Année 2001−2002

Pense−bête

Systèmes Informatiques

1ère année Matméca

Université de Bordeaux1

mis en page par Philippe Depouilly du MPB (http://www.math.u−bordeaux.fr/Math_Pures), d'après les documents électroniques de :

J.D. Durou et Ph. Joly de l'IRIT (http://www.irit.fr),

Philippe Dax de l'ENST (http://www−inf.enst.fr) et

Christian Labesse de l'A2X (http://www.math.u−bordeaux.fr/A2X)

Sommaire :

Rappels des commandes UNIX

Introduction

Un "shell" est un interpréteur de commandes du système UNIX. Il existe plusieurs versions de cet interpréteur : le "shell de Bourne" (ou "Bourne shell"), le "Korn shell", le "C_shell",...
On se placera dans le cadre du **shell de Bourne**.
Le format d'une commande sous UNIX suit le modèle suivant :

```
nom_de_la_commande -options arguments
```

Les options et arguments forment ce que l'on appelle les "paramètres" de la commande.

Les métacaractères du shell

Les métacaractères du shell permettent :

- de construire des chaînes de caractères génériques :
  - ***** désigne une chaîne de caractères quelconque ;
  - **?** désigne un caractère quelconque ;
  - **[...]** désigne les caractères entre crochets, définis par énumération ou par un intervalle.

- Exemples :

  [Aa] désigne les caractères **A** ou **a** ;
  [0-9a-zA-Z] désigne un caractère alphanumérique quelconque.

  Remarque :

  [!0-9] désigne l'ensemble des caractères sauf les chiffres.

- de modifier l'interprétation d'une commande :

  - **;** sépare deux commandes sur une même ligne ;
  - **'** délimite une chaîne de caractères contenant des espaces (à l'intérieur, tous les métacaractères perdent leur signification) ;
  - **"** délimite une chaîne de caractères contenant des espaces (à l'intérieur, tous les métacaractères perdent leur signification, à l'exception des métacaractères **`** et **$**) ;
  - **`** "capture" la sortie standard pour former un nouvel argument ou une nouvelle commande ;
  - **** annihile la signification du métacaractère qui suit ;
  - **{** et **}** permettent de regrouper un ensemble de commandes et de les exécuter dans le "shell courant" ;
  - **(** et **)** permettent de regrouper un ensemble de commandes et de les exécuter dans un "shell fils".

Les expressions régulières

On appelle "expression régulière" une chaîne de caractères composée de caractères normaux et de caractères spéciaux, appelés "métacaractères des expressions régulières" (à ne pas confondre avec les métacaractères du shell décrits plus haut !). Comme toutes les chaînes de caractères, on peut écrire une expression régulière entre apostrophes, entre guillemets ou seule, mais on évitera cette dernière possibilité, et on conseille même d'utiliser la notation entre apostrophes. Les principaux métacaractères des expressions régulières sont les suivants :

- **.** désigne un caractère quelconque ;
- ***** désigne une répétition quelconque du caractère qui précède ;
- **^** désigne un début de ligne ;
- **$** désigne une fin de ligne ;
- **** protège le caractère suivant, qu'il soit caractère normal ou métacaractère (des expressions régulières) ;
- **{**nbre**}** désigne la répétition du caractère précédent, qu'il soit caractère normal ou métacaractère (des expressions régulières) ;
- **[...]** désigne les caractères entre crochets, définis par énumération ou par un intervalle (**^** situé après le **[** désigne le complément des caractères entre crochets).

Exemples :

- **'.*'** désigne une ligne quelconque ;
- **'^$'** désigne une ligne vide ;

- **'^début'** désigne une ligne commençant par **début**
- **"'pwd'$"** désigne une ligne se terminant par le nom du répertoire de travail ;
- **'*.\\'** désigne une ligne contenant un caractère *****, suivi d'un caractère quelconque, suivi d'un caractère ****
- **'[A-Z][a-z]{9}'** désigne une ligne contenant au moins dix lettres successives, la première étant une majuscule, et les neuf autres étant des minuscules ;
- **'^[A-Z]'** désigne une ligne commençant par une lettre majuscule.

Redirections de l'entrée standard et de la sortie standard

Il est possible, sous UNIX, de rediriger l'entrée standard ou la sortie standard d'une commande :

- **<** : redirection de l'entrée standard ;

  Exemple :

  ```
  grep chaine < fichier1
  ```

- **<<** : redirection temporaire de l'entrée standard ;

  Exemple :

  ```
  [nom_commande] << [séparateur]
  ligne1
  ligne2
  ...
  [séparateur]
  ```

  La commande reçoit en entrée le texte situé entre les deux occurrences du **séparateur**.

- **>** : redirection de la sortie standard ;

  Exemple :

  ```
  ls > fichier1
  ```

  Si **fichier1** n'existe pas, il est créé ; sinon, son contenu est écrasé ;

- **>>** : redirection, avec concaténation, de la sortie standard ;

  Exemple :

  ```
  ls >> fichier1
  ```

  Si **fichier1** n'existe pas, il est créé ; le résultat de l'appel de la commande **ls** est ajouté

à la fin du fichier **`fichier1`** ;

- **`|`** : redirection de la sortie standard d'une commande vers l'entrée standard d'une autre commande ;

  Exemple :

  ```
  ls | grep *.c
  ```

- **`‘‘`** : "capture" de la sortie standard (*i.e.* l'information destinée à la sortie standard est conservée sur la ligne de commande).

Remarque :

Une redirection en entrée est toujours suivie d'un nom de fichier, mais jamais du résultat d'une commande.

Redirections de la sortie standard des erreurs

Il est possible de rediriger la sortie standard des erreurs sur un fichier, de l'une des deux manières suivantes :

```
[commande] 2> [nom_fichier]
```

ou :

```
[commande] 2>> [nom_fichier]
```

Les "scripts"

Un "script" est un ensemble de commandes UNIX rassemblées dans un fichier. Pour lancer l'exécution d'un script contenu dans le fichier **`nom_script`**, on peut :

- soit taper la commande **`sh [nom_script] [paramètres]`**
- soit le rendre exécutable (grâce à l'utilisation de la commande **`chmod`**), puis l'appeler directement, de la manière suivante :

```
tgv% chmod u+x [nom_script]
tgv% [nom_script] [paramètres]
```

Les paramètres positionnels

$0, **$1, $2**, **$3**, **$4**, **$5**, **$6**, **$7**, **$8** et **$9** désignent les paramètres positionnels, à l'appel d'un script. **$0** désigne le nom du script. On désigne également par **$#** le nombre total de paramètres positionnels (**$0** non compris), et par **$*** la liste des paramètres positionnels. Enfin, **$?** contient le code de retour de la dernière commande exécutée.

Exemple :

On écrit, dans le fichier **exemple.sh**, les commandes suivantes :

exemple.sh

```
echo "exécution de la commande $0"
echo "contenu du fichier 1"
cat $1
echo "contenu du fichier 2"
cat $2
```

On rend ce script exécutable en tapant, dans le répertoire de travail :

```
tgv% chmod u+x exemple.sh
```

Puis on l'exécute, en tapant :

```
tgv% exemple.sh fich1 fich2
```

Les opérateurs && et ||

- Dans la commande :

  liste_commandes_1 && liste_commandes_2

  l'opérateur **&&** est un séparateur de commandes provoquant l'exécution de la liste de commandes **liste_commandes_2** si la liste de commandes **liste_commandes_1** a pour code de retour **0**

- Dans la commande :

  liste_commandes_1 || liste_commandes_2

  l'opérateur **||** est un séparateur de commandes provoquant l'exécution de la liste de commandes **liste_commandes_2** si la liste de commandes **liste_commandes_1** a un code de retour différent de **0**

Dans un "script", il est possible de structurer les commandes en effectuant des appels conditionnels.

Syntaxe :

```
if condition1
then
liste_commandes_1
elif condition2
then
liste_commandes_2
else
liste_commandes_3
fi
```

Remarques :

- 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.
- En shell : la valeur **0** est associée au booléen "vrai" et toute autre valeur correspond au booléen "faux" (convention inverse de celle du langage C).

Exemple :

```
if [ $# −eq 1 ]
then
echo "Le répertoire $1 a le contenu suivant : ‘ls $1‘"
else
echo ’Mauvais nombre de paramètres’
fi
```

Ce "script" affiche le contenu du répertoire donné en paramètre. S'il n'y a pas exactement un paramètre, un message est affiché.

La structure de contrôle d'aiguillage à choix multiple

Cette structure de contrôle permet d'effectuer un branchement conditionnel sur une séquence de commandes, en fonction de la valeur d'une variable.

Syntaxe :

```
case variable in
cas1)
liste_commandes_1
;;
cas2)
liste_commandes_2
;;
cas3|cas4)
liste_commandes_3
;;
*)
liste_commandes_4
;;
esac
```

Dans ce modèle :

- **`variable`** est une variable ou un paramètre du shell courant ;
- **`cas1`**, **`cas2`**, **`cas3`** et **`cas4`** sont des chaînes de caractères, utilisant éventuellement les métacaractères du shell, ainsi que le caractère **`|`**, qui signifie "ou" ;
- le caractère **`*`**, en tant que métacaractère du shell, signifie "n'importe quelle chaîne de caractères" (c'est donc l'équivalent du *default* du langage C).

Exemple :

```
case $# in
0)
set variable=‘pwd‘
;;
1)
set variable=$1
;;
*)
echo "Erreur de syntaxe" ; exit
;;
esac
```

Dans ce "script", la variable **`variable`** reçoit comme valeur le répertoire courant (si le nombre de paramètres est nul) ou la valeur du paramètre positionnel **`$1`** (s'il n'y a qu'un paramètre). S'il y a plus d'un paramètre, un message est affiché.

Remarque :

La commande **`break`** (équivalente à l'instruction *break* du langage C) existe, mais elle n'est pas nécessaire dans la structure de contrôle d'aiguillage à choix multiple, car le double point−virgule **`;;`** est équivalent à cette commande.

La structure de contrôle "boucle For"

Cette structure de contrôle permet de lancer une séquence de commandes en boucle.

Syntaxe :

```
for variable in liste_de_cas
do
liste_commandes
done
```

Dans ce modèle, **`liste_de_cas`** est une liste de chaînes de caractères, utilisant éventuellement les métacaractères du shell.

Exemple :

```
for fichier in ‘ls *.c‘
do
chmod go−r $fichier
done
```

Cette séquence change les droits d'accès de tous les fichiers du répertoire courant comprenant l'extension **`.c`**

Remarques :

− Dans les structures de contrôle de boucles, la commande **`break`** peut être utilisée.

− La syntaxe suivante :

```
for i
do
...
```

est équivalente à :

```
for i in $*
do
...
```

La structure de contrôle "boucle While"

Cette structure de contrôle permet d'exécuter une séquence de commandes en boucle tant qu'une condition est vérifiée.

Syntaxe :

```
while condition
do
liste_commandes
done
```

La structure de contrôle "boucle Until"

Cette structure de contrôle permet de boucler sur une séquence de commandes jusqu'à ce qu'une condition devienne vraie.

Syntaxe :

```
until condition
do
liste_commandes
done
```

Les variables du shell

Tout shell permet de gérer des **variables non typées**. Le nom d'une variable peut comporter des lettres, des chiffres et le caractère _ et le premier caractère ne peut pas être un chiffre.
Avec le shell de Bourne, l'initialisation d'une variable s'effectue à l'aide de la commande d'affectation, pour laquelle l'opérateur **=** est utilisé, comme en langage C :

x=7

Attention : il ne doit pas y avoir d'espace ni avant ni après l'opérateur **=**
Une variable peut recevoir comme valeur :

- une chaîne de caractères ;
- un entier ;
- le résultat d'une commande ;
- la valeur d'un paramètre positionnel ;
- la valeur d'une variable ;
- toute "combinaison" des cas précédents.

En fait, la seule "combinaison" que l'on peut faire directement est la concaténation de plusieurs valeurs. Toute autre opération doit être faite à l'aide de la commande **expr**, qui sera vue plus loin.

Pour faire référence à la valeur d'une variable du shell, il faut faire précéder son nom du métacaractère **$**

Remarques :

− Il existe, dans tout shell, un certain nombre de variables prédéfinies. Dans un shell de Bourne, sur **tgv**, il y en a 22, dont les noms ne comportent que des lettres majuscules, des chiffres et le caractère _

− Il ne faut pas confondre les variables d'un shell avec ses paramètres positionnels. Pour modifier la valeur d'un paramètre positionnel, il faut utiliser la commande **set**.

Exemple :

```
x=7
y='pwd'
x=$y$x
echo $x
```

Si le répertoire de travail est **/home/mod4g0** alors la commande **echo** produira l'affichage suivant :

```
/home/mod4g07
```

Lancement d'un "shell fils"

Une séquence de commandes placée entre parenthèses est exécutée par un "sous−shell" ou "shell fils". Tout déplacement dans l'arborescence des répertoires et toute modification de la valeur d'une variable dans un shell fils ne sont plus effectifs dès le retour dans le shell père.

Exemple :

```
$ cd
$ pwd
/home/mod4g0
$ a=1
$ (cd /usr/include; a=2; echo $a; pwd)
2
/usr/include
$ pwd
/home/mod4g0
$ echo $a
1
```

Remarques :

− L'entrée et la sortie standard d'un shell fils peuvent être redirigées au moment de l'appel. Ceci redirige alors implicitement toutes les entrées et sorties standard des commandes appelées dans le shell fils.

− Une variable **x** du shell père ne sera connue dans un shell fils qu'après lancement de la commande **export x**, mais les modifications de la valeur de **x** effectuées dans le shell fils ne seront pas répercutées dans le shell père.

Les fonctions en shell

En shell, on appelle "fonction" une séquence de commandes repérable par un nom qui est le nom de la fonction. Les commandes constituant une fonction sont exécutées dans le cadre du shell courant, et non dans le cadre d'un shell fils, comme c'est le cas pour un script ou pour une séquence de commandes placées entre parenthèses.

Syntaxe de la déclaration d'une fonction :

```
nom()
{
commandes
}
```

Une fonction doit être déclarée avant d'être appelée. Pour appeler une fonction, il suffit d'utiliser son nom, **sans les parenthèses**. Comme pour un script shell, l'appel d'une fonction peut être éventuellement suivi d'une liste d'arguments, accessibles dans la fonction à l'aide des paramètres positionnels **$1**, **$2**, ..., **$9** (les paramètres **$#** et **$*** sont également initialisés).

Une fonction se termine soit après exécution de la dernière commande située avant l'accolade fermante, auquel cas le code de retour est celui de cette dernière commande, soit après exécution d'une commande **return [n]**, auquel cas le code de retour est l'argument **n** de la commande **return** (ou **0** si cet argument est absent). Dans les deux cas, le code de retour de la fonction est affecté au paramètre **$?** du shell courant.

Attention : la commande **return** ne peut être utilisée qu'à l'intérieur d'une fonction (remarquer l'analogie avec la fonction ***return*** du langage C).

Les principales commandes sous UNIX

cat	**grep**	**rm**
cd	**head**	**rmdir**
chmod	**kill**	**set**
cmp	**ls**	**sh**
cp	**mail**	**shift**
cut	**man**	**sort**
date	**mkdir**	**tail**
echo	**more**	**tee**
eval	**mv**	**test**
exit	**ps**	**tr**
expr	**pwd**	**wc**
file	**read**	

La commande `cat`

Elle permet d'afficher le contenu de l'entrée standard (ou, s'il y a lieu, le contenu du ou des fichiers donnés en arguments, en les concaténant) sur la sortie standard.

Syntaxe :

```
cat [-nbsuvet] [nom_fichier ...]
```

Options :

- **-n** : numérote les lignes, avant de les envoyer sur la sortie standard ;
- **-b** : identique à **-n**, sans numérotation des lignes vides ;
- **-e** : un caractère **$** est ajouté à la fin de chaque ligne ;
- autres options : *cf.* la description de la commande **cat**, en tapant : **man cat**

Exemple :

```
cat fich1 fich2 > fich3
```

Cette commande recopie **fich1** puis, à la suite, **fich2**, le tout dans **fich3**

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande `cd`

Elle permet de changer de répertoire, de manière relative ou absolue.

Syntaxe :

```
cd [nom_répertoire]
```

Lorsque la commande est tapée sans argument, on se positionne dans le répertoire principal ("home directory").

Options :

néant.

Exemple :

```
cd ../../TD
```

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande chmod

Elle permet de modifier les droits d'accès à un ou à plusieurs fichiers ou répertoires dont on est le propriétaire.

Syntaxe :

```
chmod [-fR] [type_utilisateur...]+|-r|w|x nom_de_fichier [...]
```

- Le type de l'utilisateur est **u** pour indiquer qu'il s'agit du propriétaire du fichier ou du répertoire, **g** s'il s'agit d'un membre du groupe, et **o** s'il s'agit de tout autre utilisateur.
- Le signe **+** signifie qu'on ajoute des droits d'accès, alors que **-** signifie qu'on en enlève.
- **r**, **w**, ou **x** indiquent le type d'accès que l'on modifie (lecture, écriture, ou exécution).

Options :

cf. la description de la commande **chmod**, en tapant : **man chmod**

Exemple :

```
chmod g+rx /home/mod4g0/TD/td12ex*
```

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande cmp

Elle effectue la comparaison des contenus de deux fichiers. Appelée sans option, elle renvoie le numéro du premier octet qui diffère. Elle ne retourne rien si les deux fichiers sont identiques.

Syntaxe :

```
cmp [-l] [-s] nom_fichier1 nom_fichier2 [skip1] [skip2]
```

Options :

- **`-l`** : affiche les numéros de tous les octets qui diffèrent et la valeur de la différence ;
- **`-s`** : n'affiche aucun message, mais gère les codes d'erreurs.

Exemple :

```
cmp 'ls|head -1' 'ls|tail -1'
```

Codes d'erreur retournés :

- fichiers identiques : **`0`**
- fichiers différents : **`1`**
- fichiers inaccessibles : **`2`**

Remarques :

− Si **`nom_fichier1`** vaut **`-`**, alors la comparaison est effectuée sur l'entrée standard.
− Les arguments optionnels **`skip1`** et **`skip2`** permettent de comparer le contenu du premier fichier, à partir de l'octet numéro **`skip1`**, au contenu du deuxième fichier, à partir de l'octet numéro **`skip2`**.

La commande `cp`

Elle permet de copier un ou plusieurs fichiers, en changeant éventuellement leurs noms.

Syntaxe :

```
cp [-fip] nom_fichier nom_copie_fichier
```

Si on désire copier plusieurs fichiers en une seule commande, on peut utiliser les métacaractères du shell, comme dans l'exemple ci−après, mais on ne peut pas écrire une liste de fichiers.

Options :

cf. la description de la commande **`cp`**, en tapant : **`man cp`**

Exemple :

```
cp PREPA/TD/td12ex* TD
```

Cette commande copie tous les fichiers du répertoire **`PREPA/TD`**, de nom commençant par **`td12ex`**, dans le répertoire **`TD`**, sans changer leurs noms.

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `cut`

Elle permet de supprimer une partie des lignes d'un ou de plusieurs fichiers, ou de l'entrée standard.

Syntaxe :

`cut -b`liste **`[-n] [nom_fichier ...]`**

ou :

`cut -c`liste **`[nom_fichier ...]`**

ou :

`cut -f`liste **`[-d`**délimiteur**`] [-s] [nom_fichier ...]`**

Options :

- **`-b`** : ne retient sur chaque ligne que les octets situés à une position donnée par **`liste`**
- **`-c`** : ne retient sur chaque ligne que les caractères situés à une position donnée par **`liste`** Par exemple, l'option **`-c1-50`** conduira la commande à ne garder que les 50 premiers caractères de chaque ligne ;
- **`-f`** : ne retient qu'une liste de champs (séparés par des tabulations ou par un délimiteur spécifié par l'option **`-d`**) ;
- **`-s`** : supprime les lignes ne présentant pas de délimiteur ;
- autres options : *cf.* la description de la commande **`cut`**, en tapant : **`man cut`**

Exemple :

```
ls | cut -f1-2 -s -d'.'
```

Cette commande affiche la liste des noms de fichiers ou de sous-répertoires du répertoire courant comprenant une extension.

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

Remarque :

Les options **`-b`**`liste`, **`-c`**`liste`, **`-f`**`liste` et **`-d`**`délimiteur` peuvent être écrites avec un espace situé juste avant la `liste` ou le `délimiteur`.

La commande `date`

Elle affiche la date.

Syntaxe :

```
date [-u] [+format]
```

Options :

cf. la description de la commande **`date`**, en tapant : **`man date`**

Exemple :

`date`

affiche la date sous la forme :

Mon Dec 8 21:05:47 MET 1997

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `echo`

Elle permet d'afficher une ou plusieurs chaînes de caractères à l'écran. Cette commande est interprétée de manière sensiblement différente d'un shell à un autre.

Syntaxe :

`echo [chaîne ...]`

Options :

néant.

Exemple :

`echo "répertoire courant : ‘ls‘"`

Cette commande affiche le message **`répertoire courant :`** , suivi de la liste des fichiers et sous−répertoires contenus dans le répertoire courant.

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

Remarque :

La commande **`echo`** reconnaît également certains caractères spéciaux commençant par **`\`** et appelés "caractères de contrôle", qui peuvent être écrits de deux manières différentes :

- **`\r`** pour le retour−chariot, **`\n`** pour le saut de ligne, **`\f`** pour le saut de page, **`\t`** pour la tabulation, **`\c`** pour supprimer le retour à la ligne, ... ;
- **`\nnn`**, où **`nnn`** est un code ASCII en octal (par exemple, **`\012`** désigne le saut de ligne).

La commande `eval`

Elle procède en deux temps :

- dans un premier temps, la commande est remplacée par la liste de ses arguments, avec interprétation des métacaractères du shell ;
- dans un deuxième temps, cette liste d'arguments est considérée comme une commande shell, et exécutée en tant que telle, donc en particulier avec, à nouveau, interprétation des métacaractères du shell.

Syntaxe :

```
eval [argument1 ...]
```

Options :

néant.

Exemple :

```
a=1
c=a
a1=3
eval b=\$$c$a
echo $b
```

Cette séquence affichera la valeur **3**, car la première évaluation de la commande **`eval`** produit la commande suivante :

```
b=$a1
```

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `exit`

Elle permet de sortir du shell courant, en précisant éventuellement la valeur du code de retour.

Syntaxe :

```
exit [code_retour]
```

Options :

néant.

Exemple :

Si la commande :

```
exit
```

est tapée dans la fenêtre "racine", cela provoque une déconnexion.

Codes d'erreur retournés :

- si l'argument **code_retour** existe : la valeur (entière) de **code_retour** ;
- sinon : le code d'erreur de la dernière commande exécutée dans le shell courant.

La commande expr

Elle permet d'effectuer des opérations arithmétiques et logiques, ainsi que des manipulations de chaînes de caractères, et renvoie le résultat sur la sortie standard.

Syntaxe :

```
expr argument1 opérateur1 argument2 [opérateur2 argument3 ...]
```

où les arguments **argument1**, **argument2**, **argument3**,... peuvent être des chaînes de caractères, des entiers, des résultats de commandes, des valeurs de paramètres positionnels, des valeurs de variables ou toute combinaison des cas précédents, et où l'**opérateur** appartient à la liste non exhaustive suivante :

- **+** (addition), **−** (soustraction), ***** (multiplication), **/** (quotient de la division entière), **%** (reste de la division entière), si **argument1** et **argument2** ont des valeurs entières ;
- **=** (égalité), **!=** (différence), **\>**, **\>=**, **\<**, **\<=** (comparaisons), si **argument1** et **argument2** sont des entiers ou des chaînes de caractères.

- **:** (extraction de sous−chaînes de caractères). Avec cet opérateur, l'argument **argument1** doit être une chaîne de caractères, et **argument2** doit être une expression régulière, dans laquelle on spécifie l'ensemble des caractères devant être retirés de la chaîne **argument1** par des caractères ou des métacaractères des expressions régulières, et on indique la séquence de caractères à conserver grâce à l'expression régulière **\(.*\)**
  Dans le cas où la correspondance est impossible, le résultat produit est la chaîne vide.

Options : néant.

Exemple 1 :

```
a=2
b=3
a=‘expr $b % $a‘
echo $a
```

produira l'affichage de la valeur **1**

Exemple 2 :

```
chaine="caracteres"
sschaine=‘expr $chaine : ’\(.*\)t’‘
echo $sschaine
```

produira l’affichage suivant :

```
carac
```

On aurait pu écrire l’expression régulière sous d’autres formes :

```
’\(.*\)t.*’
’\(.*\)teres’
```
...

Codes d’erreur retournés :

- résultat non nul : **0**
- résultat nul : **1**
- expression incorrecte : **2**
- opération interrompue pour cause d’erreur : **> 2**

Remarque :

Dans le cas où la commande **expr** est lancée avec plusieurs opérateurs, il faut être vigilant, car l’ordre des opérations ne se fait ni de gauche à droite, ni de droite à gauche, mais suivant un ordre prédéfini de priorité des opérateurs.

La commande `file`

Elle permet de déterminer le type d’un ou de plusieurs fichiers, ce type appartenant à la liste non exhaustive suivante :

- shell script ;
- directory ;
- data ;
- ascii text ;
- executable shell script ;
- ELF 32−bit MSB executable SPARC Version 1, dynamically linked, not stripped ;
- GIF file, v87 ;
- USTAR tar archive ;
- ...

Syntaxe :

```
file [−h] [−m types_fichiers] [−f fichier_liste] nom_fichier [...]
```

Options : *cf.* la description de la commande **file**, en tapant : **man file**

Exemple :

Si le fichier **exo1.c** contient un programme C, la commande :

```
file exo1.c
```

produira l'affichage suivant :

```
xo1.c: ascii text
```

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande grep

Elle permet de rechercher une expression régulière dans un ou plusieurs fichiers donnés en arguments.

Syntaxe :

```
grep [-bchilnsvw] expression_régulière [nom_fichier ...]
```

Options :

- **-l** : la commande ne renvoie que le nom du fichier où une occurence est trouvée (au lieu d'afficher les lignes où l'occurence apparaît) ;
- **-c** : affiche uniquement, pour chaque fichier, le nombre de lignes où l'occurence apparaît ;
- **-v** : affiche les lignes qui ne contiennent pas d'occurence de l'expression régulière ;
- autres options : *cf.* la description de la commande **grep**, en tapant : **man grep**

Exemple :

```
grep 'grep' *
```

Cette commande affiche toutes les lignes de tous les fichiers du répertoire de travail qui contiennent une occurence de la chaîne de caractères **grep**

Codes d'erreur retournés :

- une occurence au moins a été trouvée : **0**
- aucune occurence n'a été trouvée : **1**
- opération interrompue pour cause d'erreur : **2**

La commande `head`

Elle copie l'entrée standard, ou le ou les fichiers donnés en arguments, sur la sortie standard, jusqu'à une position désignée.

Syntaxe :

```
head [-position|-nposition] [nom_fichier ...]
```

Options :

- `-position` : recopie les `position` premières lignes du fichier ou de l'entrée standard (à défaut, les 10 premières lignes) ;
- autres options : *cf.* la description de la commande **`head`**, en tapant : **`man head`**

Exemple :

```
head -5 fichier.c
```

Cette commande renvoie les 5 premières lignes du fichier **`fichier.c`**

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `kill`

Elle permet d'envoyer un signal à un ou à plusieurs processus dont on connaît les identificateurs.

Syntaxe :

```
kill [-signal] identificateur_processus [...]
```

ou :

```
kill -l
```

Options :

- **`-l`** : affichage de la liste des signaux qui peuvent être envoyés à un processus ;
- **`-signal`** : envoi du signal **`signal`** au processus.

Exemple :

```
kill -9 1345
```

Cette commande envoie le signal **`9`** au processus d'identificateur **`1345`**
Le signal **`9`**, qui permet de "tuer" un processus, est le plus souvent utilisé. Le signal envoyé par défaut est **`15`**

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande `ls`

Cette commande permet d'afficher le contenu d'un ou de plusieurs répertoires, ou des renseignements concernant un ou plusieurs fichiers.

Syntaxe :

`ls [−RadLCxmlnogrtucpFbqisf1AM] [nom_répertoire ...]`
ou :
`ls [−RadLCxmlnogrtucpFbqisf1AM] [nom_fichier ...]`

Cas particulier :

- s'il n'y a aucun paramètre : la commande affiche le contenu du répertoire courant ;
- si un répertoire est passé en paramètre : la commande affiche le contenu de ce répertoire ;
- si un fichier est passé en paramètre : la commande indique si ce fichier est accessible ou non.

Options :

- **`−a`** : affiche également les fichiers commençant par le caractère **`.`**
- **`−l`** : affiche une première ligne indiquant la taille totale du répertoire en kilo−octets ; affiche ensuite, pour chaque répertoire ou fichier accessible, ses principales caractéristiques sur une ligne : nature (répertoire ou fichier), droits d'accès, nombre de liens, nom du propriétaire, nom du groupe, taille en octets, date de dernière modification, nom (qui est compris entre le 55ème caractère et le 80ème caractère de la ligne) ;
- autres options : *cf.* la description de la commande **`ls`**, en tapant : **`man ls`**

Exemple 1 :

```
tgv% ls −l
```

Exemple 2 :

```
ls ‘ls‘
```

Cette commande affiche la liste des fichiers du répertoire courant, puis la liste des contenus des sous−répertoires du répertoire courant.

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande `mail`

Elle permet d'envoyer ou de lire un courrier électronique.

Syntaxe :

Envoi d'un courrier : **`mail [-tw] [-m`**`type_du_message`**`] adresse_destinataire [...]`**
Lecture du courrier : **`mail [-ehpPqr] [-f`**`fichier`**`]`**

Options :

cf. la description de la commande **`mail`**, en tapant : **`man mail`**

Exemple :

```
mail mod4g0@mdi.edu.ups-tlse.fr < message
```

Cette commande envoie le texte contenu dans le fichier **`message`** à l'adresse du destinataire.

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- pas de courrier ou erreur d'initialisation : **`1`**
- erreur après initialisation : **`> 1`**

La commande `man`

Elle permet d'obtenir la description d'une commande à l'écran.

Syntaxe :

`man [-] [-adFlrt] [-M``arborescence`**`] [-T`**`macro-package`**`] [-s`**`section`**`] nom_commande [...]`**

Options :

cf. la description de la commande **`man`**, en tapant : **`man man`**

Exemple :

```
man ls
```

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `mkdir`

Elle permet de créer un ou plusieurs répertoires, dont les noms sont précisés en arguments. Elle renvoie un message d'erreur dans les cas où la création est impossible.

Syntaxe :

`mkdir [-m``mode`**`] [-p] nom_répertoire [...]`**

Options :

- **`-m`**`mode` : indique le mode d'accès au répertoire à créer ;
- **`-p`** : crée tous les répertoires intermédiaires, si nécessaire.

Exemple 1 :

Supposons que l'on veuille créer le répertoire **`rep1`** et son sous-répertoire **`ssrep1`**, dans le répertoire **`rep`** du répertoire courant. Il faut taper la commande :

`mkdir -p rep/rep1/ssrep1`Exemple 2 :

Supposons que l'on veuille créer le répertoire **`rep2`** dans le répertoire courant, en enlevant les droits en lecture et en écriture à toute personne non propriétaire du compte. Il faut taper la commande :

`mkdir -mog-rw rep2`Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `more`

Elle permet de faire afficher le contenu d'un ou de plusieurs fichiers page par page. Elle est une version plus interactive de la commande **`cat`** :

- Pour afficher la page suivante, il faut taper sur la barre d'espace.
- Pour afficher seulement une nouvelle ligne, il faut taper un retour-chariot.
- Pour interrompre l'affichage, il suffit de taper le caractère **`q`**

Syntaxe :

`more [-cdflrsuw] [-lignes] [+numéro_ligne] [+/modèle] [nom_fichier ...]`

Options :

cf. la description de la commande **`more`**, en tapant : **`man more`**

Exemple :

`more temp`

Codes d'erreur retournés :

cf. la description de la commande **`more`**, en tapant : **`man more`**

La commande `mv`

Elle permet de renommer un fichiers ou un répertoire, ou de déplacer un ou plusieurs fichiers dans l'arborescence, en changeant éventuellement leurs noms.

Syntaxe :

```
mv [-fi] ancien_nom nouveau_nom
```

ou :

```
mv [-fi] nom_fichier ... nom_répertoire
```

Options :

cf. la description de la commande **`mv`**, en tapant : **`man mv`**

Exemple 1 :

```
mv charabia.c ex1.c
```

Exemple 2 :

```
mv TD/td12ex* TD/td13ex* PREPA
```

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `ps`

Elle permet d'afficher la liste des processus en cours d'activité. Lancée sans option, elle n'affiche que les processus actifs de l'utilisateur associés au terminal utilisé.

Syntaxe :

```
ps [-aAcdefjl] [-g nom] [-n nom] [[-o format] ...] [-p nom] [-s
nom] [-t terminaux] [-u utilisateurs] [-U nom] [-G nom]
```

Les processus sont affichés suivant un format variable, mais le format par défaut comprend les quatre champs suivants sur une même ligne :

- **`PID`** : numéro d'identification du processus ;
- **`TTY`** : terminal associé au processus ;
- **`TIME`** : temps d'exécution cumulé du processus ;
- **`CMD`** : nom de la commande associée au processus.

Options :

- **-e** : affichage des informations sur tous les processus ;
- autres options : *cf.* la description de la commande **ps**, en tapant : **man ps**

Exemple :

```
ps -e | grep asedit | wc -l
```

Cette commande retourne le nombre de processus en cours d'exécution associés à l'éditeur **asedit**

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande pwd

Elle affiche le nom absolu du répertoire de travail (appelé aussi "répertoire courant").

Syntaxe :

```
pwd
```

Options :

néant.

Exemple :

```
tgv% pwd
/home/mod4g0/TD
```

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande read

Cette commande permet d'affecter des valeurs à des variables à partir de l'entrée standard.
Syntaxe :

```
read [-r] nom_variable [...]
```

Options :

-r : le caractère **** est considéré comme un caractère ordinaire.

Exemple :

```
read a b
echo $a
echo $b
```

Si la ligne suivante est tapée au clavier :

```
c'est un exemple
```

alors les commandes **`echo`** produiront les affichages suivants :

```
c'est
un exemple
```

ce qui signifie que, si le nombre de mots (séparés par des espaces ou des tabulations) est supérieur au nombre d'arguments de la commande **`read`**, alors la variable correspondant au dernier argument reçoit toute la fin de la ligne.

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur, ou fin de fichier détectée : **`> 0`**

La commande `rm`

Elle permet d'effacer un ou plusieurs fichiers passés en arguments.

Syntaxe :

```
rm [-f] [-i] nom_fichier [...]
```

Options :

- **`-i`** : demande confirmation avant effacement ;
- autres options : *cf.* la description de la commande **`rm`**, en tapant : **`man rm`**

Exemple :

```
rm *
```

Cet exemple doit être évité en général !

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `rmdir`

Elle permet d'effacer un ou plusieurs répertoires passés en arguments. Ces répertoires doivent être vides.

Syntaxe :

```
rmdir [-ps] nom_répertoire ...
```

Options :

cf. la description de la commande **`rmdir`**, en tapant : **`man rmdir`**

Exemple :

```
rmdir ..
```

Cette commande provoque systématiquement une erreur, car le répertoire père du répertoire de travail ne peut pas être vide !

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `set`

Cette commande permet d'affecter des valeurs aux paramètres positionnels **`$1`**, **`$2`**, **`$3`**, ..., **`$9`**. Appelée sans paramètres, elle permet aussi d'afficher les valeurs de toutes les variables du shell courant :

- les variables prédéfinies ;
- les variables définies par l'utilisateur.

Cette commande est très différente entre shell de Bourne, Korn shell et C_shell. On rappelle que c'est le shell de Bourne qui est présenté ici.

Syntaxe :

```
set [-aefhkntuvx] [valeur1] [...]
```

Options :

cf. la description de la commande **`set`**, en tapant : **`man set`**

Exemple 1 :

```
set alpha beta
```

Cette commande affecte la valeur **`alpha`** au paramètre positionnel **`$1`**, et la valeur **`beta`** au paramètre positionnel **`$2`**

Exemple 2 :

```
x=1
set
```

produira l'affichage suivant :

```
DISPLAY=141.115.12.137:0
GUIDEHOME=/usr/openwin/devguide
HOME=/home/mod4g0
...
x=1
```

Codes d'erreur retournés :

cf. la description de la commande **set**, en tapant : **man set**

La commande sh

Cette commande permet de lancer un shell fils de type **shell de Bourne**. Elle permet aussi d'exécuter un script shell dans le cadre de ce shell fils.

Syntaxe 1 :

```
sh [-acefhiknprstuvx]
```

Le shell fils qui est lancé lit et interprète les commandes tapées sur l'entrée standard. Pour retourner au shell père, il faut taper la commande **exit**

Syntaxe 2 :

```
sh [-acefhiknprstuvx] [nom_script ...]
```

Les arguments sont des noms de scripts shell.

Options :

cf. la description de la commande **sh**, en tapant : **man sh**

Exemple 1 :

```
tgv% sh
$ pwd
/home/mod4g0
$ exit
tgv%
```

Exemple 2 :

```
sh exo1.sh
```

Cette commande lance le script shell contenu dans le fichier **exo1.sh**
Une façon équivalente de lancer ce script est de le rendre exécutable, en tapant la commande :

chmod u+x exo1.sh

puis en tapant :

exo1.sh

Codes d'erreur retournés :

Le code d'erreur retourné par la commande **sh** est le code d'erreur retourné par la dernière commande exécutée dans le cadre du shell fils. Dans le cas où un script shell est lancé, si une commande retourne un code d'erreur non nul, alors l'exécution du script est interrompue.

La commande shift

Elle permet de décaler les arguments passés à l'appel d'un script d'un rang ou de **n** rangs vers la droite.

Syntaxe :

shift [n]

Options :

néant.

Exemple :

Soit le script :

deca.sh

```
echo "\nAvant décalage : $1 $2 $3 $4 $5 $6 $7 $8 $9"
shift
echo "Après décalage : $1 $2 $3 $4 $5 $6 $7 $8 $9\n"
```

Voici deux exemples d'exécution de ce script, avec des nombres d'arguments différents :

```
$ deca.sh 10 9 8 7 6 5 4 3 2 1

Avant décalage : 10 9 8 7 6 5 4 3 2
Après décalage : 9 8 7 6 5 4 3 2 1

$ deca.sh 5 4 3 2 1

Avant décalage : 5 4 3 2 1
Après décalage : 4 3 2 1
```

Donc **$1** reçoit la valeur de **$2**, **$2** reçoit la valeur de **$3**, etc...

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

Remarques :

− Lors de l'appel de la commande **shift**, l'absence d'argument **n** équivaut à une valeur de cet argument égale à **1**.
− La valeur de **$#** est mise à jour par la commande **shift**.

La commande sort

Elle lit un ou plusieurs fichiers ou l'entrée standard, et affiche les différentes lignes, triées sur un ou plusieurs champs (par défaut, sur un seul champ qui est la ligne), selon l'ordre lexicographique (les séparateurs de deux champs sont par défaut l'espace et la tabulation).

Syntaxe :

sort [−cmu] [−o sortie] [−T répertoire] [−y[mémoire]] [−dfiMnr] [−b] [−t caractère] [−k clé] [+pos1 [−pos2]] [nom_fichier ...]

Options :

- **+pos1** et **−pos2** : permettent de délimiter les champs utilisés pour le tri ; les valeurs **pos1** et **pos2** ont le format **m.n**, où **m** est le nombre de champs à ignorer depuis le début de la ligne, et **n** le nombre de caractères à ignorer depuis le début du champ (**.n** est optionnel) ; si **−pos2** est omis, le tri porte de **+pos1** jusqu'à la fin de la ligne ;
- **−u** : affichage d'un seul exemplaire des lignes identiques ;
- **−m** : "fusion" des fichiers devant être triés ;
- **−c** : si le fichier à trier est déjà trié, aucun affichage ; sinon, affichage des lignes triées, et retour d'un code d'erreur égal à **1** ;
- autres options : *cf.* la description de la commande **sort**, en tapant : **man sort**

Exemple :

Si le fichier **liste_noms** contient une liste de noms−prénoms structurés de la manière suivante :

```
Alain VERSE
Jean REVE
...
```

alors la commande :

```
sort +1 liste_noms
```

affichera ce fichier, trié par ordre alphabétique des noms de famille.

Codes d'erreur retournés :

- pas d'erreur ou option **−c** sur fichier déjà trié : **0**
- option **−c** sur fichier non trié : **1**
- opération interrompue pour cause d'erreur : **> 1**

La commande tail

Elle copie l'entrée standard, ou le fichier donné en argument, sur la sortie standard, à partir de la position désignée.

Syntaxe :

```
tail [+/−position[lbcr]] [nom_fichier]
```

Options :

- **+**position : recopie, à partir de la position donnée, en partant du début ;
- **−**position : recopie, à partir de la position donnée, en partant de la fin ;
- **l** : position indique un numéro de ligne ;
- **c** : position indique un numéro de caractère ;
- autres options : *cf.* la description de la commande **tail**, en tapant : **man tail**

Exemple :

```
tail −15c fichier.c
```

Cette commande renvoie les 15 derniers caractères du fichier **fichier.c**

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande `tee`

Elle permet d'afficher l'entrée standard simultanément sur la sortie standard et sur le ou les fichiers passés en arguments.

Syntaxe :

```
tee [-ai] [nom_fichier ...]
```

Options :

- **`-a`** : les données reçues sur l'entrée standard sont concaténées au fichier ou aux fichiers passés en arguments.
- autres options : *cf.* la description de la commande **`tee`**, en tapant : **`man tee`**

Exemple :

```
ls | tee liste_fichiers
```

Cette commande affiche, d'une part, le contenu du répertoire courant à l'écran et écrit, d'autre part, ce contenu dans le fichier **`liste_fichiers`**

Codes d'erreur retournés :

- pas d'erreur : **`0`**
- opération interrompue pour cause d'erreur : **`> 0`**

La commande `test`

Elle permet d'effectuer des tests de comparaison, en renvoyant la valeur **`0`** lorsque la comparaison est vraie, et une autre valeur sinon. Cette commande est surtout utile en complément aux structures de contrôle, comme la structure de contrôle de condition qui sera vue dans le paragraphe suivant.

Syntaxe 1 :

```
test comparaison
```

ou :

`[ comparaison ]`(bien respecter les espaces dans ce 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 ;
- **`! comparaison1`** pour le **NON** logique ;

Des parenthèses précédées de caractères **** (les parenthèses étant des métacaractères du shell)

peuvent être nécessaires en cas 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 :

```
v1 comparateur v2
```

où **v1** et **v2** sont des expressions du shell courant.

Le comparateur **comparateur** ne s'écrit pas comme en C. Si **v1** et **v2** sont entières :

- **-eq** désigne l'égalité ;
- **-ne** désigne la non égalité ;
- **-gt** signifie "strictement supérieur" ;
- **-ge** signifie "supérieur ou égal" ;
- **-lt** signifie "strictement inférieur" ;
- **-le** signifie "inférieur ou égal".

Si **v1** et **v2** sont des chaînes de caractères :

- **=** désigne l'égalité ;
- **!=** désigne la non égalité.

Remarque :

Pour tester si la variable **chaine** a comme valeur la chaîne vide, on ne peut pas écrire **test $chaine = ""** car, si cette chaîne est vraiment vide, après évaluation des métacaractères, cette commande sera évaluée sous la forme **test = ""**, qui est incorrecte. Cela justifie l'introduction de la deuxième syntaxe suivante pour la commande **test** :

Syntaxe 2 :

```
test chaine
```

Dans cette écriture, la commande **test** teste si **chaine** est une chaîne vide. Il existe une troisième syntaxe pour la commande **test** :

Syntaxe 3 :

```
test -fdrwx nom ...
```

Options :

- **-f** : vérifie que **nom** est un fichier ;
- **-d** : vérifie que **nom** est un répertoire ;
- **-r** : vérifie que **nom** est accessible en lecture pour l'utilisateur ;

- **-w** : vérifie que **nom** est accessible en écriture pour l'utilisateur ;
- **-x** : vérifie que **nom** est un exécutable pour l'utilisateur.

Exemple 1 :

```
test $1 -gt 0 -a $# -eq 2
```

Exemple 2 :

```
test $chaine
```

Exemple 3 :

```
test -d /home/mod4g0/TD
```

Codes d'erreur retournés :

- **condition** évaluée à "vrai" : **0**
- **condition** évaluée à "faux", ou **condition** absente : **1**
- opération interrompue pour cause d'erreur : **> 1**

La commande tr

Elle permet le "transcodage" de caractères entre l'entrée standard et la sortie standard. Cette commande fonctionne aussi avec les caractères de contrôle déjà signalés plus haut.

Syntaxe :

```
tr -ds chaine
```

ou :

```
tr [-c] chaine1 chaine2
```

Options :

- **-d** : supprime toutes les occurences des caractères composant la chaîne de caractères **chaine** ;
- **-s** : élimine les répétitions des caractères composant la chaîne de caractères **chaine**, en n'en laissant qu'un à chaque fois ;
- autres options : *cf.* la description de la commande **tr**, en tapant : **man tr**

Exemple 1 :

```
tr 'abc' 'ABC'
```

Cette commande attend l'entrée de données au clavier. Si on tape :

```
calebasse
```

chaque occurence d'un **a** est remplacé par un **A**, chaque occurence d'un **b** est remplacé par un **B** et chaque occurence d'un **c** est remplacé par un **C**, ce qui donne dans le cas présent :

```
CAleBAsse
```

Exemple 2 :

```
tr '\012' ' '
```

Cette commande va substituer chaque saut de ligne par un espace.

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

La commande wc

Cette commande (abréviation de "word count") permet de compter le nombre de lignes, de mots ou de lettres dans le ou les fichiers passés en arguments ou, à défaut, sur l'entrée standard.

Syntaxe :

```
wc [-c|-m|-C] [-lw] [nom_fichier ...]
```

Options :

- **-l**, **-w**, **-c** : permet de retourner respectivement les nombres de lignes, de mots ou de lettres. A défaut, la commande est appelée avec les options **-lwc**
- autres options : *cf.* la description de la commande **wc**, en tapant : **man wc**

Exemple :

```
wc -l exam
```

Cette commande affiche le nombre de lignes du fichier **exam**

Codes d'erreur retournés :

- pas d'erreur : **0**
- opération interrompue pour cause d'erreur : **> 0**

Ces pages ont été réalisées par J.D. Durou et Ph. Joly. Pour tout commentaire, envoyer un mail à durou@irit.fr ou à joly@irit.fr

Commandes de base de l'éditeur Emacs

Lancement

```
emacs
emacs &
emacs toto.c
emacs toto.c &
```

Le caractère **&** qui est à la fin de la commande sert à la lancer en "arrière−plan", ce qui libère le terminal pendant l'édition et permet d'entrer d'autres commandes, pour la compilation, par exemple.

Utilisation "à la souris"

Lorsqu'il est lancé dans l'environnement *X−Window*, *Emacs* permet de réaliser toutes les fonctions d'édition de base "à la souris", notamment :

- se déplacer avec l'ascenseur,
- placer avec le bouton gauche le point d'insertion du texte entré au clavier,
- faire des sélections et copier (bouton gauche),
- coller (bouton du milieu),
- utiliser les menus.

En particulier, dans le menu **Files**, on trouve **Save Buffer** et **Exit Emacs**. Dans la terminologie d'*Emacs*, le buffer est la zone de mémoire qui contient la version du texte en cours d'édition et **Save Buffer** permet de la sauver dans un fichier.

Les commandes *Emacs*

Dans le menu **Files**, par exemple, à côté de **Save Buffer** on voit **C−x C−s** entre parenthèses. Sous ce raccourci clavier se cache une vraie fonction Lisp « Save Buffer » interne à *Emacs.*

C−x C−s représente la séquence de touches « **Control x Control s** », i.e. la touche **Ctrl** étant enfoncée, on appuie successivement sur les touches **x** puis **s.**

De manière similaire, on retiendra que **C−x C−c** permet de **quitter *Emacs.***

Il y a aussi des commandes **Meta**, par exemple **M−v** qui remonte d'une page. Une commande **Meta** s'obtient avec la touche **Alt** qui s'utilise comme **Ctrl**.
Si le clavier ne comporte pas de touche **Meta**, on peut utiliser la touche **Escape**, par exemple, **M−v** s'obtient aussi par la séquence de touches "**Escape v**", i.e. on appuie **successivement** sur les touches **Esc** puis **v**.

Emacs ayant été conçu initialement pour une utilisation tout au clavier, on peut accéder à toutes ses fonctions internes en tapant le nom de la fonction dans le buffer créé par le raccourci **M−x .**

exemple : **M−x** suivi de « insert−file » a le même effet que le raccourci clavier **C−x i** .

Il existe un **mécanisme d'historique des commandes**, accessible par les flèches haut et bas.

Certains raccourcis sont très intéressants à connaître parce qu'ils se retrouvent dans d'autres applications, avec le même effet.
Voici quelques commandes de base d'*Emacs* qui sont également reconnues par **l'interpréteur de commandes tcsh** et par les boîtes de saisie de **l'interface *X−Window*** :

Commande	**Action**	**Fonction Lisp associée**
C−b	Recule le curseur d'un caractère	backward−char
C−f	Avance le curseur d'un caractère	forward−char
C−a	Envoie le curseur en début de ligne	beginning−of−line
C−e	Envoie le curseur en fin de ligne	end−of−line
C−p	Remonte le curseur d'une ligne	previous−line
C−n	Descend le curseur d'une ligne	next−line
C−d	Efface le caractère situé sous le curseur	delete−char
C−k	Détruit le reste de la ligne	kill−line

Autres commandes, plus spécifiques à *Emacs* :

Annulation−Répétition d'une commande :

C−_	Défait la dernière commande (réitérable)	undo
C−x u	Idem	
C−g	Interrompt la frappe d'une commande	keyboard−quit
C−x Esc	Répète la dernière commande	repeat−complex−command

Déplacement du curseur :

M−b	Recule d'un mot	backward−word
M−f	Avance d'un mot	forward−word
C−v	Écran suivant	scrool−up
M−v	Écran précédent	scrool−down
M−<	Curseur au début du buffer	beginning−of−buffer
M−>	Curseur en fin de buffer	end−of−buffer

M−x `goto−line n`	Curseur à la ligne n
M−x `what−page`	Affiche la position courante du curseur

Sélection d'une zone de texte :

La sélection courante (**région**) est la partie du texte comprise entre le dernier marquage (obtenu par `C−Space`) et la position actuelle du curseur. On peut connaître la position actuelle du marqueur en utilisant la commande `C−x C−x` expliquée ci−dessous.

C−x C−x	Échange les positions respectives du marqueur et du curseur	
C−Space	Positionne le marqueur là où se trouve le curseur (Space = barre d'espacement)	
C−@	Idem.	
M−h	Sélectionne le paragraphe contenant le curseur	
C−x h	Sélectionne tout le texte	mark−whole−buffer

Remarque : cliquer sur le bouton gauche de la souris est une autre façon de réaliser un marquage.

Couper, copier, coller :

Après avoir sélectionné une zone de texte (voir paragraphe précédent) on applique les commandes :

C−w	Coupe la région sélectionnée	kill−region
M−w	Copie la région sélectionnée	copy−region−as−kill
C−y	Colle le dernier bloc coupé ou copié	yank

Recherche−remplacement :

C−s	Recherche la première occurrence d'une chaîne de caractères, de la position du curseur jusqu'à la fin du texte (isearch−forward).
C−r	Recherche la première occurrence d'une chaîne de caractères, de la position du curseur jusqu'au début du texte (isearch−backward).

En retapant **C−s** ou **C−r** on poursuit la recherche; en tapant **RET** on arrête la recherche.

M−x replace−string RET ancienne chaîne RET nouvelle chaîne RET
Remplace toutes les occurrences de l'ancienne chaîne par la nouvelle chaîne, de la position du curseur jusqu'à la fin du texte.

M−% Remplace sur demande la chaîne spécifiée (replace−query)

Remarque : pour insérer un caractère de contrôle, comme un saut de ligne **(^J)** dans une chaîne de caractères, il faut le faire précéder de **C−q** (quoted−insert).

Utilisation des fenêtres :

C−x 2	Donne deux fenêtres dans le sens de la hauteur	split−window−vertically
C−x 5	Donne deux fenêtres dans le sens de la largeur	split−window−horizontally
C−x 0	Supprime la fenêtre courante	delete−window
C−x 1	Supprime toutes les fenêtres sauf la fenêtre courante	delete−other−windows
C−x o	Change la fenêtre courante	other−window

Fichiers et buffers :

C−x C−f	Charge un fichier dans la fenêtre active avec création d'un nouveau buffer	find−file
C−x 4 f	Comme C−x C−f avec création d'une nouvelle fenêtre	find−file−other−window
C−x C−v	Remplace le fichier du buffer courant	find−alternate−file
C−x C−s	Enregistre le buffer courant	save−buffer
C−x C−w	Enregistre le buffer courant sous un autre nom	write−file
C−x C−u	Liste les buffers * pour tuer un buffer : mettre un "d" en face du nom et valider la sélection en tapant "x". * pour activer un buffer: mettre un "f" en face du nom.	list−buffers
C−x k	Tue le buffer spécifié	kill−buffer
C−x d	Liste le contenu d'un répertoire dans la fenêtre active	dired
C−x 4 d	Comme C−x d avec création d'une nouvelle fenêtre	dired−other−window

M−x revert−buffer Revient à la dernière version enregistrée.
M−x c−mode Pour choisir le mode d'édition en C (idem pour tex−mode, ...)

Trouver de l'aide :

C−h a chaîne	Affiche les commandes dont le nom contient la chaîne	command−apropos
C−h b	Affiche une table des commandes	describe−bindings
C−h f fonction	Affiche des renseignements sur la fonction	describe−function
C−h k clé	Affiche des renseignements sur la commande associée à la clé	describe−key

Merci de signaler d'éventuelles erreurs à Christian.Labesse@math.u−bordeaux.fr

Emacs et langage C

Éditer : le mode C

Le mode C d'*Emacs* offre des facilités pour écrire les programmes. Ce mode est activé automatiquement quand on édite un fichier dont le nom se termine par .c ou .h .

TAB	Aligne correctement le texte de la ligne courante	c−indent−command
C−M−q	Au début d'un groupe () ou {}, aligne correctement le texte à l'intérieur du groupe	c−indent−exp

M−x c−indent−line−or−region Aligne correctement le texte dans la zone sélectionnée

Compiler : le compilateur gcc

M−x compile Lance une compilation dans le répertoire courant ; par défaut la commande **make** est appelée, mais on peut indiquer une autre commande

C−x ' Après une commande **M−x compile**, positionne le curseur sur la première erreur de compilation. Renouvelable.

M−x recompile Relance la dernière commande de compilation

Déboguer : le débogueur gdb

En lançant le débogueur sous *Emacs*, on peut suivre automatiquement dans le fenêtre d'édition les lignes de commandes qui s 'exécutent dans le débogueur.
Ne pas oublier de compiler avec **l'option −g** de gcc.

`M−x gdb`	Lance le débogueur

Une fois le débogueur lancé, on peut charger un programme et poser un point d'arrêt sur une fonction (par exemple *main()*), puis on lance l'exécution du programme. Le source C apparaît alors dans une autre fenêtre.

`(gdb) file nom_de_programme`	Charge le fichier binaire à déboguer
`(gdb) break nom_de_fonction`	Pose un point d'arrêt à l'entrée de la fonction
`(gdb) run [arguments]`	Lance le programme avec d'éventuels arguments

Quand le programme est arrêté , par exemple à un point d'arrêt, on peut utiliser les commandes :

`(gdb) cont`	Continue le programme après un arrêt
`(gdb) next`	Exécute le programme jusqu'à la ligne du source suivante, sans rentrer dans les fonctions appelées
`(gdb) step`	Idem en rentrant dans la(les) fonction(s) appelée(s)
`(gdb) nexti`	Exécute l'instruction suivante, sans rentrer dans les fonctions appelées
`(gdb) stepi`	Idem en rentrant dans la(les) fonction(s) appelée(s)
`(gdb) print expr`	Affiche la valeur de l'expression C donnée en argument
`(gdb) where`	Affiche les appels de fonctions actifs, c−à−d le contenu de la pile

Make et Makefile

Dominante Informatique et Filière Eurecom

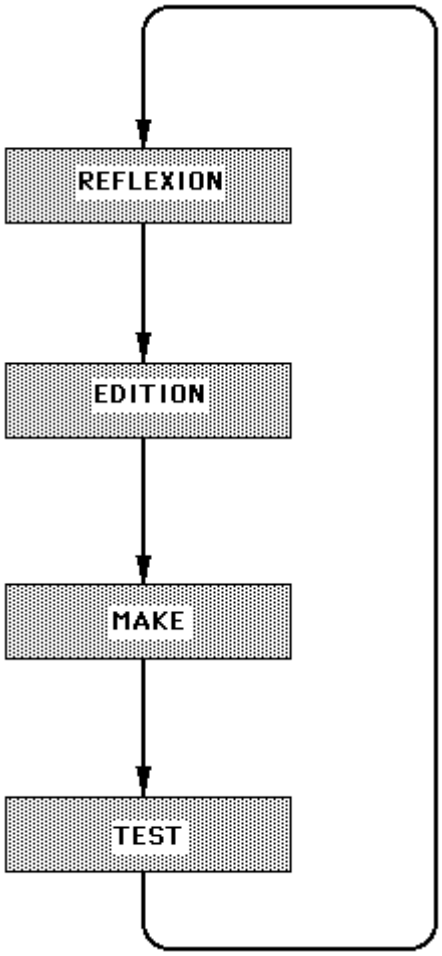
Philippe Dax

- Utilité et domaine d'utilisation
- Connaissances indispensables pour l'utiliser
- Connaissances intéressantes (grande efficacité)
    - macros de substitution
    - macros spéciales
    - règles implicites et explicites
    - directives
    - les options de la commande make
- Utilisation non conventionnelle

Utilité et domaine d'utilisation

- Composants d'une application
    - plusieurs excécutables
    - plusieurs bibliothèques
    - plusieurs sources
    - plusieurs fichiers include
- Solutions de mise à jour de l'application
    - *minimiser* le travail du calculateur, mais *augmenter* le travail du programmeur
    - *minimiser* le travail du programmeur, mais *augmenter* le travail du calculateur
    - *minimiser* le temps du calculateur, et *minimiser* le travail du programmeur

Schéma de production d’une application



Algorithme de make

- Format du fichier de description (Makefile, makefile)

```
cible: liste de dØpendance
   <tab>  action
```

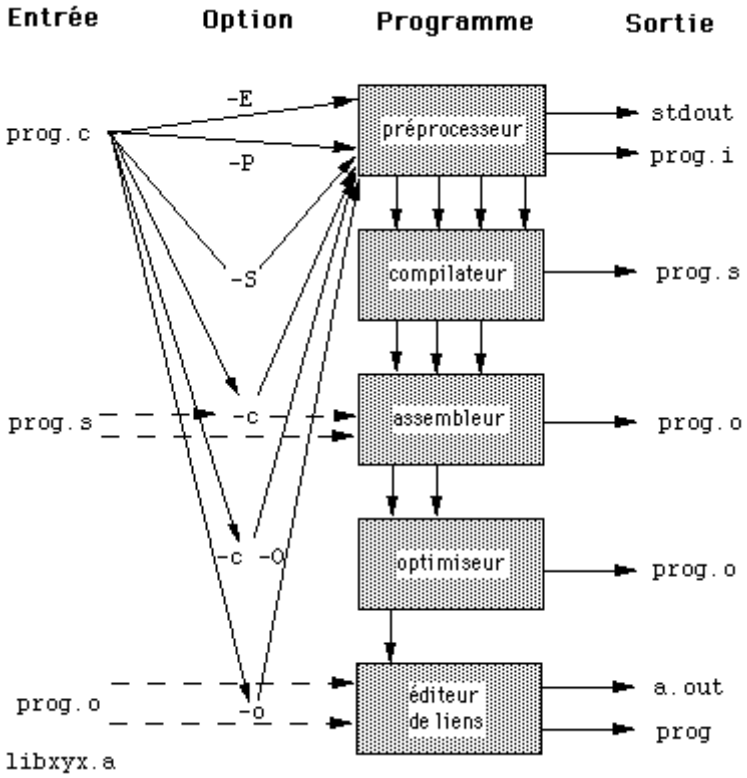
si (la **cible** est plus anciene qu’une seule **dépendance**)
ou
(la **cible** n’existe pas et que les **dépendances** existent)
alors
exécuter **action**

- Algorihme de make :
  − *make* construit un graphe de précédence pour la cible finale

  − *make* utilise l’algorithme de première profondeur pour visiter chaque noeud du graphe de précédence

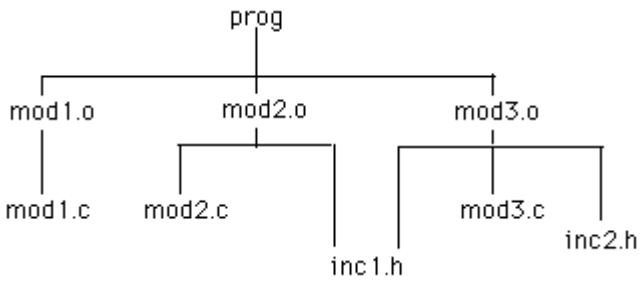
  − Pour chaque noeud, si les prérequis sont plus récents que le noeud cible, *make* exécute les commandes du noeud.

Actions de compilation



Graphe de dépendance

- Graphe de dépendance d'une application



- Makefile

```
#
# Makefile
#
prog:   mod1.o mod2.o mod3.o
    cc mod1.o mod2.o mod3.o -o prog
mod1.o: mod1.c
    cc -c mod1.c
mod2.o: mod2.c inc1.h
    cc -c mod2.c
mod3.o: mod3.c inc1.h inc2.h
    cc -c mod3.c
```

Macros de substitution

- Définition d'une macro

  − *Nom_macro* = chaîne de caractères

- Interprétation d'une macro

  − **$(***Nom_macro***)**

- Makefile

```
#
# Makefile
#
OBJS = mod1.o mod2.o mod3.o
prog:   $(OBJS)
        cc $(OBJS) −o prog
mod1.o: mod1.c
        cc −c mod1.c
mod2.o: mod1.c inc1.h
        cc −c mod2.c>
mod3.o: mod3.c inc1.h inc2.h
        cc −c mod3.c
```

Connaissances implicites

- Règles implicites connues de make

  - dépendance
  - inférence (action par défaut)
  - précédence

- Makefile

```
#
# Makefile  −  (Exemple de dépendance implicite)
#
OBJS = mod1.o mod2.o mod3.o
prog:   $(OBJS)
        cc $(OBJS) −o prog
mod2.o: inc1.h
mod3.o: inc1.h inc2.h
```

Macros spéciales

- **$@** nom de la cible à reconstruire
- **$*** nom de la cible sans suffixe
- **$<** nom de la dépendance à partir de laquelle on reconstruit la cible
- **$?** liste des dépendances plus récentes que la cible

- Makefile

```
#
# Makefile  –  (Exemple d’utilisation des macros spéciales)
#
OBJS = mod1.o mod2.o mod3.o
prog:   $(OBJS)
        cc $(OBJS) –o $@
        @echo "On reconstruit $@ a cause de $?"
mod1.o: mod1.c; cc –c $<
# mod1.o: mod1.c; cc –c $*.c
mod2.o: inc1.h
mod3.o: inc1.h inc2.h
```

Règles de précédence et d’inférence

- Règle de précédence

  – .SUFFIXES: .o .s .c .f .y

- Règle d’inférence (construction des cibles)

```
.c.o:
        cc –c $<
```

- Makefile

```
#
# Makefile  –  (Exemple de règle d’inférence)
#
OBJS = mod1.o mod2.o mod3.o
.c.o:
        $(CC) $(CFLAGS) –c $<
prog:   $(OBJS)
        $(CC) $(OBJS) –o $@
mod2.o: inc1.h
mod3.o: inc1.h inc2.h
```

Directives

- SUFFIXES règle de précédence

  – .SUFFIXES: .o .s .a .y

- .SILENT pas d’écriture sur la sortie standard stdout
- .IGNORE ignore les erreurs et poursuit
- .DEFAULT action par défaut en cas d’erreur
- .PRECIOUS inhibition des interruptions

- Makefile

```
#
# Makefile
#
.PRECIOUS mod2.o
.DEFAULT:; echo "$< n’existe pas"
OBJS = mod1.o mod2.o mod3.o
prog:   $(OBJS)
        $(CC) $(OBJS) –o $@
mod2.o: inc1.h inc3.h
mod3.o: inc1.h inc2.h
```

Exemple d’un Makefile générique

```
#
# Makefile − maintained by author − date
#
APPL = prog

SHELL = /bin/sh
PRINTER = dominique
COMPRESS = gzip

# Paramètres de compilation et d’édition de liens
CC = gcc
LD = $(CC)
CFLAGS = −g
LDFLAGS = −static
DEFS = −DENST −DPATH=\"/usr/local/$(APPL)\"
# Paramètres d’installation
BINDIR = /usr/local/bin
MANDIR = /usr/local/man
MANEXT = 1

# Objets de l’application
LIBS =
OBJS = mod1.o mod2.o mod3.o
SRCS = mod1.c mod2.c mod3.c
INCL = inc1.h inc2.h
MANS = $(APPL).1
SOURCES = README COPYRIGHT INSTALL Makefile $(SRCS) $(INCL) $(MANS)
EXEC = $(APPL)

# Règle d’inférence
.DEFAULT:; @echo "$< n’existe pas"
.c.o:
        −$(CC) $(DEFS) $(CFLAGS) $<

# Construction de l’application
.PRECIOUS $(EXEC)
$(EXEC): $(OBJS)
        $(LD) $(LDFLAGS) $(OBJS) $(LIBS) −o $@

clean:
        −@rm $(OBJS) $(EXEC) core a.out 2>/dev/null
install:
        install −m 755 −o root −g staff $(EXEC) $(BINDIR)
install_man:
        install −m 444 $(EXEC).$(MANEXT) $(MANDIR)/man$(MANEXT)
print: $(SRCS) $(INCL)
        −@lpr −P$(PRINTER) $?
        −@touch $@
printall:
        −@lpr −P$(PRINTER) $(SOURCES)
tar:
        tar cvf $(APPL).tar $(SOURCES)
archive: tar
        $(COMPRESS) $(APPL).tar

#
# Dépendances des objets
#
mod2.o: inc1.h inc3.h
mod3.o: inc1.h inc2.h
```

La commande make et ses options

- make [options] [définition macros] [cibles]
    - make mod1.o mod2.o
    - make "EXEC=application" "CC = cc"
    - make −f appli.mk
    - make −n (mise au point du makefile)
    - make −ts (touch et .SILENT)
    - make −p (liste des macros)
    - make −i (.IGNORE)
    - make −d (debug interne à make)

Génération automatique de Makefile

- genmake
    - construction automatique d'un Makefile pour une seule cible à partir des sources C et des include dans le répertoire courant du projet.
- makedepend
    - construction automatique des dépendances des fichiers include qui seront rajoutées ou modifiées en fin du Makefile.
- imake, xmkmf
    - imake : utilisation d'un fichier modèle (template) Imakefile.
    - xmkmf : script spécifique à X11 utilisant imake pour adapter les PATHS de X11.
    - fichier Imakefile.
- autoconf, automake, configure
    - adaptation automique au système d'exploitation, aux différents compilateurs, aux bibliothèques (recherche de la présence des fonctions) et utilitaires présents sur le système (yacc/bison, cc/gcc, lex/flex, awk/gawk,...).
    - fichiers Makefile.in (template), configure.in (paths).
    - très employé dans les distributions GNU.