CH.6 LE LANGAGE SHELL

- 6.1 Les langages de commandes
- 6.2 Les caractères spéciaux
- 6.3 Les variables du Shell
- 6.4 Les fichiers de commandes
- 6.5 Les variables maintenues par le shell
- 6.6 Les opérations
- 6.7 Les tests
- 6.8 Les itérations
- 6.9 D'autres commandes utiles

6.1 Les langages de commandes

Les *langages de commandes* réalisent l'interface entre le système et l'utilisateur.

Ce sont tous des langages *interprétés* permettant de transmettre des commandes avec une (plus ou moins) grande souplesse :

- usage de métacaractères (*) ;
- contrôle de l'environnement ;
- présence de nombreuses commandes prédéfinies ;
- possibilité d'écrire des commandes complexes ;
- possibilité d'écrire des fichiers de commandes (script-shell) .

La plupart des problèmes courants de gestion du travail peuvent être résolus par des commandes Shell.

Les divers langages existants sont très semblables : sh, csh, ksh ou bash (utilisé ici).

Le même mot désigne l'interpréteur de commandes (shell) et le langage dans lequel les commandes sont écrites (Shell).

6.2 Les caractères spéciaux

Caractères d'abréviation : `* ?`
(Le point en début de chaîne n'est pas reconnu par * ou ?) :

```
$ ls *.c
$ ls toto.?
```

Caractères liés au lancement d'une commande :

```
; ( ) > >> 2> 2>> < & |
# \ ' ' " " ``
```

Déspécialisation des métacaractères :
En les faisant précéder de la contre-oblique :
`\|`
ou en les incluant dans des délimiteurs : `' '` ou `" "` :

```
$ echo "bonjour > toto"
bonjour > toto
```

6.3 Les variables du Shell

On peut affecter des variables. Si **a** est une variable, **$a** représente sa valeur et **echo** permet d'en prendre connaissance :

```
$ x=gh
$ echo $x
gh
$ echo $xijk

$ echo ${x}ijk
ghijk
```

Les caractères spéciaux **'** et **"** permettent une délimitation des chaînes de caractères. Dans le premier cas **$nom** n'est pas évalué, mais il l'est dans le second :


```
$ a=truc
$ x='machin$a'
$ y="machin$a"
$ echo $x
machin$a
$ echo $y
machintruc
```

Les accents graves permettent d'affecter à une variable le résultat qui serait produit par la commande placée entre accents :

```
$ a=‘whoami‘
$ echo $a
desar
```

Enfin la commande **read** permet de lire des valeurs entrées directement :

```
$ read prenom nom adresse
Jacques Desarmenien bureau 4B089
$ echo $prenom
Jacques
$ echo $nom
Desarmenien
$ echo $adresse
bureau 4B087
```

Un certain nombre de variables sont prédéfinies, suivant le contenu de fichiers du genre **.bashrc** ou **.profile** :

```
$ echo $HOME
/users/ens/desar
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/users/ens/desar/bin:
$ echo $PS1
$
$ PS1='& '
&
```


La liste des variables et de leurs valeurs peut être obtenue par la commande **set**. Une variable peut être effacée de l'environnement par la commande **unset** :

```
$ a=bonjour
$ set
HOME=/users/ens/desar
IFS=

LOGNAME=desar
PATH=/bin:/usr/bin:/usr/local/bin:/users/ens/desar/bin:
PS1=$
PS2=>
TERM=xterm
a=bonjour
$ echo $a
bonjour
$ unset a
$ echo $a

$
```

Les variables sont locales à un processus (par exemple le shell courant).

Lorsque ce processus (père) en crée un nouveau (fils), seule une partie des variables est transmise au nouveau processus. Ce sont les variables *exportables*.

La commande **`env`** permet de connaître les variables exportables.

Pour exporter une variable, il faut le spécifier avec la commande **`export`**.

Enfin, les variables créées par le fils ne sont pas transmises au père.

Exemples :


```
$ env
HOME=/users/ens/desar
LOGNAME=desar
PATH=/bin:/usr/bin:/usr/local/bin:/users/ens/desar/bin:
TERM=xterm
$ PS1='& '
& bash                      (Nouveau shell)
$ a=salut
$ echo $a
salut
$ <ctrl-d> & echo $a        (Retour à l'ancien shell)
                            (a n'existe pas)
& a=bonjour
& export a
& env
a=bonjour                   (a est exportable)
HOME=/users/ens/desar
LOGNAME=desar
PATH=/bin:/usr/bin:/usr/local/bin:/users/ens/desar/bin:
TERM=xterm
&
```

6.4 Les fichiers de commandes

On peut fabriquer un fichier de commandes (ou script-shell). On rassemble des commandes dans un fichier qu'on peut exécuter de plusieurs façons.

```
$ cat bonjour
echo Bonjour, `whoami` !
$ bash bonjour              (crée un nouveau shell)
Bonjour, desar !
$ . bonjour                 (redirige l'entrée du shell)
Bonjour, desar !
```

La façon "classique" de procéder est de rendre le fichier exécutable :

```
$ chmod u+x bonjour
```

de la sorte le nom devient une commande "de plein droit" :

```
$ bonjour                   (comme bash bonjour)
Bonjour, desar !
```


Les différences entre ces méthodes apparaît quand des variables sont créées ou des "effets de bord" engendrés.

Avec les première et dernière méthodes, un nouveau shell est créé. Tous les effets de bord sont donc perdus, alors qu'ils sont conservés avec la deuxième méthode.

Attention de ne mettre qu'une commande par ligne, ou d'utiliser des séparateurs `;`.

6.5 Les variables maintenues par le shell

En plus des variables de l'environnement, chaque processus shell maintient un certain nombre de *variables de contrôle* :

```
$ sleep 100 &
[1] 576
$ ps
PID TTY     TIME COMMAND
514 pts/0   0:00 /bin/bash
576 pts/0   0:00 sleep 100
577 pts/0   0:00 ps u
$ echo $$                  (processus en cours)
514
$ echo $!                  (dernier processus détaché)
576
$ echo $?                  (statut de la dernière commande)
0
$ ls truc
ls: truc: Aucun fichier ou répertoire de ce type
$ echo $?
1
$
```


Les fichiers de commandes peuvent comporter des paramètres, qui sont entrés sur la ligne de commande à l'exécution :

```
$ cat echange
ln -f $1 -tempo
ln -f $2 $1
ln -f tempo $2
rm _tempo
$ cat tintin
Je suis Tintin !
$ cat milou
Et moi Milou !
$ echange tintin milou
$ cat tintin
Et moi Milou !
$ cat milou
Je suis Tintin !
```

`*` est la liste des arguments ;
`#` est le nombre d'arguments ;
`0` est le nom de la commande ;
`1, ..., 9` sont les 9 premiers arguments.
S'il y a plus de 9 arguments, on peut accéder aux suivants par la commande `shift`. Mais il est souvent préférable de manipuler la liste des arguments.
Bien entendu, il est possible de définir des variables locales dans un fichier de commandes.
Attention toutefois au passage des variables !

6.6 Les opérations

La commande `expr` pemet de faire des opérations entre ses arguments.
Attention à tous les blancs entre arguments ! Se souvenir que les variables ont comme valeur des chaînes de caractères.
Penser à déspécialiser les métacaractères.

```
$ expr 4 + 5
9
$ a=5
$ expr 4$a + 5
50
$ expr $a \< 7
1
$ expr 1 \< z
1
$ expr tintin \< milou
0
```

Les opérations possibles sont :

`+ - * / %` (reste) sur des opérandes entiers ;

`< > = >= <= !` (différent de) sur des opérandes quelconques, avec comparaisons faites suivant l'ordre lexicographique entre chaînes de caractères ;

`| &` opérations booléennes.

La priorité est la priorité usuelle. On peut grouper avec des parenthèses, qu'il faut penser à déspécialiser ; l'arithmétique est assez pénible.

```
$ a=`expr 2 \* \( 3 + 4 \)`       (affectation)
$ echo $a
14
$ a=`expr $a + 1`                 (incrémentation)
$ echo $a
15
```


6.7 Les tests

Deux formes possibles : `test` *expression* ou `[` expression `]`. Attention aux espaces !

Cette commande renvoie 0 si expression est vraie et un autre entier sinon. Attention, cette convention, cohérente avec celle d'UNIX est contraire au résultat fourni par la commande `expr` !

Tests sur les chaînes de caractères :

`test -z` *chaîne* est vrai ssi *chaîne* est vide

`test -n` *chaîne* est vrai ssi *chaîne* est non vide

`test` $chaîne_1$ `=` $chaîne_2$

`test` $chaîne_1$ `!=` $chaîne_2$

Exemple :

```
if test `pwd`= $HOME
then echo le catalogue est bien le catalogue de travail
fi
```

Tests sur les nombres **entiers** :

`test` *chaîne*$_1$ **`-eq`** *chaîne*$_2$
`test` *chaîne*$_1$ **`-ne`** *chaîne*$_2$
`test` *chaîne*$_1$ **`-lt`** *chaîne*$_2$
`test` *chaîne*$_1$ **`-le`** *chaîne*$_2$
`test` *chaîne*$_1$ **`-gt`** *chaîne*$_2$
`test` *chaîne*$_1$ **`-ge`** *chaîne*$_2$

Tests sur les fichiers (entre autres...) :

`test -f` *nom* est vrai ssi *nom* est un fichier
`test -d` *nom* est vrai ssi *nom* est un répertoire
`test -r` *nom* est vrai ssi *nom* est autorisé en lecture
`test -w` *nom* est vrai ssi *nom* est autorisé en écriture
`test -x` *nom* est vrai ssi *nom* est autorisé en exécution
`test -s` *nom* est vrai ssi *nom* est de taille non nulle

Connecteurs : **`!`** (non), **`-a`** (et), **`-o`** (ou). L'ordre d'évaluation peut être forcé par des parenthèses déspécialisées **`\( \)`**.

Les tests sont rarement utilisés seuls (bien que cela soit possible).
Ils le sont en général dans une conditionnelle.

Les formes possibles sont :

`if` *commande*$_1$	si le code de retour de *commande*$_1$ vaut 0
`then` *commande*$_2$	*commande*$_2$ est exécuté,
`else` *commande*$_3$	sinon c'est *commande*$_3$.
`fi`	

Attention : les mots-clés **`then`**, **`else`** et **`fi`** doivent être en **début de ligne** ou précédés de **`;`**

La clause **`else`** est facultative.

Si on enchaîne des tests, on peut utiliser la syntaxe
`if ... then ... elif ... then ... else ... fi`

Enfin **`exit`** *n* interrompt le fichier de commandes et renvoie *n* comme code de retour (au lieu de 0).

Quelques exemples :

```
$ cat rencontre
ls $1 > /dev/null 2> /dev/null
if test $? -eq 0
then echo "Oui, $1 existe, je l\'ai rencontre."
else echo "Pour $2, repassez plus tard."
fi
$ ls
adieu affiche rencontre tintin
$ rencontre tintin
Oui, tintin existe, je l'ai rencontre.
$ rencontre Dieu
Pour Dieu, repassez plus tard.
$ cat affiche
if [ $# -ne 1 ] ; then echo "syntaxe : $0 nom_de_fichier"; exit 1; fi
if [ - f $1 ]; then cat $1
elif [ -d $1 ]; then ls $1
else echo "$1 est un fichier special ou n'existe pas."
fi
```

```
$ ls
adieu affiche rencontre tintin
$ affiche adieu tintin
syntaxe : ./affiche nom_de_fichier
$ echo $?
1
$ affiche adieu
Bonsoir !
$ echo $?
0
$ affiche .
adieu affiche rencontre tintin
$ affiche Dieu
Dieu est un fichier special ou n'existe pas.
```

L'aiguillage **`case ... esac`** a une syntaxe différente du **`switch`** de C :

`case` *chaîne* **`in`**
motif$_1$**`)`** *commande*$_1$ **`;;`**
motif$_2$**`)`** *commande*$_2$ **`;;`**
`...`
`esac`

On examine si *chaîne* satisfait au *motif*$_1$; si oui, *commande*$_1$ est exécutée et on sort de la structure ; sinon on examine *motif*$_2$ et ainsi de suite.

Pour la constitution d'un motif, on peut utiliser les caractères **`?`** et **`*`** ainsi qu'une liste de caractères **`[`***chaîne***`]`**, un intervalle **`[`***c*1-*c*2**`]`** et l'opérateur d'union **`|`**.

Par exemple :

`[0-9]|[1-9][0-9]` représente les nombres de un ou deux chiffres,
`*.[cp]` représente tous les noms de fichiers d'extension **`c`** ou **`p`**.

Comme exemple, la commande suivante appelle le bon compilateur selon l'extension du fichier. Le nom de celui-ci peut être entré en argument ou entré en ligne :

```
case $# in
0) echo -n "Fichier a compiler : "      (pas de passage à la ligne)
   read reference
   set $reference;;                     (affecte reference à 1)
esac
case $1 in
*.c) cc $1 ;;
*.p) pc $1 ;;
*.f) ff $1 ;;
*) echo "Compilation de $1 impossible" ;;
esac
```


6.8 Les itérations

L'itération bornée :

`for` *variable* **`in`** *chaîne*$_1$ *chaîne*$_2$ **`...`**
`do` *commande*
`done`

La liste de chaînes peut être explicite, la valeur d'une variable, souvent **`$*`**, le résultat d'une commande entre accents graves, le répertoire de travail avec **`*`**.

Les mots-clés **`do`** et **`done`** doivent être en **début de ligne** ou précédes de **`;`**

Exemple : Pour afficher le contenu de tous les fichiers dont l'extension est .txt :

```
for i in `ls *.txt`; do cat $i; done
```

Autres itérations :

`while ... ; do ... ; done` et `until ... ; do ... ; done`

Exemple :

```
$ cat invite
echo Repondre par oui ou non :
read reponse
while [ "$reponse" != oui -a "$reponse" != non ]
do echo Repondre par oui ou non :
read reponse
echo "Votre reponse est : $reponse."
done
$ invite
Repondre par oui ou non :
oui
Votre reponse est : oui.
$ invite
Repondre par oui ou non :
zut
Repondre par oui ou non :
non
Votre reponse est : non.
```


Création dynamique de fichiers :

```
$ cat cree
i=1
until [ $i -gt $1 ]
do echo "valeur $i" > xx$i
i=`expr $i + 1`
done
$ cree 3
$ ls xx*
xx1 xx2 xx3
$ cat xx2
valeur 2
```

La commande **`break`** fait sortir d'une itération et **`break`** *n* fait sortir de *n* niveaux d'imbrication.

La commande **`continue`** fait passer au pas suivant de l'itération. On peut aussi faire **`continue`** *n*.

Exemple :

```
$ cat options
for i in $*; do
case $arg in
   _*) opt="$opt $i";;
    *) continue;;
esac
done
echo $opt
$ options -a toto -z -gh lulu -k
-a -z -gh -k
```


6.8 D'autres commandes utiles

Faire **`man bash`**, pour avoir une liste de ce que comprend le shell.

La commande **`alias`** *nom*`="`*chaîne*`"` crée un alias *nom* pour une *chaîne*.
La commande **`alias`** permet de connaître la liste des alias actifs.

La commande **`cut`** pour sélectionner des champs.

Les *filtres* lisent sur l'entrée standard, la modifient et renvoient le résultat sur la sortie standard. L'entrée standard peut souvent être remplacée par une liste de fichiers spécifiés en paramètres.
On peut les utiliser dans des tubes.
Pour chacune de ces commandes, voir le **`man`** correspondant.

Identité **`cat`** ;
Décompte de lignes, mots et caractères **`wc`** ;
Les premières et les dernières **`head`** et **`tail`** :

`head -`*n* garde les *n* premières lignes ;
`tail -`*n* garde les *n* dernières lignes ;
`tail +`*n* garde les lignes à partir de la ligne *n* ;

Remplacement de caractères **`tr`** :

```
$ tr "[a-z]" "[A-Z]" < fichier
```

met toutes les lettres en majuscules dans le fichier.

Ordre **`sort`** ;
Édition de texte **`sed`** :

`sed s/`*motif*`/`*chaîne*`/g` fait une substitution globale de *motif* par *chaîne*. La syntaxe est celle des **expressions régulières**.

Sélection **`grep`** :

`grep` *motif* sélectionne les lignes contenant *motif*, suivant la même syntaxe.