Les shell-scripts sous *Unix*

Création d'une shellscript

Un shell est un programme utilisant des commandes *Unix*. Il se présente sous forme d'un fichier texte contenant un ensemble de commandes. A l'exécution, le fichier est interprété.

On considère le fichier de commandes suivant nommé `sauvegarde`

```
#!/bin/bash
BACKDIR=~/backup/‘date ’+%a%d%b%Y’‘
echo "Sauvegarde en cours dans $BACKDIR"
mkdir $BACKDIR
cp -v *.tex *.bib *.eps Makefile $BACKDIR
```

Quelques remarques

- La première ligne `#!/bin/bash` indique quel est le shell (ou la commande) à exécuter pour interpréter le fichier.
- On a intérêt à passer tout shell avec une permission `+x`. Cela se fait (une fois pour toutes), en tapant dans le cas du fichier `sauvegarde`:

  ```
  $ chmod +x sauvegarde
  ```

- Si le shell se trouve dans une des directories contenues dans la variable `PATH`, il sera possible de l'exécuter de partout.

Exemple d'exécution

```
$ ls
Preprint.bib       filt1-1.eps   preprint.blg  singfun-1.eps
Preprint.bix       filt2-1.eps   preprint.dvi
fancyheadings.sty  holder-1.eps  preprint.tex
fancyheadings.tex  preprint.bbl  sing1.eps
$ sauvegarde
Sauvegarde en cours dans /home/pascal/backup/Sat25Sep1999
fancyheadings.tex -> /home/pascal/backup/Sat25Sep1999/fancyheadings.tex
preprint.tex -> /home/pascal/backup/Sat25Sep1999/preprint.tex
Preprint.bib -> /home/pascal/backup/Sat25Sep1999/Preprint.bib
filt1-1.eps -> /home/pascal/backup/Sat25Sep1999/filt1-1.eps
filt2-1.eps -> /home/pascal/backup/Sat25Sep1999/filt2-1.eps
holder-1.eps -> /home/pascal/backup/Sat25Sep1999/holder-1.eps
sing1.eps -> /home/pascal/backup/Sat25Sep1999/sing1.eps
singfun-1.eps -> /home/pascal/backup/Sat25Sep1999/singfun-1.eps
cp: Makefile: No such file or directory
$
```

Les variables

Affectation: elle se fait en utilisant la syntaxe: *var* = *valeur*

Valeur: on accède à la valeur de la variable `var` en la faisant précéder de `$` (*i.e.* avec `$var`).

Exception: Il peut être nécessaire d'écrire `${var}` afin de différencier les cas suivants:

`$variable`	contenu de `variable`
`${var}iable`	contenu de `var` concaténé avec le texte `iable`

Evaluation: ou le bon usage de `"` , `'` et `` ` ``

	le contenu de `texte`
`'texte'`	n'est pas interprété
`` `texte` ``	est évalué et renvoyé
`"texte"`	n'est pas interprété sauf les parties précédées de `$ \ ` ``

```
$ a=4
$ echo $a
4
```

```
$ b="pim pam pom"
$ echo $b
pim pam pom
```

```
$ echo "${a}6 $b"
46 pim pam pom
```

```
$ c=`date`
$ echo $c
Sat Sep 25 21:09:58 CEST 1999
```

```
$ d='$a > $b'
$ echo $d
$a > $b
```

Opérations arithmétiques

En shell, toute expression ne contenant que des chiffres peut être utilisée pour du calcul arithmétique. Seules les valeurs **entières** sont autorisées.

➤ Évaluation arithmétique

`$((`*`expr`*`))` évaluation immédiate de l'expression *expr*.

```
$ a=3
$ echo $a + 4
4 + 3
$ echo $(( a + 4 ))
7
$ echo $(( a / 2 ))
1
$
```

`let` *`var=expr`* évaluation suivie d'une affectation.

```
$ let a="1+2"
$ echo $a
3
$ let b="$a + 4"
$ echo $b
7
$ let c="${a}4 + 4"
$ echo $c
38
$ let d='a+4'
$ echo $d
7
```

A noter que si le sens de " ne change pas, le sens de ' est différent.

➤ Principaux opérateurs

opérateurs binaires classiques: `+ - * / %` (reste) `**` (puissance)

affectations: `= += -= /= *= %=`

```
$ a=9
$ let "a=a+1"
$ echo $a
10
$ let "a/=2"
$ echo $a
5
$ echo $((a%2))
1
$
```

Opération sur les chaînes de caractères

➤ **Opération de manipulation de chaînes**: toutes ces fonctions laissent la chaîne de caractères *var* inchangée.

`${#var}`	Renvoie la longueur de la chaîne de caractères *var*.
`${var#exp}`	Enlève l'expression *exp* au début de la chaîne *var* (plus petit retrait possible avec *exp* si *exp* est une expression régulière).
`${var##exp}`	Idem `${var#exp}` mais en prenant le plus grand retrait possible.
`${var%exp}`	Enlève l'expression *exp* de la fin de la chaîne *var* (plus petit retrait possible avec *exp* si *exp* est une expression régulière).
`${var%%exp}`	Idem `${var%exp}` mais en prenant le plus grand retrait possible.
`${var:pos:len}`	Extrait de *var* la portion de chaîne de longueur *len* commençant à la position *pos*.
`${var:pos}`	Extrait de *var* la fin de la chaîne en commençant à la position *pos*.
`${var/exp/str}`	Retourne la chaîne *var* en remplaçant la première occurence de *exp* par *str*.
`${var//exp/str}`	Idem mais en remplaçant toutes les occurences de *exp*.

➤ **Chaînes de caractères conditionnelles**

`${var:-word}` si *var* est affecté alors renvoie $*var*
sinon renvoie *word*

`${var:=word}` si *var* n'est pas affecté alors affecte *var* à *word*
puis, renvoie $*var*

`${var:+word}` si *var* est affecté alors renvoie *word*
sinon ne renvoie rien

`${var:?word}` si *var* est affecté
alors renvoie $*var*
sinon affiche le message d'erreur *word* sur *stderr*
sort du shell-script (**exit**).

Opération sur les chaînes de caractères : Exemples

```
$ z="123456789012345"
$ echo ${#z}
15
$ echo ${z#123}
456789012345
$ echo ${z#*2}
3456789012345
$ echo ${z##*2}
345
$ echo ${z%345}
123456789012
$ echo ${z%3*}
123456789012
$ echo ${z%%3*}
12
$ echo ${z:6}
789012345
$ echo ${z:6:4}
7890
$ echo ${z/2/a}
1a3456789012345
$ echo ${z//2/a}
1a345678901a345
$
```

```
$ A="abcd"
$ B=""
$ echo ${A:-1234}
abcd
$ echo ${B:-1234}
1234
$ echo ${A:+1234}
1234
$ echo ${B:+1234}

$ echo ${A:=1234}
abcd
$ echo ${B:=1234}
1234
$ echo $B
1234
```

```
$ C=""
$ D="/home/kroc"
$ pwd
/
$ cd ${C:?Invalid directory}
bash: C: Invalid directory
$ cd ${D:?Invalid directory}
$ pwd
/home/kroc
$
$
```

Variables prédéfinies dans un shell

Un certain nombre de variables non modifiables sont prédéfinies par défaut dans un shell.

➤ **Les variables générales**: parmi celle-ci, on peut citer:

`$$`	`pid` du shell.
`$PPID`	`pid` du processus père.
`$?`	valeur de sortie de la commande précédente (0 si terminée avec succés).
`$SECONDS`	le nombre de secondes écoulés depuis le lancement du script.
`$RANDOM`	génère un nombre aléatoire entre 0 e t 32767 à chaque appel.

➤ **Variables pour le passage d'arguments**

Il est possible de passer des paramètres à un shell, et de les exploiter. Pour cela, on utilise les variables spéciales suivantes:

- `$#` : renvoie le nombre d'arguments.
- `$@` ou `$*` : l'ensemble des arguments.
- `$0` : nom du shell (ou de la fonction).
- `$`n : $n^{\text{ième}}$ argument (exemple: `$3` est le $3^{\text{ième}}$ argument).

```
$ cat testarg
echo "shellscript : $0"
echo "nb arguments: $#"
echo "arguments   : $*"
echo "1er argument: $1"
echo "2nd argument: $2"
```

```
$ ./testarg oui 3
shellscript : ./testarg
nb arguments: 2
arguments   : oui 3
1er argument: oui
2nd argument: 3
```

Par ailleurs, deux fonctions servent à manipuler les arguments:

- `set` : réaffecte les arguments.
- `shift` p : décale les arguments vers la gauche (l'argument n devient l'argument $n-p$, l'argument 0 ne change pas, les arguments de 1 à p sont perdus). Si p n'est pas spécifié, il est égal à 1.

```
$ set ‘date‘
$ echo $*
Sat Sep 25 22:49:42 CEST 1999
$ echo $6
1999
```

```
$ shift 3
$ echo $*
22:49:42 CEST 1999
$ echo $3
1999
```

Composition des commandes

- Séparateurs et groupement de commandes

  Les commandes doivent toujours être séparées soit par une fin de ligne (retour chariot), soit par le caractère **;**

  Un ensemble de commandes peut être groupé en utilisant les accolades: { *commandes* **;** }

```
$ ls ; wc -l toto.c
toto.c  toto.o
    264 toto.c
$
```

```
$ { ls ; wc -l toto.c }
toto.c  toto.o
    264 toto.c
$
```

- Opérateurs `&&` et `||`

`com1 && com2`
: exécute la commande `com1`. Puis, si la commande `com1` a un statut de sortie de 0 (`exit 0` est en général équivalent à une sortie sans erreur), alors la commande `com2` est exécutée.

`com1 || com2`
: exécute la commande `com1`. Puis, si la commande `com1` a un statut de sortie différent de 0 (en général équivalent à une sortie sur erreur), alors la commande `com2` est exécutée.

```
$ ls
toto.c  toto.o
$ ls *.o && rm *.o
toto.o
$ ls
toto.c
```

```
$ wc -l toto || echo > toto
wc: toto: No such file or directory
$ wc -l toto || echo > toto
       1       0       1 toto
$
```

- Sous-shell

  Les commandes passées entre parenthèses sont exécutées comme un sous-shell.

```
$ a=3; ( echo -n $a; a=2; echo -n $a ); echo $a
323
```

Fonctions

Il est possible dans un shell de définir des fonctions:

- **syntaxe**: la syntaxe d'une fonction est la suivante:

```
function NOM_FONCTION
{
    CORPS_FONCTION
}
```

- **arguments**: les arguments ne sont pas spécifiés, mais ceux-ci peuvent être traités en utilisant `$#`, `$*` et $\$n$.
- **portée de la fonction**: elle n'est connue que dans le shell qui la crée (mais ni dans le père, ni dans le fils).
- **portée des variables**
  - `VAR=`*valeur* est une variable globale.
  - `typeset VAR=`*valeur* est une variable locale.

```
$ cat testfun1
function test1
{
echo "Nb arg: $#"
echo "Arg 1 : $1"
echo "Shell : $0"
a=$2
typeset b=$3
echo "in  : a=$a  b=$b"
}

a=1
b=2
echo "init: a=$a  b=$b"
test1 pim pam pom
echo "out : a=$a  b=$b"
```

```
$ ./testfun1
init: a=1  b=2
Nb arg: 3
Arg 1 : pim
Shell : ./testfun1
in  : a=pam  b=pom
out : a=pam  b=2
```

Entrée/Sortie

- `echo` : affichage sur la sortie standard.

  options:

  `-n` pas de passage automatique à la ligne.

  `-e` active l'interprétation des codes suivants (entre guillemets).

`\a`	bip!
`\b`	backspace
`\c`	comme `-n`
`\f`	saut de page
`\n`	saut de ligne

`\r`	retour chariot
`\t`	tabulation
`\\`	caractère `\`
`\nnn`	caractère dont le code octal est *nnn*

```
$ echo coucou
coucou
$ echo $HOME
/home/pascal
```

```
$ echo -n 1 ; echo 2
12
$ echo -e "1\t2"
1       2
```

- `read` : lecture à partir de l'entrée standard.
  - si aucun nom de variable n'est spécifié, la valeur lue est mise dans la variable `REPLY`.
  - si plusieurs noms de variables sont spécifiés, l'expression est parsée entre les différentes variables. La dernière variable contient éventuellement la fin complète de l'entrée.
  - `read` renvoie toujours `0` sauf sur le caractère `EOF` où `1` est renvoyé. Ceci permet d'utiliser cette fonction pour lire des fichiers.

```
$ read x
pim pam pom
$ echo $x
pim pam pom
$ read
pom pam pim
$ echo $REPLY
pom pam pim
```

```
$ read x y
pim pam pom
$ echo $x
pim
$ echo $y
pam pom
```

Les tests

Il existe en shell trois principales catégories de test:

- Les tests sur les chaînes de caractères.
- Les tests sur les entiers.
- Les tests sur les fichiers.

Afin d'éviter les confusions ou les problèmes d'écriture, nous utiliserons des écritures uniques pour chacune des catégories.

Contrairement aux langages de programmation classique, un test renvoie **0 si le test est VRAI**, et une valeur différente de 0 si le test est faux. Ceci est lié au fait que, sous Unix, un programme se terminant avec succés[1] renvoie la valeur 0.

L'écriture canonique du test d'une expression *expr* s'écrit:

`test` *expr* **ou** `[` *expr* `]`

Dans tous les cas, on pensera à **toujours** placer les variables entre guillemets afin d'éviter les problèmes de parsing, et à mettre des espaces **avant et après** les doubles crochets ou parenthèses.

• Comparaison de chaînes de caractères

`[` *str1* `==` *str2* `]`	vrai si *str1* est égale à *str2*
`[` *str1* `!=` *str2* `]`	vrai si *str1* est différente de *str2*
`[` *str1* `\<` *str2* `]`	vrai si *str1* est inférieure à *str2* (ordre alpha.)
`[` *str1* `\>` *str2* `]`	vrai si *str1* est supérieure à *str2* (ordre alpha.)
`[ -z` *str* `]`	vrai si *str* est nulle.
`[ -n` *str* `]`	vrai si *str* est non nulle.

• Comparaison d'entiers

`[` *num1* `-eq` *num2* `]`	vrai si *num1* est égal à *num2*
`[` *num1* `-ne` *num2* `]`	vrai si *num1* est différent de *num2*
`[` *num1* `-lt` *num2* `]`	vrai si *num1* est inférieur à *num2*
`[` *num1* `-le` *num2* `]`	vrai si *num1* est inférieur ou égal à *num2*
`[` *num1* `-gt` *num2* `]`	vrai si *num1* est supérieur à *num2*
`[` *num1* `-ge` *num2* `]`	vrai si *num1* est supérieur ou égal à *num2*

[1] Une conséquence importante est qu'il est possible de tester directement une commande pour savoir si elle s'est bien déroulée.

Les tests (2)

- Tests sur les fichiers

  `test -a` *file* vrai si *file* existe
  `test -f` *file* vrai si *file* est un fichier
  `test -s` *file* vrai si *file* est un fichier de taille non nulle
  `test -d` *file* vrai si *file* est un répertoire
  `test -L` *file* vrai si *file* est un lien symbolique
  `test -r` *file* vrai si l'utilisateur a le droit `r` sur *file*
  `test -w` *file* vrai si l'utilisateur a le droit `w` sur *file*
  `test -x` *file* vrai si l'utilisateur a le droit `x` sur *file*

  Attention, ces options peuvent être dépendantes de la machine (par exemple, le test d'un lien symbolique sur Solaris).

- Combinaisons logiques

	forme avec [ ]	forme avec `test`
expr1 **ET** *expr2*	`[ expr1 && expr2 ]`	`test expr1 -a expr2`
expr1 **OU** *expr2*	`[ expr1 || expr2 ]`	`test expr1 -o expr2`

  La négation s'exprime avec le symbole `!`.

- Exemples

```
$ A=8
$ [ "$A" -gt 5 ] && echo "vrai"
vrai
$ [ ! "$A" -ge 3 ] && echo "vrai"
vrai
```

```
$ B="ab"
$ [ "$B" != "abc" ] && echo "vrai"
vrai
$ [ -n "$B" ] && echo "vrai"
vrai
```

```
$ ls -l
-rw-r--r--   1 Pascal   users    1 Sep  7 12:07 toto.c
$ test -f toto.c && echo "vrai"
vrai
$ test ! -x toto.c && echo vrai
vrai
$ ! test -x toto.c && echo vrai
vrai
```

```
$ [ "aa" \< "$B" ] && [ "$B" \< "ac" ] && echo "vrai"
vrai
$ [ 3 -le "$A" ]  && [ "$A" -ge 10 ] && echo "vrai"
vrai
$ test -f toto.c -o -r toto.c  && echo "vrai"
vrai
```

Structure de contrôle conditionnelle : `if`

Il y a 3 structures de base:

```
if tst
then
   cmd1
else
   cmd2
fi
```

```
if tst
then
   cmd
fi
```

```
if tst1
then
   cmd1
elif tst2
  then
     cmd2
  else
     cmd3
fi
```

```
$ cat if1
echo -n "$1 est un "
if ! (test -a $1)
then echo "trouvable"
elif test -L $1
then echo "lien symbolique"
elif test -f $1
then echo "fichier"
elif test -d $1
then echo "repertoire"
else echo "connu"
fi
$ ls -l
total 2
drwxr-xr-x   2 pascal   users   1024 Sep 26 00:24 tata
lrwxrwxrwx   1 pascal   users      4 Sep 26 00:24 titi -> toto
-rw-r--r--   1 pascal   users     40 Sep 26 00:23 toto
$ if1 tata
tata est un repertoire
$ if1 titi
titi est un lien symbolique
$ if1 toto
toto est un fichier
$ if1 tutu
tutu est un trouvable
```

```
$ if cd; then echo "succes"; else echo "echec"; fi; pwd
succes
/home/pascal
$ if cd /bachibouzouk; then echo "succes"; else echo "echec"; fi; pwd
bash: cd: /bachibouzouk: No such file or directory
echec
/home/pascal
```

Structure de contrôle conditionnelle : `case`

Analogue au **case** en *C*

```
case mot in
cas1) cmd1;;
cas2) cmd2;;
⋮
casn) cmdn;;
esac
```

avec

mot contenu d'une variable, expression arithmétique ou logique, résultat de l'évaluation d'une commande.

cas_i expression régulière constante de type nom de fichier. Il est possible de donner plusieurs expressions régulières en les séparant par le caractère `|` (ou). Le cas `default` du *C* est obtenu en prenant `*` pour valeur cas_n.

cmd_i commande ou suite de commandes à exécuter si le résultat de *mot* correspond à cas_i .

```
$ cat dus
case $# in
0) dir=‘pwd‘;;
1) dir=$1;;
*) echo "Syntaxe: dus [repertoire]"
   exit;;
esac
du -s $dir
```

```
$ dus
143     /home/pascal/Universite/StageUnix
$ dus /home/pascal
557392  /home/pascal
$ dus /home/pascal /mnt/cdrom
Syntaxe: dus [repertoire]
```

Structures de contrôle : les boucles

- `for`

  syntaxe

  `for` *var* `in` *liste*
  `do`
  commandes
  `done`

  exemples de listes valides

`1 2 3 4`	chiffres de 1 à 4
`*.c`	fichiers `C` du répertoire courant
`$*`	arguments
`‘users‘`	utilisateurs

- `while` ou `until`

  syntaxe

`while` COMMAND	`until` COMMAND
`do`	`do`
commandes	*commandes*
`done`	`done`

  condition d'arret

  COMMAND est une commande renvoyant 0 pour vrai (voir la fonction `test`).

  exemples

```
$ cat listarg
for u in $*
do
   echo $u
done
$ listarg pim pam pom
pim
pam
pom
```

```
$ cat boucle
i=0
while test $i -lt 3
do
   echo $i
   let i=i+1
done
$ boucle
0
1
2
```

Contrôle d'exécution

- `exit` n
  - elle permet une sortie immédiate du shell, avec l'erreur n.
  - si n n'est pas fourni, $n = 0$ (sortie normale).
- `break`

  fonction de contrôle de boucle permettant une sortie immédiate de la boucle en cours.
- `continue`

  fonction de contrôle de boucle permettant un passage immédiat à l'itération suivante.

```
$ cat testcont
i=0
while test $i -lt 8
do
   let i=i+1
   if test $((i%2)) -eq 0
   then continue
   fi
   echo $i
done
$ testcont
1
3
5
7
```

```
$ cat testbreak
i=0
while test $i -lt 8
do
   let i=i+1
   if test $((i%3)) -eq 0
   then break
   fi
   echo $i
done
echo "Fin"
$ testbreak
1
2
Fin
```

Redirections et boucles

Il est possible de conjuguer les redirections < et > pour effectuer des opérations sur des fichiers.

```
$ cat testfor1
for u in 1 2 3
do
   echo $u > flist1
done
$ testfor1
$ cat flist1
3
```

```
$ cat testfor2
for u in 1 2 3
do
   echo $u
done > flist2
$ testfor2
$ cat flist2
1
2
3
```

```
$ cat litfic
while read u
do
   echo $u
done < flist2
$ litfic
1
2
3
```

```
$ cat litficbad
while read u < flist2
do
   echo $u
done
$ litficbad
1
1
(boucle infinie)
```

Notes

- Attention à la position des redirections dans la boucle.
- Pour forcer le `read` à utiliser l'entrée du clavier, utiliser:

```
read var < /dev/tty
```

Choix interactif: `select`

Cette fonction permet d'effectuer un choix interactif parmi une liste proposée.

Syntaxe

`select` *var* `in` *liste*
`do`
commandes
`done`

Fonctionnement

1. La liste des choix est affichée sur `stderr`.
2. La variable *var* est affectée à la valeur choisie seulement si le choix est valide (non affectée sinon).
3. Puis, *commandes* est exécuté.
4. Le choix est reproposé (retour en 1) jusqu'à ce que l'instruction **break** soit rencontrée dans *commandes*.

Option

La variable `PS3` permet de régler la question posée (voir exemples).

Exemples

```
$ cat select1
#!/bin/bash
select u in A B C
do
[ -n "$u" ] && break;
done
echo "Le choix est: $u"

$
$ ./select1
1) A
2) B
3) C
#? zorglub
#? 2
Le choix est: B
$
$
```

```
$ cat ./select2
#!/bin/bash
PS3="Votre choix:"
select v in "Ou suis-je?" Quitter
do
        case $v in
        "Ou suis-je?") pwd;;
        "Quitter") break;;
        *) echo "Taper 1 ou 2"
        esac
done
$
$ ./select2
1) Ou suis-je?
2) Quitter
Votre choix:1
/home/pascal
Votre choix:3
Taper 1 ou 2
Votre choix:2
$
```

L'éditeur `emacs`

Emacs est un éditeur extrêment puissant et complètement reprogrammable. Les séquences de touches données sont les valeurs par défaut. On notera `^x` pour la séquence [ctrl]-[x], et `m-a` pour [Echap]-[a].

1. **Déplacement**

`^a`/`^e`	début/fin de ligne	`^v`/`m-v`	page suiv./préc.
`m-<`/`m->`	début/fin du fichier	`^x-<`/`^x->`	scroll gauche/droite

2. **Destruction**

`^d`/[del]	suppression du caractère préc/suiv
`^k`	destruction jusqu'en fin de ligne

3. **Répetition** La séquence `m-`*n command* permet de répéter *n* fois la commande *command*.

4. **Copier/Couper/Coller** Ces commandes utilisent la définition de région, et s'appliquent du début de la région à la position du curseur.

`^`[espace]	début de région	`m-w`	copier
`^w`	couper	`^y`	coller

5. **Rechercher** `^s`/`^r` recherche avant/arrière
   Remplacer `m-%` remplacer

6. **Divers**

`^g`	annuler requête
`^x-l`	position du curseur
`^x-u`	undo

Fenêtres

`^x-2`/`^x-5`	division horizontale/verticale
`^x-o`	changement de fenêtre
`^x-0`	suppression de la fenêtre

7. **Gestion des fichiers**

`^x-^f`	ouverture d'un fichier	`^x-^w`	sauvegarde sous un autre nom
`^x-^s`	sauvegarde du fichier	`^x-^c`	sortie

8. **Commandes activées à partir de `m-x`**

`m-x apropos`	aide sur les commandes à partir d'un mot-clé
`m-x help`	aide générale
`m-x goto-line`	va à la ligne *n*
`m-x replace-string`	remplacer une chaîne (non interactif)
`m-x global-set-key`	définit l'action d'une touche

L'éditeur `vi`

Il est malgré tout nécessaire de connaître un peu cet éditeur ancien et assez peu intuitif, car il est petit et rapide; il existe sur tous les systèmes *Unix*; il fonctionne dans une console texte; lorsqu'il y a peu de modifications à faire, `vi` est souvent le bon choix.

1. **Déplacement du curseur/position**

`h`/`l`	caractère préc./suiv.	`+`/`-`	début de ligne préc./suiv.
`b`/`w`	mot préc./suiv.	`ctrl`-`f`	page précédente
`0`/`$`	début/fin de ligne	`ctrl`-`b`	page suivante
`j`/`k`	ligne préc./suiv.	*n* `G`	va à la $n^{\text{ième}}$ ligne du fichier

2. **Destruction** (et placement dans le buffer)

`d` `d` ou `D`	destruction de la ligne courante
`d` `0`/`d` `$`	destruction du curseur jusqu'au début/fin de ligne
`x`/`X`	destruction du caractère sous le/à gauche du curseur.

3. **Répétition** on peut répéter la majorité des commandes en tapant le nombre de fois à répéter la commande avant de la taper. Par exemple, `12D` efface 12 lignes.

4. **Mode insertion/remplacement**

`i`/`a`	avant/après le curseur	`o`/`O`	avant/après la ligne courante
`I`/`A`	en début/fin de ligne	`Echap`	sortie du mode insertion

`r` *c*	remplace le caractère courant par *c*
`J`	joint la ligne courante et la suivante
`u`/`U`	annulation (undo)/restaure la ligne

5. **Copier/Coller**

`y` *pos*	copie du curseur jusqu'à *pos* (`y` `y` pour la ligne complète)
`p`/`P`	insertion du buffer avant/aprés le curseur

6. **Recherche**

`/`/`?` *chaîne*	recherche avant/arrière de *chaîne*
`n`/`N`	occurence suiv./préc.

7. **Sauvegarde**

`:` `w`	sauvegarde du fichier
`:` `w` *file*	sauvegarde sous le nouveau nom *file*
`:` `q`	sortie
`:` `w` `q`	sauvegarde le fichier et sort