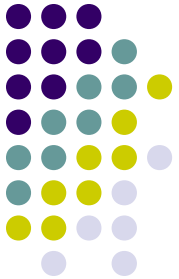
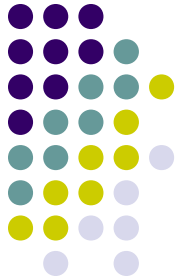


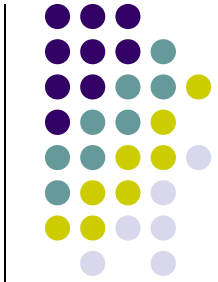
Le shell
Principales commandes

Qu'est-ce que le shell ?



- Le shell est un programme exécutable en mode terminal, dont la principale fonction est de permettre à l'utilisateur d'interagir avec le système via un terminal.
- Il est parfois appelé *interpréteur de commandes*.
- Deux modes d'utilisation :
  - Interactif : l'utilisateur saisit et exécute ses lignes de commandes une par une dans un terminal ;
  - Non interactif : le shell lit un ensemble de commandes à partir d'un fichier appelé *shell script*.
- Il existe aujourd'hui plus d'une trentaine de shells différents, mais deux grandes familles dominent :
  - Csh, tcsh : shells orientés administration, avec une syntaxe inspirée du langage C

Qu'est-ce que le shell ?



- Sh (à l'origine: ash), bsh (Bourne shell), bash (Bourne again shell) : shells orientés utilisateur, majoritaires aujourd'hui. La plupart des scripts shell sont écrits en sh, ou au moins compatibles sh.

Le shell UNIX standard est sh. Nous n'étudierons que lui cette année.

Bash supplante de plus en plus souvent sh (c'est le cas sur Linux). Il consiste en un mélange de sh, de quelques fonctions du csh, et d'autres du Korn shell (ksh), mais il est 100% compatible sh.

Principe d'exécution

Principe général d'exécution : le shell

1) Lit une **ligne de commande** soit à partir du terminal, soit à partir d'un fichier script
2) Effectue une première analyse qui détermine quels sont les **mots** de cette ligne, et quels en sont les **délimiteurs** (espace, ‘, **"**, tabulations)**.**
3) Catégorise les mots trouvés en : **opérateurs**, **chaînes de caractères**, et **mots ordinaires**.
4) Compile la ligne de commande en **commandes simples**, en appliquant un jeu de priorités fixé sur les opérateurs identifiés
5) Evalue chaque commande simple en attendant (exécution séquentielle) ou en n'attendant pas (exécution parallèle) sa terminaison pour évaluer suivante selon la présence de l'opérateur &.

Grammaire

- Les mots (comme les opérateurs) sont soit des suites ininterrompues de caractères, soit des chaînes de caractères délimitées par ‘ ou ".
- Deux types d’opérateurs :
  - ***Contrôle*** : ils servent à séparer deux commandes ou deux listes. Ce sont: `&  &&  ( )  {}  ;  ;;  |  ||`
  - ***Redirection*** : ils servent à rediriger les entrées/sorties et portent sur une seule commande. Ce sont: `<  >  >|  <<  >>  <&  >&  <<-  <>`

Grammaire

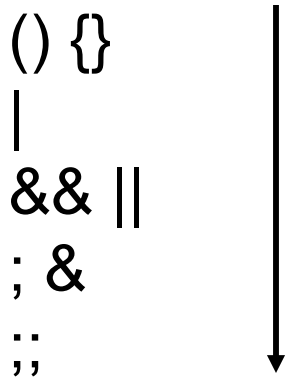
Une ligne de commande est :

- **simple** si elle ne comporte aucun opérateur de contrôle ;
- **composée** dans le cas contraire.

Priorités des opérateurs de contrôle :

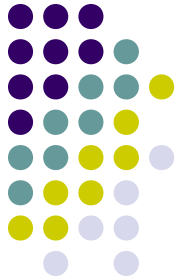
Priorités égales (*)

(*) L'opérateur rencontré le premier remporte la priorité



Priorités croissantes (*priment sur les priorités horizontales*)

Grammaire



Un exemple : traitement de la ligne de commande

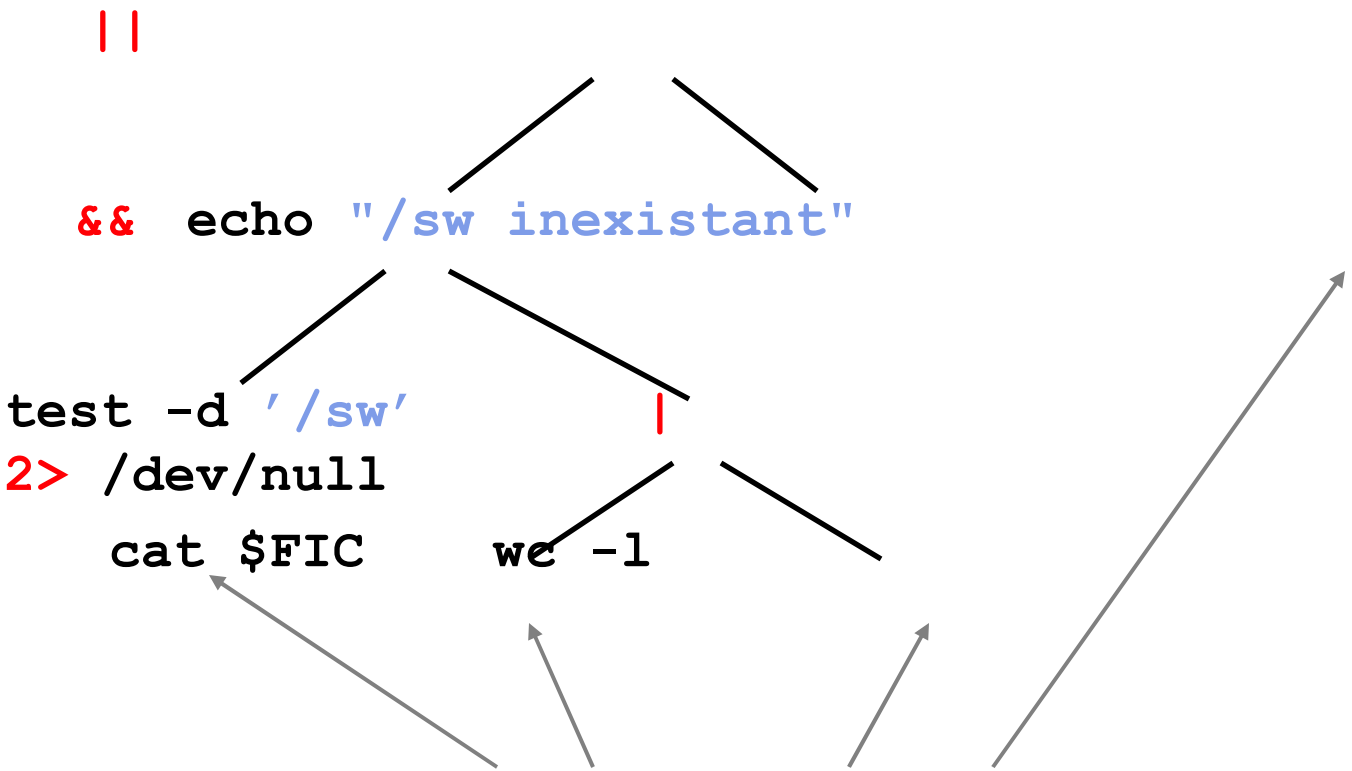
```
 test -d   '/sw'   2>/dev/null   <tab>   && cat
$FIC | wc -l || echo  "/sw inexistant"
```

Etapes 1 à 3 : après identification des mots (ordinaires, chaînes de caractères) et des opérateurs, on obtient :

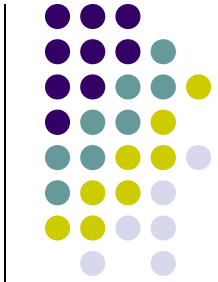
```
test -d '/sw' 2> /dev/null && cat $FIC | wc -l
|| echo  "/sw inexistant"
```

Grammaire

Etape 4 : arbre compilé :



Commandes simples



Une commande simple a la forme suivante:

`[assign] [com] [args…] [redir]`

- **assign** consiste en aucune, une ou plusieurs affectations de *variables* portant sur la commande qui suit (et seulement elle)
- **com** est la commande elle-même. Il s'agit soit d'un nom de commande interne, soit d'un nom de fichier exécutable accessible par la *variable d'environnement* PATH.
- **args** sont les arguments optionnels de la commande
- **redir** consiste en aucune, une, ou plusieurs redirections de fichiers

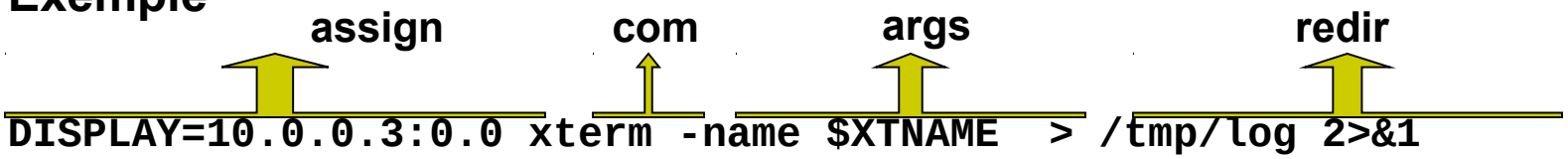
Commandes simples

Evaluation d'une commande simple : ce qui se passe (dans l'ordre)

1. Les assignations **[assign]** sont provisoirement retirées de la ligne de commande
2. Ce qui reste subit le **processus de développement** du shell
3. Les redirections **[redir]** sont appliquées, puis retirées de la ligne de commande
4. Les assignations sont replacées sur la ligne de commande. Deux cas de figure peuvent se produire:
    1. **[nom]** est vide → les variables affectées par **[assign]** sont celles du shell courant.
    2. **[nom]** n'est pas vide → les variables affectées par **[assign]** sont celles de la commande [com] qui va être lancée

Commandes simples

Exemple



1. Assignations: `DISPLAY=10.0.0.3:0.0` est retiré

`xterm -name $XTNAME > /tmp/log 2>&1`

2. Développement : une seule substitution à faire ici, `$XTNAME` qui est une variable, valant myterm par exemple

`xterm -name myterm > /tmp/log 2>&1`

3. Redirections : `> /tmp/log` indique que la sortie standard doit être redirigée vers le fichier `/tmp/log`, `2>&1` que la sortie d'erreur standard et la sortie standard doivent être unifiées

`xterm -name myterm`

4. Assignations : `DISPLAY=10.0.0.3:0.0` assigne `10.0.0.3:0.0` à la variable DISPLAY, et le reste de la ligne est lancé avec cette valeur :

`xterm -name myterm`

Variables

- Le shell connaît deux types de variables : **ordinaires** et **d'environnement**.

- Une **variable ordinaire** n'est connue que par le shell : aucun des programmes que le shell lancera ne pourra la consulter/modifier.

- A l'inverse, une **variable d'environnement** est une variable ordinaire dont les processus lancés par le shell recoivent une copie.

- On peut assigner une valeur à une variable avec l'opérateur '=' (attention: pas d'espace)

```
$ mavariable=3
$ echo $mavariable
3
```

- Pour faire une variable d'environnement d'une variable ordinaire, on utilise la commande **export** :

```
$ sh -c 'echo $mavariable' # lance la com. dans un sous-shell
$ export mavariable
$ sh -c 'echo $mavariable'
3
```

Développement

C’est une technique de réécriture très puissante durant laquelle les variables, les noms de fichiers, et certains caractères sont substitués soit par leur propre valeur, soit par un traitement sur cette valeur.

Développement de variables

Forme générale: ${expression}

Forme	var définie	var indéfinie
${var} ou $var	Substitue var	Substitue null
${var:-mot}	Substitue var	Substitue mot
${var:+mot}	Substitue mot	Substitue null
${var:?[mot]}	Substitue var	Erreur + substitue mot si spécifié
${var:=mot}	Substitue var	Assigne mot à var; substitue mot

Développement

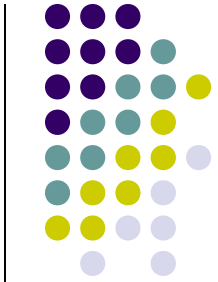
Forme	Substitution
${#var}	Longueur de la chaîne var
${var%motif}	var privée de la plus courte occurrence droite de motif
${var%%motif}	var privée de la plus longue occurrence droite de motif
${var#motif}	var privée de la plus courte occurrence gauche de motif
${var##motif}	var privée de la plus longue occurrence gauche de motif

```
$ var=aaabccc
$ echo $var
aaabccc
$ echo ${va:-mot}
mot
$ echo ${var:-mot}
aaabccc
$ echo ${var:+mot}
mot
$ echo ${va:?test}
```

```
va: test
$ echo ${va:?}
va: parameter null or
not set
$ echo ${va:=mot}
mot
$ echo $va
mot
$ echo ${#var}
7
```

```
$ echo ${var%e}
aaabccc
$ echo ${var%c}
aaabcc
$ echo ${var%%c}
aaabcc
$ echo ${var%c*}
aaabcc
$ echo ${var%%c*}
aaab
```

Développement



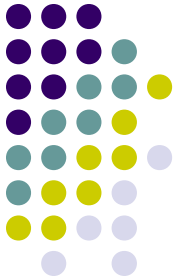
Variables spéciales (utilisables seulement dans un script) :

Nom	Substitué par
*	L'ensemble des paramètres concaténés en une seule chaîne
@	L'ensemble des paramètres en autant de chaînes que nécessaire
#	Nombre de paramètres passés
-	Option courante
0,1,2…	Chaque paramètre, y compris le nom de la commande (0)

Variables spéciales toujours utilisables :

Nom	Substitué par
$	Numéro de processus (PID) du shell courant
!	PID de la dernière commande d'arrière-plan terminée
?	Code de retour du dernier processus terminé

Développement



Développement arithmétique entière

Forme générale: $((expression)) où expression est une expression arithmétique entière

```
$ echo $((3+1))
4
$ echo $((3+2*6))
15
$ echo $((3-(1+1)*6))
-9
$ i=2
$ i=$((i+1))
$ echo $i
3
$ echo $((3+(1+1.5)*6))
sh: 3+(1+1.5)*6: missing `)' (error token is ".5)*6")
```

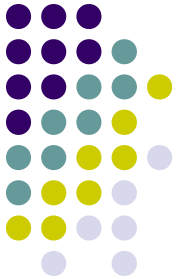
Développement

Développement des noms de fichiers

Tout mot contenant les métacaractères ‘*’, ‘?’, ‘[‘, et ‘]‘ est développé par le shell pour concorder avec un nom de fichier existant s’il en existe un.
Le shell génère alors autant de chaînes qu’il peut trouver de correspondances. S’il n’en trouve pas, le mot est laissé tel quel.
Le tilde ‘~’ seul, suivi de / ou d’un nom de login est toujours développé, et en premier.

?	Remplace un seul caractère
*	Remplace un nombre quelconque de caractères (y compris aucun)
[p]	Remplace un seul caractère parmis ceux indiqués dans p (même syntaxe que pour les expressions régulières)
~*nom*	Substitué par la variable $HOME de l’utilisateur *nom*

Développement



```
$ echo ~
/Users/xavier
$ echo $HOME
/Users/xavier
$ ls
cap1.tiff       cap2.tiff       test
$ echo cap?.tiff capi*.tiff
cap1.tiff cap2.tiff capi*.tiff
$ echo *es*
test
$ echo /???
/bin /dev /etc /lib /tmp /usr /var
$ echo /???/??
/bin/cp /bin/dd /bin/df /bin/ed /bin/ln /bin/ls /bin/mv
   /bin/ps /bin/rm /bin/sh /dev/fd /etc/rc /var/at
   /var/db /var/vm /var/yp
$ echo /???/[a-c]??
/bin/cat /bin/csh /usr/bin
```

Développement

Chaînes de caractères

- Tous les mots non délimités par des " et des ' subissent le développement de variables, arithmétique, et de noms de fichiers
- Les chaînes encadrées par des " subissent le développement de variables et le développement arithmétique, mais pas le développement des noms de fichiers
- Les chaînes de caractères encadrées par des ' ne subissent aucune développement ni aucune autre transformation
- Un caractère précédé de '\' ne subit pas de développement
- Les chaînes encadrées par des " ou par des ' sont toujours considérées comme un seul mot

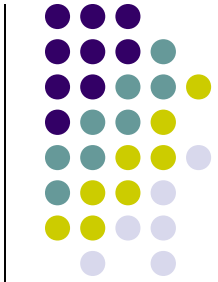
Exemples

```
$ i=1
$ ls
cap1.tiff       cap2.tiff
   test
$ echo  $i $((i+1)) cap*
1 2 cap1.tiff cap2.tiff
```

```
$ echo  "$i" "$((i+1))" "cap*"
1 2 cap*
$ echo  '$i' '$((i+1))' 'cap*'
$i $((i+1)) cap*
$ echo  \$i \$\(\(i+1\)\) cap\*
$i $((i+1)) cap*
```

Développement

Substitution de commandes



Forme générale: `$(commande)` ou `` `commande` ``

Le résultat de `commande` (sur la sortie standard) est substitué sur la ligne de commande courante. Les délimiteurs autres que l'espace (tabulations, retours chariots) sont remplacés par des espaces.

Exemple

```
$ ls
cap1.tiff       cap2.tiff       test
$ ls | wc -l
       3
$ echo Il y a $(ls | wc -l) fichiers
Il y a 3 fichiers
$
```

Redirections

A son lancement, tout processus dispose de trois fichiers ouverts par le système :

- l'entrée standard (stdin, de numéro 0)
- la sortie standard (stdout, de numéro 1)
- la sortie d'erreur standard (stderr, de numéro 2)

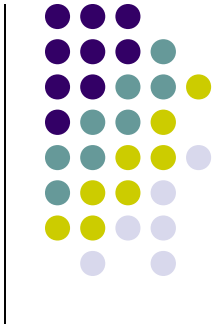
Par défaut:

- les caractères tapés par l'utilisateur dans le terminal d'exécution alimentent stdin
- le terminal est alimenté par stdout et stderr

Les redirections servent à remplacer le clavier et la sortie terminal par des fichiers.

```
$ ls
cap1.tiff       cap2.tiff       test
$ ls > /tmp/toto # redirige stdout vers /tmp/toto
$ ls -l /tmp/toto
-rw-r--r--   1 xavier  wheel  25 Feb 25 23:24 /tmp/toto
```

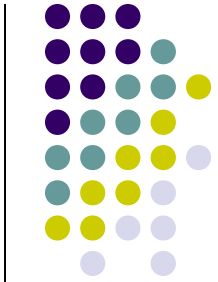
Redirections



```
$ cat /tmp/toto
cap1.tiff
cap2.tiff
test
$ wc -l < /tmp/toto # lit stdin a partir de /tmp/toto
       3
```

[n]> fichier	Redirige stdout (ou n) vers le fichier
[n]>> fichier	Ajoute stdout (ou n) vers le fichier
[n]< fichier	Alimente stdin (ou n) à partir du fichier
[n1]>&n2	Unifie stdout (ou n1) et n2 en sortie
[n1]<&n2	Unifie n2 et stdin (ou n1) en entrée
[n]>&-	Fermer la sortie standard (ou n)
[n]<> fichier	Associer le fichier en lecture/écriture à stdin (ou n)
[n]<<chaine	Lit sur stdin (ou n) jusqu'à l'apparition de la chaîne

Commandes composées



Une commande composée est :

- soit une liste,
- soit une exécution groupée (en sous-shell ou shell courant),
- soit une instruction de contrôle,
- soit une fonction

Listes

Une liste est une suite de commandes, éventuellement composées (opérateurs ‘&&’, ‘||’, ‘;’, ‘&’, ‘|’), et terminée par un ‘;’, une ‘&’ ou un retour chariot.

Soit C1 et C2 deux *commandes simples*. Alors:

- C1 && C2 : lance C1 puis C2 seulement si C1 a réussi
- C1 || C2 : lance C1 puis C2 seulement si C1 a échoué
- C1 ; C2 : lance C1 puis C2
- C1 & : lance C1 en arrière-plan
- C1 | C2 : lance C1 et C2 en parallèle et en redirigeant la sortie de C1 vers l’entrée de C2 (*pipeline)*

Commandes composées

Les pipelines

Forme générale: C1 | C2 où C1 et C2 sont 2 commandes simples

Le pipeline ('|') lance C1 et C2 en parallèle après avoir redirigé la sortie standard de C1 vers l'entrée standard de C2 - donc C2 lit son entrée directement dans la sortie de C1

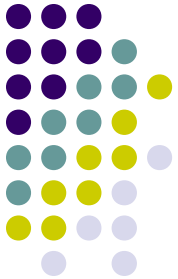
Le pipeline est composable à volonté (C1 | C2 | ... | Cn lance C1 à Cn en // avec redirection de stdout(Ci) vers stdin(Ci+1) pour tout i)

Exemple

```
$ ls -1
cap1.tiff
cap2.tiff
test
$ ls -1 | sed "s/tiff/tif/"
cap1.tif
cap2.tif
test
```

```
$ ls -1 | sed "s/tiff/tif/" |
       sort -r
test
cap2.tif
cap1.tif
$
```

Commandes composées



Exécution groupée

Formes générales:

{ *liste*; } pour une exécution dans le shell courant

(*liste*) pour une exécution dans un sous-shell

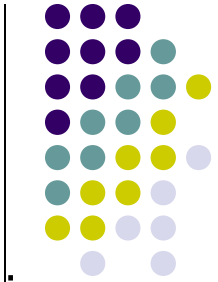
Modifie la priorité d'analyse (lexicale) de liste. Pour l'exécution dans un sous-shell, tout se passe comme si liste constituait un processus à part entière (y compris pour les entrées/sorties) et toutes les variables (ordinaires et d'environnement) du shell courant sont préservées.

Illustration

```
{ c1 && c2; } || c3
si c1 alors
        si non c2 alors c3 fsi
sinon
        c3
fsi
```

```
c1 && { c2 || c3; }
si c1 alors
        si non c2 alors
                c3
        fsi
fsi
```

Commandes composées



Instructions de contrôle

Le shell admet les instructions de contrôle: if, while, for, break, continue, case. La sémantique est la même qu'en langage C.

Syntaxe de if

```
if liste;      # si le code de retour de liste est zéro
then liste     # alors exécuter cette liste
[elif liste
then liste ] …
[ else liste ]
fi
```

Exemple

```
$ if test "$SHELL"
> then echo SHELL="$SHELL"
> else echo "Variable indefinie"
> fi
SHELL=/bin/bash
$
```

Commandes composées

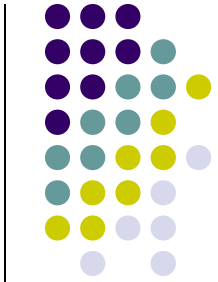
Syntaxe de while

```
while liste; # tant que code de retour de liste = 0
do liste     # exécuter cette liste
done
```

Exemple:

```
$ i=0
$ while test $i -le 2      # tant que i <= 2
> do echo i=$i ; i=$((i+1))
> done
i=0
i=1
i=2
$ echo $i
3
$
```

Commandes composées



Syntaxe de for

```
for variable in mot1 [mot2…]; do
   liste      # exécuter cette liste
done
```

Exemple:

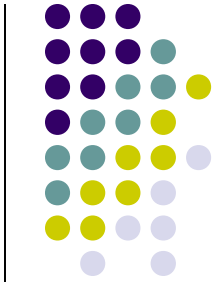
```
$ for i in un 2 trois; do
> echo $i
> done
un
2
trois
```

Syntaxe de break et continue

```
break [n]
continue [n]
```

Sort du nième niveau (ou du courant) d'itération soit pour arrêter (break) ou poursuivre (continue) . Idem langage C.

Commandes composées



Syntaxe de case

```
case mot in
motif) liste ;;
…
esac
```

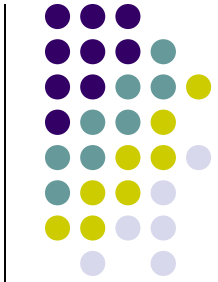
Motif peut comporter plusieurs motifs élémentaires séparés par des '|', chacun suivant les mêmes règles de développement que pour les noms de fichiers. Sémantique identique à celle du langage C, mais break n'est pas nécessaire.

Exemple:

```
$ case $OSNAME in
> Lin*) echo "motif 1";;
> tux|*nux) echo "motif 2";;
> *) echo "defaut";;
> esac
motif 1
```

```
$ case toto in
> Lin*) echo "motif 1";;
> tux|*nux) echo "motif 2";;
> *) echo "defaut";;
> esac
defaut
```

Commandes composées



Les fonctions

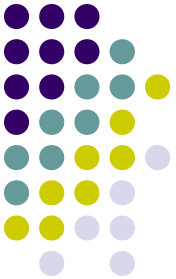
Forme générale: **`nomfonc () liste`**

où `nomfonc` est un mot et `liste` une liste (généralement entre {}).

- N'est commodément utilisable que dans un script.
- Après déclaration nomfonc est vu comme un nom de commande, qui peut recevoir des arguments
- Les arguments passés sont accessibles par $*, $@, $0, $1,…
- La fonction peut renvoyer une valeur entière via l'instruction return, récupérable par l'appelant à travers la variable $?
- Les variables qui ne sont pas déclarées `local` sont en fait celles du shell, et la fonction les altère (!)

Commandes composées

Fonctions (suite)



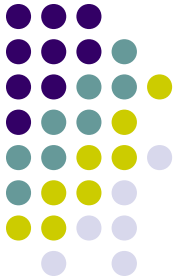
```
$ compte_fic ()
> {
>     local v
>
>     test -d "$1" && ls -l "$1" | wc -l && v=0 || v=1
>     return $v
> }
$
$ compte_fic /tmp/
       3
$ echo "Retour1=$?"
Retour1=0
$ compte_fic /inexistant/
$ echo "Retour2=$?"
Retour2=1
```

Scripts shell

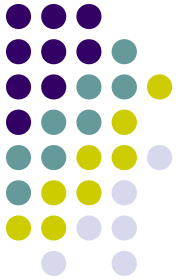
- On peut stocker un ensemble de lignes de commandes dans un fichier texte pour en faire un script shell exécutable.
- Pour pouvoir être lancé, le fichier créé doit avoir le droit d'exécution (chmod +x)
- Si la première ligne du fichier débute par #!, elle doit être suivie du chemin du shell à lancer. Si ce n'est pas le cas, le shell courant l'exécutera dans un sous-shell.
- On peut alors lancer le script en l'appelant directement par son nom

```
$ ls monscript.sh
-rw-r--r--   1 xavier  xavier  165 Feb 26 16:44
   monscript.sh
$ chmod +x monscript.sh
$ cat monscript.sh
#!/bin/sh
#
```

Scripts shell

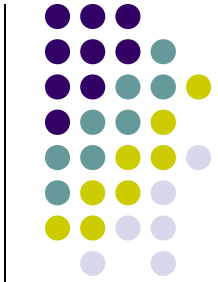


```
compte_fic ()
{
    local v

    test -d "$1" && ls -l "$1" | wc -l && v=0 || v=1
    return $v
}


compte_fic "$1"
if test $? -ge 1
then echo Erreur
fi
$ ./monscript.sh .
       7
$ ./monscript.sh /inexsitant
Erreur

$
```



Principales commandes
Principaux utilitaires

Commandes internes



Ce sont des commandes propres au shell - elles ne sont ni des scripts, ni des fichiers binaires, ni des fonctions. Commandes essentielles:

Nom	Fonction	Exemple
pwd	Donne le chemin courant	$ pwd /Users/xavier/tmp $ echo $PWD /Users/xavier/tmp

cd [dir]	Change le chemin courant à $HOME (ou dir)	$ cd /tmp
		$ pwd
		/tmp
		$ cd
		$ pwd
		/Users/xavier

Commandes internes

Nom	Fonction	Exemple
read	Lit une variable depuis l'entrée standard	$ read ligne Ceci est un test. echo $ligne Ceci est un test.
shift [n]	Décale tous les paramètres de 1 (ou n) vers la gauche	shift # $1 devient $0, $2 devient $1,…
jobs	Liste les processus lancés en arrière-plan	$ jobs [1]- Running xterm & (wd: ~/tmp) [4]+ Running xterm & (wd: ~/tmp)
fg [n]	Ramène le dernier processus d'arrière-plan (ou n) au premier-plan	$ fg 4 xterm (wd: ~/tmp)

Commandes externes

- Au sens large, comprennent tous les fichiers binaires accessibles par la variable PATH.
- La plupart des utilitaires et commandes système se trouvent dans :

/bin,/sbin,/usr/bin,/usr/sbin,/usr/local/bin,/usr/X11R6/bin

Accessibles lors du boot	Accessibles après montage de usr	Partagées sur réseau local	Clients X11

Une commande très utile : man

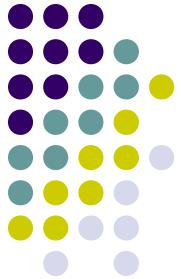
`man -a commande`

affiche les pages manuel décrivant `commande`

`man -k mot-clef`

recherche les commandes qui concordent avec le mot-clé fourni.

Commandes externes



Commandes générales

Nom	Description
`test ou [`	**Test l'existence de fichiers/répertoires, l'égalité, la supériorité, l'infériorité sur des entiers et des chaînes de caractères**
`clear`	Efface le terminal
`id`	Affiche les informations d'identité de l'utilisateur
`uname`	Nom du système d'exploitation
`who`	Liste les utilisateur connectés
`passwd`	Modification du mot de passe utilisateur
`su`	Changer d'identité utilisateur (substitute user)
`hostname`	Nom de la machine exécutant le shell

Commandes externes

Manipulation de fichiers

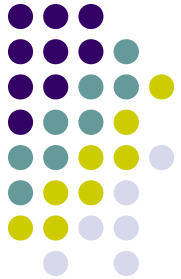
Nom	Description
`cp`	Copie de fichiers
`mv`	Renomme/déplace des fichiers
`mkdir`	Créer des répertoires
`rmdir`	Supprimer des répertoires
`ls`	Lister les fichiers
`find`	Chercher des fichiers dans l'arborescence
`chmod`	Modifier les droits d'accès
`chown`	Changer le propriétaire d'un fichiers
`mktemp`	Créer un fichier temporaire

Commandes externes

Commandes sur fichiers texte

Nom	Description
`cat`	Concatène des fichiers
`sort`	Trier les lignes d'un fichier
`cut`	Sélectionner des champs/colonnes
`paste`	Fusionner des champs entre eux
`tr`	Substituer des caractères à d'autres
`grep`	Sélectionner des lignes
`sed`	Editeur de flux (stream editor)
`vi, vim, ed`	Editeurs en mode texte
`less`	Consulter un fichier interactivement

Commandes externes



Commandes sur processus

Nom	Description
`ps`	Afficher les processus chargés
`top`	Afficher les processus chargés en temps réel
`kill`	Envoyer un signal à un processus
`renice`	Changer la priorité d'exécution d'un processus

Commandes externes

Commandes réseau

Nom	Description
`ifconfig`	Afficher les informations sur les interfaces réseau
`netstat`	Afficher l'état des ressources réseau (trafic, sockets, ports, tables de routage)
`ping`	Envoyer des paquets ICMP
`arp`	Résolution d'adresse Internet->Ethernet
`traceroute`	Afficher l'itinéraire hôte-hôte par ICMP
`nslookup`	Résolution nom d'hôte -> Adresse IP
`ypbind`	Maintenance de la liaison avec un serveur NIS