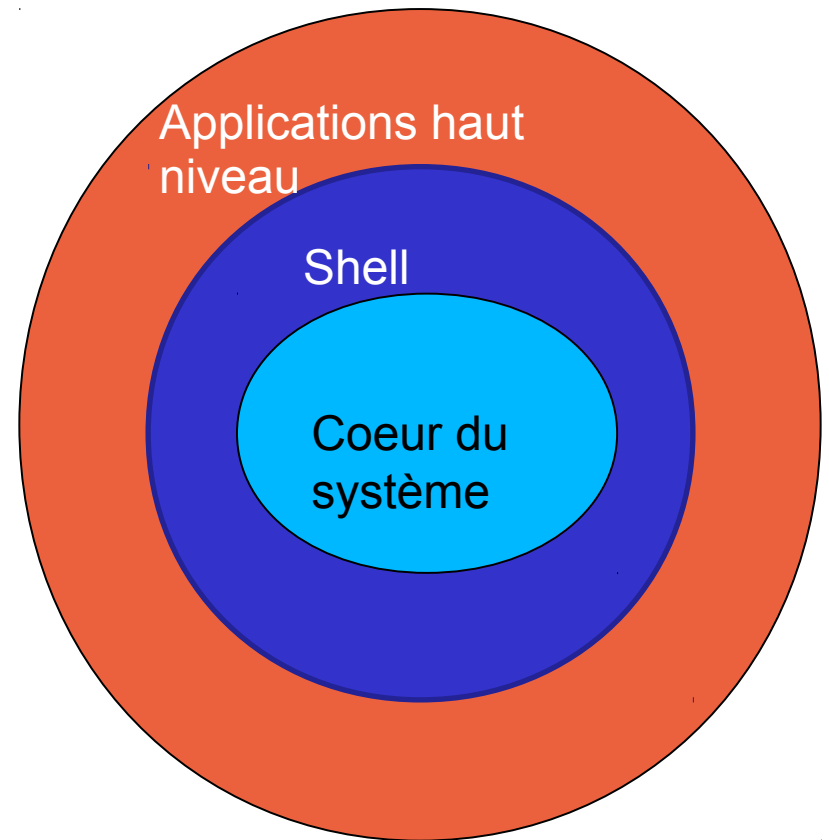
- Introduction
- Motifs de paramètres
- Substitution dans les paramètres, variables et expressions
- Conditionnelles et case
- Boucles
- Fonctions

Contenu

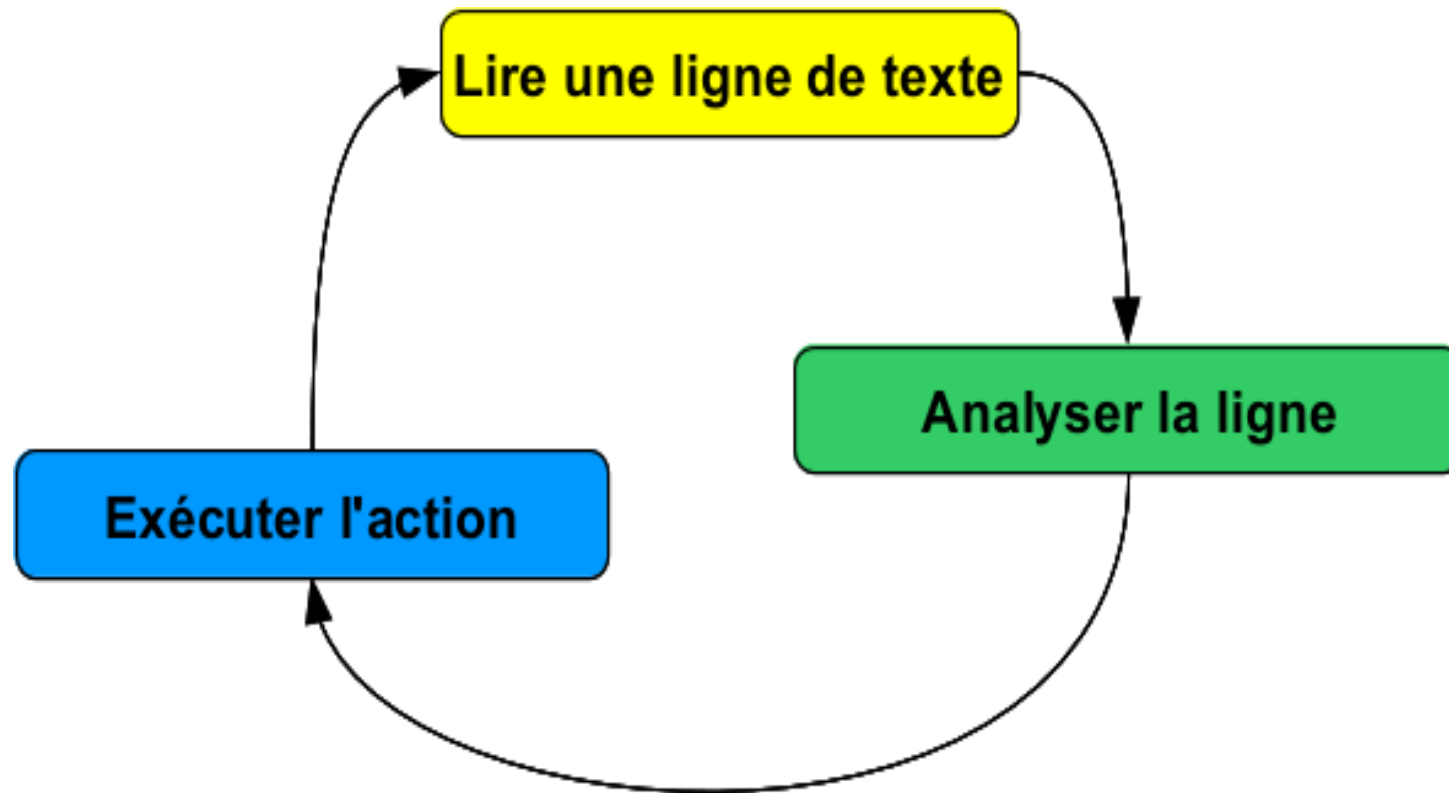
- Introduction
- Motifs de paramètres
- Substitution dans les paramètres, variables et expressions
- Conditionnelles et case
- Boucles
- Fonctions

Shell et système d'exploitation

- Le shell est une interface minimale (mais puissante !) d'accès aux fonctions du système.
- Les applications peuvent faire appel au shell et inversement.



Fonctionnement d'un shell



- Action peut-être une commande interne au shell ou un programme à charger depuis le système de fichier.
- La plupart des actions de ce cours sont des actions système (ls, mkdir, ps, kill, ...)
- Nous utilisons bash, il en existe d'autres (csh, ksh, etc..)

Syntaxe d'une commande shell

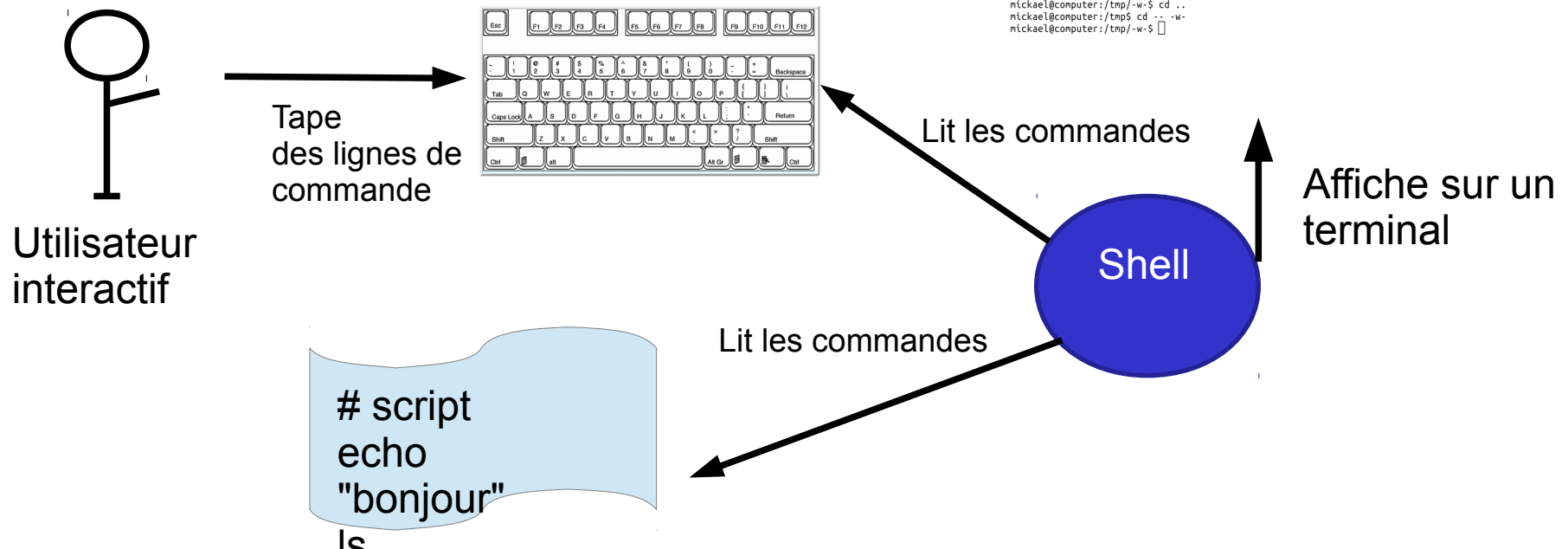
- Nom du programme à exécuter
 - Des options
 - Des paramètres, si ils existent
-
- Ces éléments sont séparés par des espaces.
 - Les options commencent par -
 - On indique la fin des options avec --

Exemple

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
mickael@computer:/bin$ ls --color=no -l ls
-rwxr-xr-x 1 root root 109944 nov 19 2012 ls
mickael@computer:/bin$ stat ls
  Fichier : «ls»
   Taille : 109944      Blocs : 216      ES blocs : 4096   fichier
Device : 812h/2066d    Inode : 9437378   Liens : 1
Accès  : (0755/-rwxr-xr-x) UID : (  0/   root)  GID : (  0/   root)
Accès  : 2013-09-25 09:27:29.541221498 +0200
Modi.  : 2012-11-19 23:20:57.000000000 +0100
Chgt   : 2013-05-18 15:54:26.222199909 +0200
Crée   : -
mickael@computer:/bin$ cd /tmp/
mickael@computer:/tmp$ cd -w-
bash: cd: -w : option non valable
cd : utilisation : cd [-L|[-P [-e]]] [dir]
mickael@computer:/tmp$ cd ./-w-
mickael@computer:/tmp/-w-$ cd ..
mickael@computer:/tmp$ cd -- -w-
mickael@computer:/tmp/-w-$
```

Scripts shell

- Les commandes shell peuvent provenir
 - D'un utilisateur
(fonctionnement interactif)
 - D'un **fichier** texte, appelé **script**.



Exemple de script

- Dans sa forme la plus simple, un shell **script** est une **suite de commande**

```
# exemple de script

echo nous sommes le
date

echo en Australie la date est
TZ='Australia/Adelaide' date
```

- Qui sont exécutées **séquentiellement**

Execution d'un script

- Un shell script est dans un fichier
 - On peut interpréter le script dans le shell courant avec la commande `source`.
 - `ex: source script.sh`
 - On peut le lancer manuellement en l'injectant dans un **nouvel** interpréteur (`ex: bash script.sh`)
 - `bash` peut exécuter un interpréteur (`python`, `bash`, `perl` ou autre) pour vous si on active l'autorisation `+x` sur le fichier et qu'on indique le chemin d'accès de l'interpréteur sur la première ligne, préfixé par `#!`
 - `ex: chmod u+x script.sh ; ./script.sh`
 - Première ligne de `script.sh` : `#!/bin/bash`
 - **Débugger un script : ajouter la ligne `set -x` en début de fichier**

Fichiers et Unix

- Remarque importante :
 - Pour Unix, que le shell lise depuis le clavier, ou bien depuis un fichier c'est la même chose !
 - L'entrée standard du programme shell pointe simplement vers un endroit différent, qui est un fichier (**au sens UNIX**) dans tous les cas !
 - Ainsi dans les cinq commandes suivantes, le programme bash effectue **exactement** le même traitement : les redirections sont transparentes aux processus.

```
1) echo "ls" | bash
2) echo "ls" > ls.sh ; bash < ls.sh
3) echo "ls" > ls.sh ; bash ls.sh

4) echo "ls" > ls.sh ; source ls.sh
5) ls
```

- Au niveau processus, quelle est la différence de la quatrième et de la cinquième commande avec les trois premières?

Intérêts des scripts

- Définir des séquences d'opérations complexes.
- Les faire exécuter facilement et rapidement.
- Tâches répétitives.
- Tâches programmées.

Programmer des scripts

- Un script ne contient pas seulement une suite séquentielle de commandes, il y a aussi possibilité:
 - d'évaluer des expressions arithmétiques
 - d'indiquer des paramètres au script
 - d'utiliser des variables
 - de faire des répétitions (boucles)
 - de prendre des décisions
 - de découper le script en fonctions.
- Le langage de commande d'un shell est donc un véritable langage de programmation.
- **Mais** :
 - Pas de primitive d'accès bas niveau au matériel (ex : outb du C).
 - Pas fait pour des tâches nécessitant du calcul intensif (trop lent).
 - Structure de données quasi-absentes, pas de pointeurs.
 - Aucune variable n'est typée : tout est une chaîne de caractère, ensuite le contexte décide du traitement (arithmétique ou chaîne).

En résumé

- Les shell-scripts
 - Sont des programmes qui lancent des commandes UNIX.
 - Permettent d'automatiser les tâches répétitives ou complexes.
- Pour écrire des scripts il faut.
 - Connaître les principales commandes UNIX, les mécanismes de redirection, pipe, etc.
 - Apprendre à écrire des boucles, des décisions, etc dans le langage de l'interprète.
- Le langage de commande est un véritable langage de programmation **mais à usage bien particulier.**

Contenu

- Introduction
- **Motifs de paramètres**
- Substitution dans les paramètres, variables et expressions
- Conditionnelles et case
- Boucles
- Fonctions

Motifs de paramètres chaînes (1)

- Si un paramètre contient :
 - `{x,y,...}` avec `x,y` étant des chaînes de caractères de longueur quelconque.
 - Le paramètre est vu comme un **motif** est à substituer par une liste de toutes les combinaisons possibles indiquées entre les `{ }`.
 - Il est possible de combiner des `{ }` entre eux
 - Exemples :
 - `echo a{d,c,b}e : echo ade ace abe`
 - `echo {a,b}{c,d} : echo ac ad bc bd`
 - `mkdir a a/{b,c},b/{f,g},c/{d,e},b/g/h`
 `-> mkdir a a/b a/c a/b/f a/b/g a/c/d a/c/e a/b/g/h`
 - `wget http://docs.example.com/slides_part{1,2,3,4,5,6}.html`

Motifs de paramètres chaînes (2)

- Si un paramètre contient :
 - `{x..y}` avec `x` et `y` un caractère alphabétique ou un nombre
 - Le paramètre est vu comme un **motif** est à substituer par la plage de tous les caractères/nombres possibles indiquées entre les {}, `x` et `y` inclus.
 - Il est possible de combiner des {} entre eux
 - Exemples :

```
- echo {0..10}:echo 0 1 2 3 4 5 6 7 8 9 10
```

```
- echo {a..e} : echo a b c d e
```

```
- echo {a..c}{0..2} : echo a0 a1 a2 b0 b1 b2 c0..
```

```
- wget http://docs.example.com/slides_part{1..6}.html
```

Motifs de paramètres fichiers (1)

- Si un paramètre contient les caractères *, ? ou [
 - Le paramètre est vu comme un **motif** dont la **correspondance** est à substituer par une **liste de fichier** triée alphabétiquement.
 - Règles de correspondance du motif :
 - * : toutes les chaînes de caractères, y compris la vide.
 - ? : Un seul caractère.
 - [...] : Une expression de caractère entre crochets.
 - [XYZ] : correspond soit au caractère X, soit à Y soit à Z
 - [X-Z] : corresponds à tous les caractères entre X et Z
 - [[:class:]] : corresponds à un caractère provenant des caractères d'une classe donnée: `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `space`, `upper`, `word`, `xdigit`
 - Exemples de motifs :
 - ?a* : tous les mots ayant 'a' en deuxième caractère
 - [a-e]* : tous les mots commençant par a,b,c,d ou e.
 - *{ab,cd} : tous les mots finissant par 'ab' ou 'cd'

Motifs de paramètres fichiers (2)

- Exemples

Fichier Édition Affichage Rechercher Terminal Aide

```
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ ls -l
total 0
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:13 hello
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:13 Hello
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:13 hello world
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:13 Helloworld
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:13 Hello world
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:14 Hello world?
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:14 Hello world\!
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:14 Hello world+
-rw-rw-r-- 1 mickael mickael 0 sep 30 09:13 HelloWoRlD
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ echo *
hello Hello hello world Helloworld Hello world Hello world? Hello world\! Hello world+ HelloWoRlD
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ echo [Hh]"ello world"
hello world Hello world
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ echo Hello*
Hello Helloworld Hello world Hello world? Hello world\! Hello world+ HelloWoRlD
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ echo "Hello world"*
Hello world Hello world? Hello world\! Hello world+
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ echo "Hello world"[:punct:]
Hello world? Hello world+
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$ echo Hello*[dD]
Helloworld Hello world HelloWoRlD
mickael@computer:~/work/current/hepia/programmation système/semaine_3_Cours_programmation_shell/présentation/exemple$
```

Motifs de paramètres fichiers (3)

- Exemple navrant

Fichier Édition Affichage Rechercher Terminal Aide

```
mickael@computer:/tmp/navrant$ ls -l
total 0
-rw-rw-r-- 1 mickael mickael      0 sep 30 16:28 *
-rw-rw-r-- 1 mickael mickael 142438 sep 30 16:29 data.bin
-rw-rw-r-- 1 mickael mickael   1278 sep 30 16:29 FICHER_HYPER_IMPORTANT.txt
-rw-rw-r-- 1 mickael mickael   2894 sep 30 16:30 programme.c
mickael@computer:/tmp/navrant$ rm *
mickael@computer:/tmp/navrant$ ls -l
total 0
mickael@computer:/tmp/navrant$
```

ARGH !!

- Que s'est-il passé ?

- Toujours tester et re-tester un script en préfixant les commandes par **echo** avant de le lancer !
- Vérifier qu'il fonctionne dans tous les cas, en particulier sur les noms de fichiers particulier (avec espace par ex).
- Voir option **-i** de **rm**

Contenu

- Introduction
- Motifs de paramètres
- Substitution dans les paramètres, variables et expressions
- Conditionnelles et case
- Boucles
- Fonctions

Substitution de paramètre

- Tout paramètre du shell qui commencent par un \$ est **substitué** par une valeur calculée par le shell.
- Le nom du paramètre peut être référencé entre {} pour éviter l'ambiguïté sur son nom lors de la **substitution**. Ces accolades ne sont pas obligatoires.
 - Ex : `${mon_parametre}`
- La commande est exécutée **après** la substitution.
 - Les commandes ne reçoivent que le **résultat** de la substitution par le shell !
 - Ex : `echo $USER => echo mickael => mickael`

Trois types de paramètres

- Les **paramètres positionnels** commencent obligatoirement par un chiffre
 - Ex : `$1`, `${10}`
 - Pourquoi les accolades sur le 10 ?
- Les **variables**, dont le nom contient uniquement soit des lettres majuscules ou minuscules soit des underscore (`_`)
 - Ex : `$votre_login`, `$_VoTrE_lOgIN`
- Les **paramètres spéciaux** qui commencent par un symbole spécial, et qui sont renseignés automatiquement par le shell pour différents usages.
 - Ex : `$$`, `$@`

Paramètres positionnels

- En supposant qu'un script est invoqué de la manière suivante :

```
./mon_script.sh geneve hepia "cours systeme"
```

- Pendant l'exécution du script :
 - \$1 = "geneve"
 - \$2 = "hepia"
 - \$3 = "cours systeme" => en deux mots et "" !
- Très similaire à argc et argv[] du C !
 - \$# = nombre de paramètres
 - \$* = \$@ = geneve hepia cours systeme
- Et \$0 ?

Variables et expressions arithmétiques

- Affectation :
 - Deux formes
 - Affectation simple
 - `NOM=CHAINE`
 - sans espace entre le nom de variable et sa valeur !
 - `CHAINE` peut être elle même une variable à substituer
 - Ex : `ville="geneve";current_ville=$ville;echo $ville`
 - Affectation avec calcul arithmétique sur des entiers :
 - `let NOM=EXPRESSION`
 - `EXPRESSION` est une expression arithmétique (voir **help let**)
 - Exemple : Ex : `long=2;larg=3 ; let p=long*larg ; echo $p`
 - Essayer : `echo $((2**64))`
 - Pour calculer sur autre chose que des entiers ou en précision arbitraire
 - voir la commande **bc**

Variables d'environnement

- Beaucoup de variables sont déjà pré-définies par le shell :
 - `PWD`, `USER`, `HOME`, `PATH`, ...
 - On obtient la liste avec la commande `env`
- On appelle ces variables, "variables d'environnement"

Variables

- Substitution
 - trois formes
 - Substitution de variables :
 - Ex: `echo bonjour $USER`
 - Substitution par la valeur d'une d'expression arithmétique :
 - Ex: `echo périmètre=$((2*(longueur+hauteur))`
 - Substitution par le résultat d'une commande :
 - Ex: `echo mon nom utilisateur est $(who am i)`
- La substitution
 - se fait dans les chaines délimitée par "..."
 - pas dans les chaines délimitées par '...'
 - Ex: `echo 'la variable $USER' "contient $USER".`

Variables : Affectation & substitution

- Essayer :

```
# exemple de script
set -x

a=12
b=42

c=a+b
echo $c

d=$((a+b))
echo $d

let e=a+b
echo $e

f=$((a+b))
echo $f

g=$((2#1111))
echo $g
```

Variables : substitution avec traitement

Choisir entre deux substitutions de paramètre p et w selon des critères précis :

- Substituer w si p est vide ou indéfini : $\${p}:-w\}$
 - Si p est indéfini ou vide, le shell substitue w . Sinon c'est p qui est substitué. **La valeur de p change pas !**
 - Ex :

```
p="";w="abcd";x=${p}:-$w}; echo $x
```

```
p="efgh";w="abcd";x=${p}:-$w}; echo $x
```
- Substituer w si p est déjà défini ou non vide : $\${p}:+w\}$
 - Si p est défini ou non vide, le shell substitue w . Sinon il ne substitue rien.
 - Ex :

```
p="efgh";w="abcd";x=${p}:+$w}; echo $x
```
- Copier w dans p si p est vide ou indéfini : $\${p}:=w\}$
 - Si p est indéfini ou vide, le shell assignera à p la valeur de la substitution de w . Sinon p reste inchangé. Ensuite p est substitué.
 - Ex :

```
p="";w="abcd";x=${p}:=w}; echo $p
```

Variables : substitution avec traitement

- Usage typiques

- `${p:-w}` : `w` est un paramètre par défaut, utilisé si `p` n'est pas défini.
 - Ex : `working_dir="${1:-/tmp}"`
 - Si le premier paramètre (paramètre positionnel) du script est défini, il sera assigné à la variable `working_dir`, sinon c'est `"/tmp"` qui sera utilisé.
- `${p:+w}` : afficher un texte si `p` est défini
 - Ex : `echo ${USER:+votre nom est $USER}`
 - Si `USER` est défini, affiche qqch sinon n'affiche rien.
- `${p:=w}` : La même chose que pour `${p:-w}` mais on assigne la variable `p` pour une utilisation ultérieure (si `p` non défini).
 - Ex : `echo Repertoire de travail : ${HOME:=/home/$USER}`
`echo $HOME sera votre repertoire de travail.`

Variables : substitution avec traitement

Traiter des chaînes de caractères :

- `${var:offset}`, `${var:offset:length}`, `${#var}`
 - Renvoie la sous-chaîne `$var` en commençant à l'index de caractère `offset`, l'index commençant par 0 et à gauche de la chaîne. Si `length` est omis, renvoie tout le reste de la chaîne depuis l'index, sinon s'arrête après `length` caractères. Le caractère `#` indique qu'il faut renvoyer la longueur de la chaîne.
- Ex : si `var="12345"`, `${var:1}` renvoie 2345, `${var:0:1}` renvoie 1 et `${#var}` renvoie 5.
- `${var^}`, `${var^^}`, `${var,}`, `${var,,}`
 - Met respectivement en majuscule le premier caractère de `$var`, toute la chaîne de caractère `$var`. Pour l'opérateur « , », fait la même chose, mais convertit en minuscule.
- Ex : si `var="ab"`, `${var^}` renvoie Ab et `${var^^}` renvoie AB
- `${var/motif/subst}`, `${var//motif/subst}`
 - Remplace respectivement la première occurrence de `motif` par `subst` ou toutes les occurrences de `motif` par `subst`. `Subst` peut être une chaîne vide.
- Ex : si `var= "titi"`, `${var/ti/to}` renvoie "toti" `${var//ti/to}` renvoie "toto" et `${var/ti/}` renvoie "ti".

Variables : substitution avec traitement

Traiter des chaînes de caractères avec motif:

- `${var#motif}`, `${var##motif}`
 - **Enlève** le plus court (#) ou le plus long (##) motif correspondant de la chaîne `var` en partant depuis **la gauche/le début** de la chaîne. Les motifs sont exprimés de la même manière que les motifs de fichiers.
 - Ex : si `var='Salut !? ça un bail! Comment vas-tu ?'`,
`${var#*\!}` renvoie `"? ça un bail! Comment vas-tu ?"`
`${var##*\!}` renvoie `" Comment vas-tu ?"`
 - Note : on utilise un `\` parce que `!` est un caractère spécial dans `bash` (gestion de l'historique des commandes)
- `${var%motif}`, `${var%%motif}`
 - La même chose que pour # et ##, à la seule différence que c'est en partant depuis **la droite/la fin** de la chaîne.
 - Ex : `var` garde la même valeur qu'avant.
`${var%\!*}` renvoie `"Salut !? ça fait un bail"`
`${var%%\!*}` renvoie `"Salut "`

Variables : substitution avec traitement

- Usage typiques
 - Extraire le nom d'un fichier sans l'extension :
 - `${FILENAME%.*}`
 - On enlève la plus courte chaîne quelconque depuis la fin/droite et commençant par un point.
 - Extraire l'extension d'un fichier :
 - `${FILENAME##*.}`
 - On enlève la plus longue chaîne quelconque depuis le début/gauche terminant par un point.
 - Extraire le répertoire d'un chemin d'accès :
 - `${PATHNAME%/ *}`
 - On enlève la plus courte chaîne quelconque depuis la fin/droite et commençant par un /
 - Extraire nom de fichier d'un chemin d'accès :
 - `${PATHNAME##*/}`
 - On enlève la plus longue chaîne quelconque depuis le début/gauche terminant par un /
- Voir aussi les commandes `dirname` et `basename`

Variables : substitution avec traitement

Substituer avec indirection (ressemble aux pointeurs):

- Substituer le paramètre dont le nom est obtenu après avoir substitué `p` : `${!p}`
 - Ex : `var="abcd";pvar="var";x=${!pvar}; echo $x`
 - à `x` sera substitué `$var`, c'est à dire « abcd »

Variables : tableaux

- Deux types : associatifs ou indexés sur un entier positif.
 - Déclaration :
 - `ARRAY=()` déclare un tableau indexé.
 - `declare -A ARRAY` déclare un tableau associatif.
 - Stocker des valeurs :
 - `ARRAY[N] = value` où N est un entier.
 - `ARRAY = (V1, V2, ..., VX)` où V1, V2 ... sont stocké aux indexes 0,1, ..., X
 - `ARRAY[STRING] = value`, ou STRING est une chaine de caractères.
 - Récupérer des valeurs :
 - `${ARRAY[N]}` = la valeur à l'indexe N.
 - `${ARRAY[STRING]}` = la valeur à l'entrée STRING
 - `${ARRAY[@]}` = toutes les valeurs du tableau.
 - `${#ARRAY[@]}` = le nombre d'éléments du tableau.

Variable : lecture externe

- Utilisation de la commande interne **read** qui lit une ligne depuis l'entrée standard pour l'affecter à une variable
 - Lecture d'une seule variable
 - Ex : `read name ; echo Hello $name`
 - Lecture d'une suite de variable
 - Ex : `read firstname lastname ;`
`echo Hello $firstname $lastname`
 - Si il y a plus de mots lus que de variable
 - La dernière variable contient le reste des mots.
 - Lecture d'une suite de variable dans un tableau
 - Ex : `read -a text ;`
`Ceci est un texte`
`echo ${text[0]}`
 - Lecture depuis un fichier
 - Ex : `read line < fichier.txt ; echo $line`

IMPORTANT : Ordre de substitution (1)

- Le shell substitue pour chaque ligne de commande, **une seule fois** et successivement les paramètres ou variables **dans cet ordre** :
 1. Motif de paramètre chaîne, ex : {a..e}
 2. Variables, ex : \$1, \$a
 3. Résultats de la sortie d'une commande, ex : \$(ls)
 4. Expressions arithmétiques, ex : \$((1 + 2))
 5. Motifs de paramètres fichiers, ex : *
- Ex : Dans le code suivant : `a=* ; ls $a`
 - La valeur « * » est affectée à \$a (règle 2), qui est substituée par * dans la commande ls (règle 2), qui sera exécutée avec comme paramètre la liste de tous les fichiers du répertoire de travail courant (règle 5).

IMPORTANT : Ordre de substitution (2)

- Selon l'ordre précédent :
 - `echo {1..3}`
 - Affichera 1 2 3 (règle 1, **une seule fois**)
 - `a=3 ; echo {1..$a}`
 - Affichera {1..3} (règle 2, **une seule fois**,
ne revient pas sur la règle 1 !)
 - Comment obtenir 1 2 3 avec une variable alors ?
 - Avec la commande `eval`
 - `a=3 ; eval echo {1..$a}` (règle 2, **une seule fois**)
 - > `eval echo {1..3}` (règle 1 **une seule fois**)
 - > `eval echo 1 2 3`

Contenu

- Introduction
- Motifs de paramètres
- Substitution dans les paramètres, variables et expressions
- Conditionnelles et case
- Boucles
- Fonctions

Décisions

- Code de sortie (exit status)
 - Le **code de sortie** d'un programme indique si il s'est bien terminé ou pas par un entier positif ou égal à 0
 - Par convention un code de retour 0 signifie OK
 - Après le lancement d'une commande, le code de sortie est renseigné par le shell dans la variable `$?`
 - Pour quitter un script à la première commande dont le code retour n'est pas 0/OK : ajouter `set -e` en début de script

Code de sortie

- Exemple

```
Fichier  Édition  Affichage  Rechercher  Terminal  Onglets  Aide
hoerdtm@A433-01: ~ x hoerdtm@A406-07: ~/programmation_systeme/TP1 x
hoerdtm@A406-07:~/programmation_systeme/TP1$ ls -ld /tmp/
drwxrwxrwt 10 root root 4096 sep 25 22:42 /tmp/
hoerdtm@A406-07:~/programmation_systeme/TP1$ echo code de retour = $?
code de retour = 0
hoerdtm@A406-07:~/programmation_systeme/TP1$ ls -l sdqflkfhj
ls: impossible d'accéder à sdqflkfhj: Aucun fichier ou dossier de ce type
hoerdtm@A406-07:~/programmation_systeme/TP1$ echo code de retour = $?
code de retour = 2
hoerdtm@A406-07:~/programmation_systeme/TP1$
```

Code de sortie et documentation

- Les codes de retour sont décrits dans les pages `man`

Fichier Édition Affichage Rechercher Terminal Aide

Using `color` to distinguish file types is disabled both by default and with `--color=never`. With `--color=auto`, `ls` emits color codes only when standard output is connected to a terminal. The `LS_COLORS` environment variable can change the settings. Use the `dircolors` command to set it.

Exit status:

- 0 if OK,
- 1 if minor problems (e.g., cannot access subdirectory),
- 2 if serious trouble (e.g., cannot access command-line argument).

AUTHOR

Written by Richard M. Stallman and David MacKenzie.

REPORTING BUGS

Report `ls` bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
Report `ls` translation bugs to <http://translationproject.org/team/>

COPYRIGHT

Copyright © 2011 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.

SEE ALSO

The full documentation for `ls` is maintained as a Texinfo manual. If the `info` and `ls` programs are properly installed at your site, the command

Commande exit

- Par défaut un script retourne le code de sortie de la dernière commande exécutée.
- Il peut retourner un code spécifique par
 - `exit [numero]`
- Exemple

```
# exemple de script  
  
echo nous sommes le  
date  
  
exit 10
```

If ... then ... fi

- La structure de contrôle `if then else fi` utilise le code de sortie d'une commande

```
if gcc -o $1 $1.c
then
    echo "La compilation a réussi"
else
    echo "La compilation a échoué"
fi
```

- `else` est optionnel.

test et if

- Le code de sortie du programme test **dépend** d'une condition indiquée en paramètres
 - Exemple
 - `test -f superfichier`
 - `$?` indiquera si le fichier a été trouvé
- On peut donc combiner `test` avec la structure de contrôle `if` :

```
if test -f $1.c
then
    gcc -o $1 $1.c
else
    echo "fichier $1 inexistant"
    exit 1
fi
```

test et if

- Autre notation : `[ condition ]`
- Autres conditions :

- <code>[-d nom]</code>	existence d'un répertoire
- <code>[chaine1 = chaine2]</code>	comparaison de chaines
- <code>[chaine1 != chaine2]</code>	
- <code>[chaine1 \< chaine2]</code>	
- <code>[chaine1 \> chaine2]</code>	
- <code>[nombre1 -eq nombre2]</code>	comparaison de nombres
- <code>[nombre1 -ne nombre2]</code>	
- <code>[nombre1 -lt nombre2]</code>	
- <code>[nombre1 -gt nombre2]</code>	
- <code>[nombre1 -le nombre2]</code>	less or equal
- <code>[nombre1 -ge nombre2]</code>	greater or equal

- voir la commande `help test`

Enchainement conditionnel

- Syntaxe
 - `COMMANDE1 && COMMANDE2`
 - `COMMANDE1 || COMMANDE2`
- Exécute la seconde commande seulement si la première a réussi (`&&`) ou échoué (`||`).
- Exemple :
 - `gcc prog.c && echo "compilation OK"`
 - `read age; [ $age -le 25 ] && echo "vous êtes bien jeune !"`
- Exercice :
 - Écrire un script qui affiche le maximum de deux nombres fournis en paramètre.
 - Affectation
 - Test `-ge`
 - Affectation conditionnelle

Conditions multiples

- Les commandes peuvent être reliées entre elles par les symboles d'enchaînement conditionnels `&&` ou `||`

Ex :

```
if [ -f $1.c ] && gcc $1.c
then
    echo "fichier $1.c compilé"
else
    echo "fichier $1 inexistant ou échec de compilation"
    exit 1
fi
```

- Attention : dans la conditionnelle au niveau de l'exécution de `CMD1 && CMD2`
 - `CMD2` n'est exécuté par le shell que si `CMD1` réussit
- Au niveau de l'exécution de `CMD1 || CMD2`
 - `CMD2` n'est exécuté par le shell que si `CMD1` échoue
 - Ex : `gcc` ne sera pas exécuté si `$1.c` existe dans la commande `if [ -f $1.c ] || gcc $1.c`

Case

- Utile pour traiter des options fournies en paramètre du script.
- Syntaxe

```
case chaine in
    motif1)
        COMMANDES
        ;;

    motif2)
        COMMANDES
        ;;

    *)
        COMMANDES
        ;;
esac
```

- Les motifs suivent les mêmes règles syntaxiques que pour les motifs de fichiers.

Case

- Exemple

```
echo "Ecrire 1 ou 2 en toute lettre:"
read v1
case $v1 in
    [Uu]n)
        echo "Vous avez écrit 1"
        ;;
    [Dd]eux)
        echo "Vous avez écrit 2"
        ;;
    (*)
        echo "Chiffre inconnu"
        ;;
esac
```

Contenu

- Introduction
- Motifs de paramètres
- Substitutions dans les paramètres, variables et expressions
- Conditionnelles et case
- **Boucles**
- Fonctions

Boucles while

- Syntaxe :

```
- while COMMANDE
do
    COMMANDES
done
```

- La condition de la boucle est le code de retour de la commande, **comme pour if**

- Ex :

```
Compteur=0 ;
while [ $compteur -lt 25 ]
do
    let compteur++
done
```

- L'instruction `break` permet de sortir d'une boucle si besoin

- Ex :

```
while true
do
    ...
    echo "voulez-vous arreter ?"
    read response
    [ $response = oui ] && break
    ...
done
```

- L'instruction `continue` permet de sauter à la prochaine itération

Boucle while : traitement de fichiers

- Conversion binaire – décimal d'un fichier indiqué en paramètre contenant deux nombre par ligne séparés par des espaces

```
while read num1b num2b
do
    num1=$((2#$num1b))
    num2=$((2#$num2b))
    echo $num1 $num2
done < $1
```

- Si les nombres sont séparés par autre chose, utiliser la variable **IFS**

```
IFS="," ; do while read num1b num2b ...
```

Boucle for : première forme

- Syntaxe :

```
for i [in] [liste]
do
COMMANDES
done
```

- `liste` est une suite de mots séparés par des espaces ou des retours à la ligne (`\n`). `liste` est optionnel.
- Si pas de liste fournie, c'est `$@` (liste de la valeur des paramètres positionnels = `$1 $2 $3 ...`) qui sera utilisé pour la boucle.
- Cette liste peut-être issue d'une variable, d'une expression ou d'un motif de fichiers/généré.
 - Ex :

```
for i in *
do
    mv $i $i.tmp
done
```

```
do
    ...
done
```

```
for i in item{0..10}
do
    ...
done
```

```
for i in $var
```

Boucle for : deuxième forme

- Syntaxe très proche du C:

```
for ( ( INIT ; LOOPTEST ; COUNTING) )  
do  
COMMANDES  
done
```

- Ex :

```
for ( (i=0; i<5; i++) )  
do  
    ...  
done
```

- Boucle infinie : `for ( ( ; ; ) ) ; do ... ; done`

Contenu

- Introduction
- Motifs de paramètres
- Substitutions dans les paramètres, variables et expressions
- Conditionnelles et case
- Boucles
- Fonctions

Fonctions

- Un script peut comporter des **fonctions** avec des **paramètres positionnels**

- Les avantages : code plus facile à lire, découpage logique, réutilisation de code.

- Syntaxe

```
function nom-de-fonction
{
    COMMANDE
    COMMANDE
    ...
}
```

- Exemple

```
function archiver
{
    tar -czf /var/svgd/$1.tgz $2
}

archiver photos /home/hoerdtm/photos
archiver musique /home/hoerdtm/musique
```