Introduction à UNIX et à la programmation Shell

Sylvain BAUDRY

ESME-Sudria[1]
Département Informatique
38 rue Molière 94200 IVRY SUR SEINE

Version 4.0.1

Septembre 2007

1. Tél: +33 (0) 56 20 62 00

Préface

L'ensemble de ce document a été réalisé avec le *package* LaTeX 2_ε.

TeX est une marque déposée de l'*American Mathematical Society*.

Unix est une marque déposée d'Unix *System Laboratories*.

PostScript est une marque déposée d'*Adobe Corp*.

MS-DOS, Windows et Windows–NT sont des marques déposées de *Microsoft Corp*.

Macintosh et MacOS sont des marques déposées d'*Apple Computer*.

VT, Vax, Vax/Vms, OpenVms et Digital Unix sont des marques déposées de *Digital Equipment Corp*.

HP–UX est une marque déposée de *Hewlett-Packard*.

SunOS et Solaris sont des marques déposées de *Sun Microsystems*.

NFS, *NIS* et *NIS+* sont des marques déposées de *Sun Microsystems*.

Irix est une marque déposée de *Silicon Graphics Inc*.

AIX, SNA et RS/6000 sont des marques déposées de *International Business Machine*.

Netscape Navigator est une marque déposée de *Netscape Corp*.

Table des matières

I Introduction à Unix **1**

1 Concepts de base sous Unix **3**

1.1 Notions générales 3

1.2 Connexion et déconnexion 5

1.3 Format d'une commande 6

1.4 Le manuel Unix 7

1.4.1 Introduction, les sections 7

1.4.2 Format d'une page du manuel 7

1.4.3 La commande `man` 8

1.4.3.1 Equivalences 9

1.5 Introduction à la notion de *file system* 9

1.5.1 Structure arborescente 9

1.5.2 Les chemins d'accès 10

1.5.3 Principaux répertoires Unix 11

1.6 Les entrées/sorties 12

1.7 Les filtres 13

2 Commandes Unix **15**

2.1 Commandes liées au file system 15

2.1.1 Commandes `pwd` et `cd` 15

2.1.2 Commande `ls` 15

2.1.3 Commandes `mkdir` et `rmdir` 17

2.1.4 Répertoires `"."` et `".."` 17

2.2 Commandes de manipulation de fichiers 18

2.2.1 Attributs d'un fichier 18
2.2.2 Affichage du contenu d'un fichier - Commandes `cat` et `more` 19
2.2.3 Manipulation de fichiers - Commandes `cp`, `mv` et `ln` 19
2.2.3.1 Visualisation du nombre de liens avec la commande `ls` 21
2.2.4 Effacement d'un fichier - Commande `rm` 22
2.3 Protections sur les fichiers 23
2.3.1 Notion d'identité sous UNIX 23
2.3.2 Permissions 24
2.3.3 Changement de protection - Commande `chmod` 25
2.3.4 Remarques sur les protections 27
2.4 Les filtres 28
2.4.1 Rappels, Propriétés 28
2.4.2 Filtres déjà vus 28
2.4.3 Filtre `sort` 29
2.4.4 Filtre `grep` 29
2.4.5 Filtre `wc` 30
2.4.6 Filtre `cut` 30

3 Commandes usuelles de communication réseau 33
3.1 Connexion à une autre machine – commande `telnet` 33
3.2 Transfert de fichiers - commande `ftp` 34
3.3 Connexion automatique – commande `rlogin` 37
3.4 Transfert de fichiers automatique – commande `rcp` 38
3.5 Exécution d'une commande à distance - commande `rsh` (ou `remsh`) 40
3.5.1 Comparaisons `fp` telnet/`fp` rlogin et `fp` ftp/`fp` rcp 41

II Introduction au shell 43

1 Notions élémentaires du Bourne Shell 45
1.1 Introduction 45
1.2 Zones mémoire code, variables locales, variables d'environnement du shell 46
1.2.1 Description 46

1.2.2 Les commandes de gestion des variables du shell 48
1.2.3 Variables usuelles 48
1.2.4 Visualisation d'une variable 49
1.3 Exécution d'une commande 49
1.4 Redirection des entrées/sorties 49
1.4.1 Introduction 49
1.4.2 Redirection de l'entrée standard (`fp stdin`) 50
1.4.3 Redirection de la sortie standard (`fp stdout`) 50
1.4.4 Redirection de la sortie d'erreurs standard (`fp stderr`) . . 50
1.4.5 Redirection d'une sortie standard vers une autre sortie standard 51
1.4.6 Redirection de la sortie standard d'une commande dans l'entrée standard d'une autre 53
1.5 Génération de noms de fichiers - Les métacaractères 54
1.5.1 Introduction 54
1.5.2 Utilisation du métacaractère "`fp?`" 54
1.5.3 Utilisation des métacaractères "`fp []`" 55
1.5.4 Utilisation du métacaractère "`fp *`" 55
1.6 Les quotes et les caractères spéciaux 55
1.6.1 Introduction 55
1.6.2 Les caractères d'échappements 56
1.6.3 Résumé 56

2 Le mode multi-tâche **59**
2.1 Tâche de fond – le *background* 59
2.2 Substitution de commande - caractère *back quote* 60
2.3 Commandes associées 60
2.3.1 Commande "`kill`" 60
2.3.2 Commande "`wait`" 61
2.3.3 Les commandes "`fg`" et "`bg`" 61
2.3.4 Commandes "`at`" 62
2.4 Répétition de tâches : `crontab` 65
2.4.1 Introduction – Syntaxe 65
2.4.2 Fichier de description de tâches 66

2.4.3 Exemple de fichier de description 69

III Introduction à la programmation Bourne Shell 71

1 Introduction 73

1.1 Définition 73

1.2 Historique des shells 73

1.3 Quelques règles et recommandations pour l'écriture des shells-scripts 75

1.3.1 Règles à observer 75

1.3.2 Recommandations 76

2 Les arguments des programmes shell 79

3 Les variables 81

3.1 Les variables spéciales 81

3.2 Manipulation sur les variables 82

4 Commandes évoluées pour les scripts 85

4.1 La commande "`fp shift`" 85

4.2 La commande "`fp read`" 85

4.3 La commande "`fp expr`" 86

5 Les listes de commandes et les fonctions 89

5.1 Les listes de commandes 89

5.2 Les fonctions 90

5.2.1 Déclaration et utilisation des fonctions 90

5.2.2 Commande "`fp return`" 92

6 Les commentaires et les techniques d'exécution 93

6.1 Les commentaires 93

6.2 Interprétation spéciale du signe "`fp #`" sur la première ligne d'un shell script 93

6.2.1 Les appels d'un shell script au niveau de la ligne de commandes 94

7 Les tests et les boucles **97**

7.1 Les tests . . . 97

7.1.1 La commande « fp test » . . . 97

7.1.2 Tests sur les fichiers . . . 97

7.1.3 Tests sur les chaînes de caractères . . . 98

7.1.4 Les test numériques . . . 99

7.1.5 Autres opérations . . . 99

7.2 La construction « fp if » . . . 99

7.3 La construction « fp case » . . . 100

7.4 La boucle « fp while » . . . 101

7.5 La boucle « fp until » . . . 101

7.6 La boucle « fp for » . . . 101

7.7 Les commandes « fp break », « fp continue » et « fp exit » . . . 102

7.8 Signaux et traps . . . 103

7.8.1 Définition des signaux et des traps . . . 103

7.8.1.1 Les signaux . . . 103

7.8.1.2 Les *traps* . . . 103

7.8.1.3 La commande « fp trap » . . . 104

8 Les expressions régulières **105**

8.1 Introduction - Définition . . . 105

8.2 Spécification de caractères particuliers . . . 105

8.3 Exemples d'application . . . 106

8.4 Recherche de caractères suivant un nombre d'occurrences . . . 106

9 Utilisation avancées de certains filtres **109**

9.1 Introduction . . . 109

9.2 Utilisation avancée de "grep", "egrep" et "fgrep" . . . 109

9.2.1 Introduction - Rappels . . . 109

9.2.2 Utilisation de "grep" . . . 110

9.3 Utilisation de "egrep" . . . 111

9.4 Utilisation de "fgrep" . . . 112

9.5 Remarque sur l'utilisation de l'option "-f" . . . 113

10 Utilisation de "sed" 115

10.1 Introduction . . . 115
10.2 Mode de fonctionnement . . . 116
10.3 Formulation des requêtes . . . 116
10.3.1 Introduction . . . 116
10.3.2 Définition des adresses . . . 116
10.3.3 Les Commandes . . . 117
10.3.4 Les symboles particuliers . . . 119
10.4 Exemples avancés . . . 121
10.4.1 Exemple 1 . . . 121
10.4.2 Exemple 2 . . . 123
10.4.3 Exemple 3 . . . 124
10.4.4 Exemple 4 . . . 126
10.4.5 Exemple 5 . . . 128
10.5 Remarque sur l'utilisation de l'option "-f" . . . 129

11 Utilisation de "awk" 131

11.1 Introduction . . . 131
11.2 Les sélecteurs . . . 132
11.2.1 Introduction, Définition . . . 132
11.2.2 Les sélecteurs prédéfini . . . 132
11.2.3 Les règles de sélection . . . 133
11.2.4 Les caractères spéciaux pour la sélection . . . 133
11.2.5 Les expressions logiques pour la sélection . . . 134
11.2.6 Syntaxe des sélecteurs . . . 134
11.3 Les variables . . . 135
11.3.1 Les tableaux . . . 137
11.4 Les actions . . . 137
11.4.1 Les fonctions prédéfinies . . . 138
11.4.2 Les fonctions utilisateur . . . 140
11.4.3 Les structures de contrôle . . . 141
11.5 Trucs et astuces . . . 142
11.5.1 Commentaires . . . 142

11.5.2 Passage de paramètres du shell vers le programme "fp awk" 142
11.5.3 Utilisation de variables du Shell dans le corps du programme "fp awk" . . . 143
11.6 Exemple . . . 144
11.7 Remarque sur l'utilisation de l'option "fp -f" . . . 145

12 Programmation avancée de Shell Scripts 147
12.1 Introduction . . . 147
12.2 Tri par adresse IP du fichier "/etc/hosts" . . . 147
12.2.1 Étude des fonctionnalités . . . 147
12.2.2 Méthode utilisée . . . 148
12.2.3 Développement . . . 149
12.2.4 Programme obtenu . . . 150
12.3 Recherche des UID et GID disponibles . . . 151
12.3.1 Étude des fonctionnalités . . . 151
12.3.2 Méthode utilisée . . . 151
12.3.3 Développement . . . 152
12.3.4 Programme obtenu . . . 157
12.4 Traduction de fichiers d'informations . . . 159
12.4.1 Étude des fonctionnalités . . . 159
12.4.2 Méthode utilisée . . . 161
12.4.3 Développement . . . 162
12.4.3.1 Détermination des entrées de référence . . . 165
12.4.3.2 Extraction des anciennes entrées et mise-à-jour . . . 166
12.4.3.3 Création des nouvelles entrées utilisateur . . . 167
12.4.3.4 Suppression des répertoires inutiles . . . 169
12.4.3.5 Création des nouvelles entrées "*projet*" . . . 170
12.4.3.6 Création des répertoires utilisateurs et "*projet*" 171
12.4.4 Programmes obtenus . . . 172
12.4.4.1 Fichier "mkpasswd.define" . . . 173
12.4.4.2 Fichier "mkpasswd.functions" . . . 176
12.4.4.3 Fichier "mkpasswd.check" . . . 177
12.4.4.4 Fichier "mkpasswd" . . . 179

IV Annexes **187**

A Instructions de formatage **189**

A.1 Exercices d'utilisation des expressions régulières 191

B Correspondances entre le Bourne Shell et le C Shell **193**

B.1 Manipulation des variables 193

B.2 Évaluation de variables 194

B.3 Expression arithmétiques 194

B.4 Variables formelles 196

B.5 Environnement 197

B.5.1 Empilement de variables 197

B.5.2 Variables d'environnement 197

B.6 Entrées/Sorties et redirections 198

B.6.1 Entrée en ligne 198

B.6.2 Séparation, regroupement des sorties (standard et d'erreurs) 199

B.7 Les fonctions 199

B.7.1 Les fonctions internes (*built in*) 199

B.7.2 Les fonctions externes 200

B.8 Les structures de contrôle 201

B.8.1 Les tests ("`fp` if") 201

B.8.2 Choix multiples (`fp` case, `fp` switch) 202

B.8.3 Les boucles 204

B.8.3.1 Boucle du type "`fp` for" 204

B.8.3.2 Boucle du type "`fp` while" 204

B.8.4 Tableaux et listes 205

B.8.5 Interruptions, déroutements sur signaux 206

B.8.6 Les retours 206

C Utilisation des éditeurs de texte sous UNIX **209**

C.1 Introduction 210

C.2 Commandes de base de l'éditeur `vi` 212

C.2.1 Introduction et conventions 212

C.2.2 Modes de fonctionnement de "`vi`" 213

C.2.3 Démarrage d'une session "vi" . . . 214
C.2.4 Sauvegarder et quitter "vi" . . . 214
C.2.5 Commandes d'état . . . 215
C.2.6 Insertion de texte . . . 216
C.2.7 Annuler ou répéter des commandes . . . 217
C.2.8 Déplacement du curseur . . . 218
C.2.9 Effacement de texte . . . 220
C.2.10 Recherche . . . 222
C.2.10.1 Expression de recherches . . . 222
C.2.10.2 Opérations de recherche . . . 224
C.2.10.3 Recherche globale et substitution . . . 224
C.2.11 Indentation de texte . . . 225
C.2.12 Copier/Coller . . . 226
C.2.13 Modifier du texte . . . 227
C.2.14 Placement direct du curseur et ajustement du texte à l'écran 229
C.2.15 Interaction avec le *shell* . . . 230
C.2.16 Macros et abréviations . . . 232
C.2.17 Commandes de configuration de "vi" . . . 233
C.2.18 Fichier d'initialisation de "vi" : ".exrc" . . . 239
C.3 Commandes de base de l'éditeur emacs . . . 241
C.3.1 Introduction . . . 241
C.3.2 Organisation de l'écran . . . 242
C.3.3 Clés de Fonction . . . 242
C.3.4 Notion de buffers . . . 243
C.3.4.1 Introduction . . . 243
C.3.4.2 Les "*mini-buffers*" . . . 244
C.3.5 Utilisation de l'aide . . . 244
C.3.6 Utilisation de quelques commandes . . . 245
C.3.6.1 Déplacement dans le "buffer" . . . 245
C.3.6.2 Recherche / Remplacement de caractères . . . 245
C.3.6.3 Réponses possibles pour la recherche et le remplacement de caractères . . . 245
C.3.6.4 Couper / Copier / Coller . . . 246

C.3.6.5 Opérations sur les fenêtres 247
C.3.6.6 Autres commandes diverses 248
C.3.7 Macros 248

D Licence associée à ce document **251**
D.1 Conventions et Notations 260

Première partie

Introduction à Unix

Chapitre 1

Concepts de base sous UNIX

1.1 Notions générales

UNIX est un système d'exploitation orienté fichiers contrairement à MS-DOS[1] ou à OpenVMS[2] qui sont plutôt orientés disque. Il n'existe pas sous UNIX, au niveau utilisateur, de notion de disques physiques. On ne voit qu'une seule arborescence dont différents points sont rattachés à un système de fichiers. Celui-ci peut correspondre physiquement à :

- une partition d'un disque physique (la partition peut correspondre soit à une partie ou bien à la totalité d'un disque physique) ;
- un ensemble de disques physiques, notion qu'il est possible d'utiliser soit dans un environnement RAID 1, soit à partir d'un volume logique (*Logical Volume*) dans l'environnement LVM[3], soit un *Meta Device* sous Sun Solaris ;
- un système de fichier distant résidant sur une autre machine du réseau (service NFS entre machines UNIX, ou service SMB/CIFS avec l'environnement Windows).

La figure 1.1 donne un aperçu des liaisons possibles entre certains points de l'arborescence et les disques physiques sur une machine UNIX.

Toutes les commandes UNIX distinguent les majuscules des minuscules. Par exemple « `ls` », « `Ls` » , « `lS` » et « `LS` » sont quatre mots différents au niveau de l'interpréteur de commandes.

Chaque commande UNIX est exécutée dans un process séparé. Lorsqu'on tape une commande au niveau de l'interpréteur de commandes, celui-ci crée un sous processus dans lequel il exécute la commande. Lorsque celle-ci est terminée, le sous process disparait en rendant la main au processus père.

1. MS-DOS= Microsoft Disk Operating System
2. VMS = Virtual Memory System, système de Digital Equipment
3. LVM: Logical Volume Manager. Cet outil, développé à la base par IBM pour ses systèmes AIX est présent maintenant sur la plupart des UNIX commerciaux et dans les environnements LINUX.

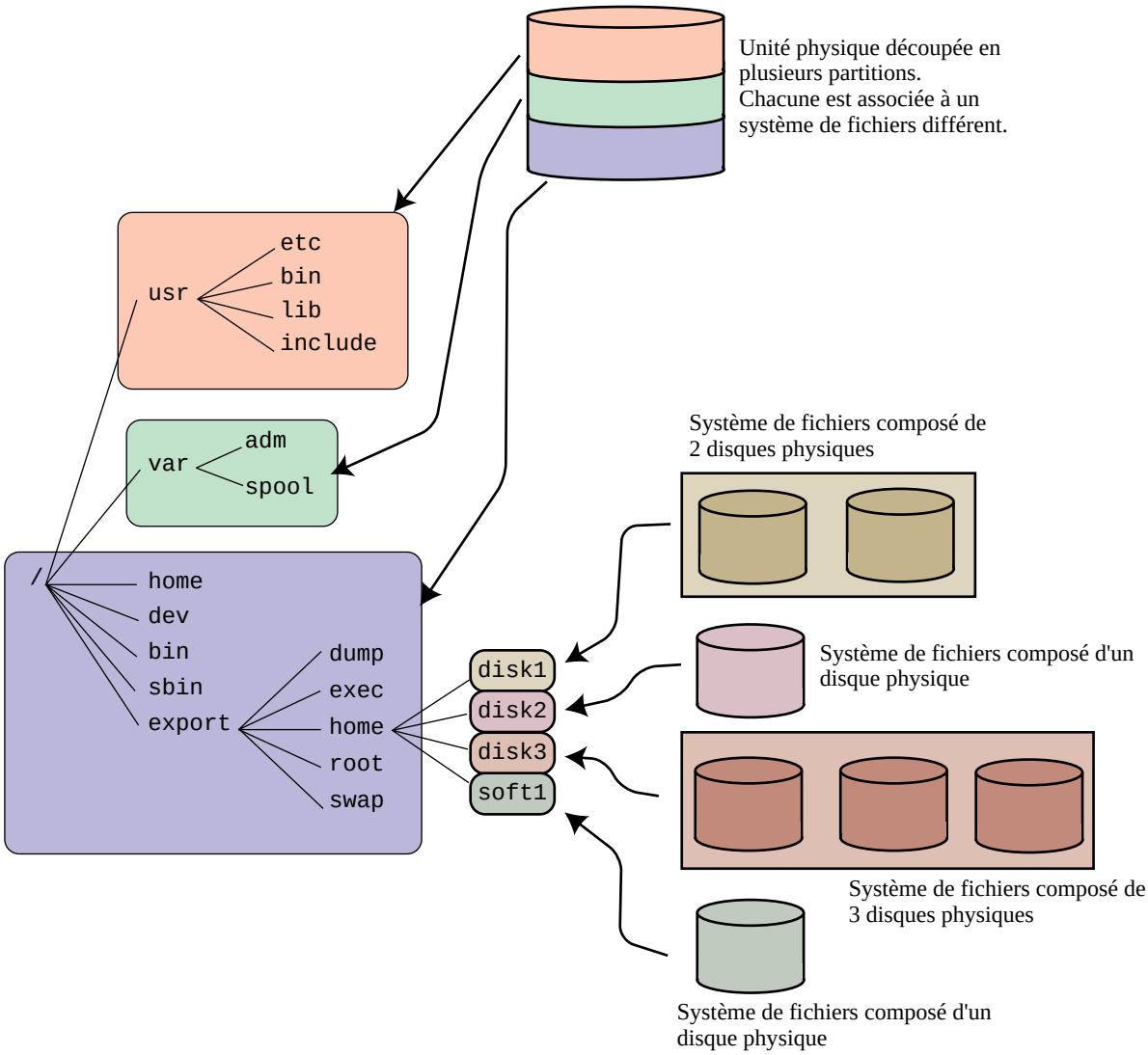


FIG. 1.1 – *Organisation de l'arborescence* UNIX *avec les systèmes de fichiers*

1.2 Connexion et déconnexion

Comme tout système multi-utilisateurs et multi-tâches, UNIX associe à chaque utilisateur :

- un nom appelé « *login* » (équivalent au « *Username* ». de OpenVMS, ou au « *logon name* » de Windows–NT) ;
- un mot de passe ;
- un numéro d'utilisateur unique ou « *UID*[4] » (équivalent à l'« *UIC* » OpenVMS ou « *logon ID* » de Windows–NT).

Par conséquent, la première chose que vous demandera le système sera votre « *login* » et votre mot de passe. Si les deux sont valides, UNIX initialisera votre environnement de travail.

Remarque 1.1 :

*Pour des raisons de sécurité, comme sous Open*VMS *et Windows–NT,* UNIX *ne verifie pas que le nom de « login » existe. Une fois que les deux informations seront saisies, c'est-à-dire nom de « login » et mot de passe, il ira chercher dans la base des utilisateurs s'il existe un enregistrement et un seul pour lesquelles ces deux informations sont exactes. Si cela, n'est pas le cas,* UNIX *affichera le message d'erreur suivant :*

```
Invalid login name.
```

sans autre forme de renseignements.

En fonction du type de poste de travail, c'est-à-dire terminal passif ou station de travail disposant d'un environnement graphique, la méthode de déconnexion du système varie.

Si votre environnement de travail est sur un terminal passif, la commande de déconnexion est :

```
exit
```

Dans le cas des stations de travail graphiques, cela dépend de l'interface sélectionnée. En effet, il en existe plusieurs types dont :

- « *CDE* » ou « Common Desktop Environment », environnement graphique développé à la base sur les stations Hewlett-Packard (*HP–VUE*) et répandue sur l'ensemble des UNIXs commerciaux comme *Solaris*, *Digital* UNIX, HP–UX, AIX, *Irix*, etc.
- « *fvwm95* », environnement régit par la « *Free Software Foundation* » livré de base avec LINUX, environnement dont la présentation ressemble à Windows95 (et supérieur) et Windows–NT 4 (ou supérieur). Les sources de cet environnement sont disponibles et libres d'accès. Il est donc possible de les mettre sous n'importe quel plateforme UNIX (voire même OpenVMS.
- « *KDE* », environnement régit par la « *Free Software Foundation* » livré de base avec LINUX, environnement dont la présentation est un savant mélange entre « *CDE* » et l'« *Active Desktop* » de *Microsoft Corp.*.

4. *UID* = User IDentifier

- « *gnome* », environnement régit par la « *Free Software Foundation* » livré de base avec les futures versions de LINUX[5], environnement orienté objet et intégrant roralement les ressources locales et distante sur le bureau.
- « *4Dwm* », environnement graphique disponible plus particulièrement sur les stations *Silicon Graphics Inc.*.
- etc.

Il est clair que chaque interface dispose d'une boite de dialogue permettant de terminer la session en cours. Nous de détaillerons pas ici la méthode de déconnexion pour chacune de ces interfaces.

Comme tout système où un mot de passe est associé à un utilisateur, il existe une commande pour le changer. Sous UNIX cette commande est « `passwd` ». Elle vous demandera l'ancien mot de passe, le nouveau mot de passe puis une confirmation.

En résumé :

exit	Déconnexion du système.
passwd	Changement du mot de passe.

Le tableau 1.1 donne un équivalent entre les systèmes UNIX etOpenVMS de *Digital Equipment*.

UNIX	OpenVMS
`exit`	`LOGOUT`
`passwd`	`SET PASSWORD`

TAB. 1.1 – *Équivalences* UNIX *et Open*VMS *pour la déconnexion et le changement de mot de passe.*

Remarque 1.2 :

Si vous tes connect distance sur le systme UNIX *grce la commande « `telnet` » (ou tout autre type de protocole similaire comme « LAT », le comportement obtenu est quivalent celui d'un terminal passif.*

1.3 Format d'une commande

Une commande est constituée d'un nom suivi d'arguments. Le séparateur entre chaque mot d'une commande peut être un ou plusieurs espaces ou tabulations.

Syntaxe :

5. RedHat Corp. prévoit d'intégrer ce type d'interface en standard dans ces prochaines versions.

```
% commande [options] [arguments] <CR>
```

Exemple 1.1 :

```
% banner HI
% ls -l shmoll lancelot DuLac
```

Remarque 1.3 :

Sous UNIX*, on fait la différence entre majuscules et minuscules. Par conséquent* `ls`*,* `Ls`*,* `lS` *et* `LS` *sont quatres termes différents. De même, si on demande, par exemple, de taper « `q` », il ne faudra pas taper « `Q` ».*

1.4 Le manuel UNIX

1.4.1 Introduction, les sections

Le manuel de référence est divisé en plusieurs sections. Chacune correspond à un sujet bien particulier.

Section 1 :Les commandes utilisateur.
Section 1m :Les commandes d'administration système.
Section 2 :Les appels systèmes (programmation).
Section 3 :Les librairies de sous-routines (programmation).
Section 4 :Les fichiers spéciaux.
Section 5 :Les formats des fichiers.
Section 6 :Liste des jeux.
Section 7 :Possibilités diverses.
Section 8 :Les commandes d'administration système.
Section 9 :Glossaire.

En général, on retrouve dans la documentation papier la copie conforme du manuel sur le système. Il est bon de savoir la section dans laquelle se trouve le manuel d'une commande. En effet, dans toute documentation, les références sont fournies sous le format suivant : « `cmd (n)` » où « `cmd` » est le nom de la commande et « `n` » le numéro de la section.

1.4.2 Format d'une page du manuel

Une page de manuel contient en général les 11 parties suivantes :

- *NAME*
- *SYNOPSYS* (Syntaxe)
- *HARDWARE DEPENDENCIES*
- *EXAMPLES*
- *FILES*
- *RETURN VALUE*
- *SEE ALSO*
- *DIAGNOSTICS*

- *BUGS*
- *WARNINGS*
- *AUTHOR*

Chacun de ces paragraphes décrit les points suivants :

NAME
: Contient le nom de la commande ainsi qu'une description succincte. Le texte de cette section est utilisé pour générer l'index général.

SYNOPSYS
: Indique comment la commande est libellée. Affiche en gras les caractères tels qu'ils doivent être exactement frappés au terminal. Les termes entre crochets (« `[]` ») sont optionnels. Les champs réguliers peuvent être remplis par des textes appropriés. Les parenthèses sont utilisées pour montrer que l'argument précédent peut être répété. S'il y a doute sur la description de *SYNOPSYS*, lisez *DESCRIPTION*.

DESCRIPTION
: Contient une description détaillée de ce que fait la commande.

HARDWARE DEPENDENCIES
: Signale les variantes d'exploitation selon le matériel.

EXAMPLES
: Certaines pages du manuel ont des exemples pas toujours limpides mais qui peuvent aider à mieux comprendre la commande.

FILES
: Tous les fichiers spéciaux que la commande utilise.

RETURN VALUE
: Décrit les valeurs en retour d'un appel programme.

SEE ALSO
: Renvoie à d'autres pages du manuel ou à une autre documentation pour de plus amples renseignements.

DIAGNOSTICS
: Si la commande fait ressortir des messages d'erreurs, ils sont explicités ici (en général !!!).

BUGS
: Décrit les bugs (eh oui). Souvent utilisé pour vous prévenir des problèmes potentiels dans l'utilisation de cette commande.

WARNINGS
: idem que *BUGS*.

1.4.3 La commande `man`

La commande « `man` » recherche dans les différentes sections (1,2,3,4,5,6,7,8,9,1m) le mot clef. Elle s'arrête sur le premier mot clef trouvé. Ainsi, par exemple, on verra systématiquement l'aide sur « `passwd (1)` » par la commande « `man passwd` »; pour voir « `passwd(4)` », il faudra taper la commande « `man 4 passwd` ». Une fois trouvée, elle affiche le contenu d'un fichier d'aide à l'écran en faisant de la pagination. Celui-ci correspond à la page de manuel associée.

Pour passer à la ligne suivante : taper sur « RETURN »
Pour passer à l'écran suivant : taper sur « SPACE »
Pour quitter : taper sur « Q » ou « q »

Syntaxe :

`% man [n] commande`
Où l'option « `n` » est le numéro d'une section

Remarque 1.4 :

Il est possible d'avoir une liste de commandes concernées par un mot clef de deux façons :

`% man -k mot-clef` *(*`k` *comme keyword)*

ou bien

`% apropos mot-clef`

Ceci ne fonctionnera que si l'administrateur a constitué la base de données référençant les mots clefs dans le manuel (fichier `whatis`*).*

1.4.3.1 Equivalences

Le tableau 1.2 donne un équivalent entre les systèmes OpenVMS de *Digital Equipment*, MS-DOS et UNIX.

UNIX	OpenVMS	MS-DOS
`man`	`HELP`	`HELP`
`apropos`	`HELP`	`HELP`

TAB. 1.2 – *Équivalences OpenVMS, MS-DOS et* UNIX *pour accéder à l'aide*

1.5 Introduction à la notion de *file system*

1.5.1 Structure arborescente

Nous avons vu précédemment qu'il n'y a plus de notions de disques sous UNIX. Il en est de même pour les périphériques. De façon générale, tout est fichier. On ne voit donc, au niveau utilisateur, qu'une seule arborescence constituée de répertoires et de fichiers décrivant les ressources du système (cf. figure 1.2). Le nom d'un fichier sous UNIX est une suite de caractères. Il n'existe pas de notion de type de fichier et de numéro de version comme sur OpenVMS ou MS-DOS. Le caractère « . » est considéré comme étant un caractère dans le nom de fichier et non pas, comme sur OpenVMS ou MS-DOS, le caractère de séparation entre le nom du fichier et son type. Il est donc possible d'avoir un fichier dont le nom comporte plusieurs « . ». La philosophie des noms de fichiers ressemblerait donc à celle de l'environnement MacOS.

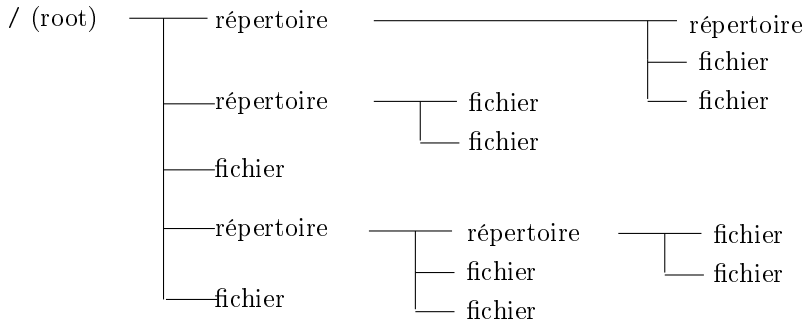


FIG. 1.2 – *Structure arborescente du système* UNIX

Remarque 1.5 :

Certains caractères sont à éviter dans les noms de fichiers tels que :

- *l'espace ;*
- *la tabulation ;*
- *le caractère « ; » ;*
- *tout caractère de contrôle (*ESC , CTRL–*H*, CTRL–*C*, *etc.).*

Remarque 1.6 :

Les noms de fichiers tiennent compte des majuscules/minuscules. Par exemple : « `SIMPSONS` » et « `simpsons` » sont deux fichiers différents.

1.5.2 Les chemins d'accès

On distingue deux types de chemins d'accès aux fichiers :

- les chemins d'accès absolus ;
- les chemins d'accès relatifs.

Les **chemins d'accès absolus** spécifient un nom de fichier ou de répertoire à partir de la racine. Les chemins d'accès absolus commencent donc par « / » qui est le nom de la racine ou du *root directory*.

Les **chemins d'accès relatifs** spécifient un nom de fichier ou de répertoire à partir du répertoire courant, appelé aussi *working directory*. Par conséquent, ces chemins ne commencent pas par « / » (exemple : `dir/essai`).

En résumé :

Chemins d'accès absolus :

- Commencent par « / » ;
- Représentent la position relative par rapport au *root directory*.

Chemins d'accès relatifs :

- Ne commencent pas par « / »,
- Représentent la position relative par rapport au répertoire courant.

Exemple 1.2 :

système	déplacement relatif	déplacement absolu
UNIX	`cd ../repertoire`	`cd /home/users/schmoll`
OpenVMS	`set default [-.repertoire]`	`set default disk$users:[schmoll]`
MS-DOS	`CD ..\REPERT`	`CD C:\USERS\SCHMOLL`

1.5.3 Principaux répertoires UNIX

Root directory (« / »)
C'est le haut de l'arborescence. Il n'y a qu'une et une seule entrée sur le file system. Le *root directory* est le seul répertoire sans père. Tous les fichiers et chemins d'accès absolus ont le *root directory* dans le chemin d'accès.

Répertoires `/bin` *et* `/sbin`
Les répertoires `/bin` et `/sbin` contiennent les commandes de base UNIX de la section 1 (`ls`, `cp`, `rm`, `mv`, `ln`, etc.) utilisées entre autre lors du démarrage du système. Les fichiers contenus dans ces répertoires ne sont que des exécutables.

Répertoire `/dev`
Le répertoire `/dev` contient les fichiers spéciaux permettant de communiquer avec tous les périphériques comme les disques, les terminaux, les dérouleurs de bandes, les imprimantes, etc.

Répertoire `/etc`
Le répertoire `/etc` contient tous les fichiers d'administration et un certain nombre de commandes système. Le répertoire `/etc` porte bien son nom[6].

Répertoires `/home` *et* `/root`
Le répertoire `/home` contient les répertoires personnels des utilisateurs du système.
Le répertoire `/root` est le répertoire personnel de l'administrateur.

Répertoire `/usr` *et sous-répertoires*
Le répertoire `/usr` contient un certain nombre de sous-répertoires. Chacun contient les fichiers nécessaires au fonctionnement en mode normal du système[7]. Par exemple :

- `/usr/bin` et `/usr/sbin` contiennent les commandes supplémentaires non contenues dans `/bin` et `/sbin` ;
- `/usr/lib` contient les bibliothèques pour le fonctionnement d'UNIX et pour le développement ;
- `/usr/include` contient les fichiers de déclaration des fonctions système pour le développement.

6. `etc` a vraiment été choisi pour la signification de *et caetera*
7. mode multi-utilisateurs

Répertoire `/var`
: Le répertoire `/var` contient les fichiers variables (succeptibles d'être modifiés fréquemment) : journaux (logs), e-mails, bases de données, archives, ...

1.6 Les entrées/sorties

A chaque création de processus, celui ci se voit affecté trois canaux de communication :

- l'entrée standard ;
- la sortie standard ;
- la sortie d'erreurs standard.

La figure 1.3 illustre les trois canaux de communications que possède chaque processus sur le système UNIX.

FIG. 1.3 – *Entrées/Sorties d'un processus*

Chacun des trois canaux se voit affecter un nom de fichier et un numéro :

Canal de communication	Fichier	Numéro logique
Entrée standard	`stdin`	0
Sortie standard	`stdout`	1
Sortie d'erreurs standard	`stderr`	2

Le fichier `stdin` (*entrée standard*), est le fichier à partir duquel le process va lire les données nécessaires en entrée. Il est ouvert avec le numéro logique 0(*file descriptor C*), et il est, par défaut associé au clavier.

Le fichier `stdout` (*sortie standard*), est le fichier dans lequel le process va écrire les messages qu'il produit en sortie, dans le cas d'une exécution normale. Il est ouvert avec le numéro logique 1 (*file descriptor C*), et il est, par défaut associé à l'écran.

Le fichier `stderr` (*sortie d'erreurs standard*), est le fichier dans lequel le process va écrire les messages d'erreur. Il est ouvert avec le numéro logique 2 (*file descriptor C*), et il est par défaut associé à l'écran.

Remarque 1.7 :

On distingue toujours deux canaux de sorties (un pour les sorties normales et un pour les erreurs). En effet, si un process doit écrire

dans un fichier et qu'une erreur se produit (impossibilité d'écrire par exemple), il faut afficher un message sur un canal de sortie différent.

Le tableau 1.3 donne les équivalences des noms des canaux d'entrées/sorties entre les systèmes UNIX et OpenVMS de Digital.

UNIX	OpenVMS
`stdin`	`SYS$INPUT`
`stdout`	`SYS$OUTPUT`
`stderr`	`SYS$ERROR`

TAB. 1.3 – *Équivalences des noms des canaux d'entrées/sorties entre* UNIX *et Open*VMS

1.7 Les filtres

Un filtre est une commande UNIX devant effectuer une action sur les données en lecture à partir de l'entrée standard et afficher le résultat en écriture sur la sortie standard.

La figure 1.4 montre le principe des filtres sous UNIX.

FIG. 1.4 – *Principe des filtres sous* UNIX

Il est possible d'enchaîner plusieurs filtres les uns aux autres. Par contre, les process en début et en fin de chaînes ne doivent pas se comporter comme des filtres.

Chapitre 2

Commandes UNIX

2.1 Commandes liées au file system

2.1.1 Commandes `pwd` et `cd`

`pwd` = Print Working Directory
`cd` = Change Directory

Syntaxe :

`pwd`	Affiche le chemin d'accès absolu du répertoire courant.
`cd repertoire`	Change le répertoire courrant.

Exemple 2.1 :

```
% pwd
/home/users/bart
% cd /home/users/schmoll
```

Le tableau 2.1 donne une équivalence des commandes de déplacement dans l'arborescence entre UNIX et OpenVMS .

UNIX	OpenVMS
`cd`	`SET DEFAULT`
`pwd`	`SHOW DEFAULT`

TAB. 2.1 – *Équivalence des commandes de déplacement dans l'arborescence sur le système*

2.1.2 Commande `ls`

Syntaxe :

`ls [-ladFR] [fichier ...]`
Liste le contenu d'un répertoire.

La commande "`ls`" sans argument, liste les noms de fichiers (ou de répertoires) présents dans le répertoire courant. Cette commande, utilisée avec un nom de fichier comme argument, permettra de vérifier l'existence de celui-ci. Si l'argument utilisé est un nom de répertoire, "`ls`" en listera le contenu.

Il existe de très nombreuses options pour la commande "`ls`". La liste ci-dessous est les options les plus utilisées.

- `-l` affiche le type de fichier, les protections, le nombre de liens avec le fichier, le propriétaire, le groupe, la taille en octets, la date de dernière modification et le nom du fichier.
- `-F` ajoute un "`/`" après le nom de chaque répertoire, un "`*`" après chaque fichier possédant le droit d'exécution et un "`@`" après chaque fichier lien (cf. section 2.2.3).
- `-a` liste tous les fichiers y compris les fichiers cachés.
- `-R` liste les fichiers et les répertoires de façon récursive.
- `-d` ne descend pas dans un répertoire si le paramètre est un nom de répertoire.

Exemple 2.2 :

```
% ls -F
dir1/ fic1 fic2* fic3@
% ls -a
. .. .profile dir1 fic1 fic2 fic3
% ls -l
drw-rw-rw- 3 schmoll esme 24 Jul 25 10:00 dir2
-rw-r--r-- 1 schmoll esme 37 Jul 25 12:00 fic1
-rwxr-xr-x 1 schmoll esme 37 Jul 25 12:00 fic2
lrw-r--r-- 1 schmoll esme 37 Jul 25 12:00 fic3 -> /tmp/ficref
% ls -R
dir1 fic1 fic2 fic3
./dir1:
fic4 fic5
% ls -d dir1
dir1
```

Le tableau 2.2 montre les équivalences entre les commandes des systèmes UNIX, OpenVMS et MS-DOS pour afficher la liste des fichiers d'un répertoire.

UNIX	OpenVMS	MS-DOS
`ls`	`DIRECTORY`	`DIR`
`ls -l`	`DIRECTORY/FULL`	*N/A*
`ls -R`	`DIRECTORY [...]`	*Absent de* `COMMAND.COM`
`ls rep`	`DIRECTORY [.REP]`	`DIR \REP`
`ls -d rep`	`DIRECTORY REP.DIR`	`DIR REP`

TAB. 2.2 – *Équivalences entre systèmes pour afficher la liste des fichiers d'un répertoire*

2.1.3 Commandes `mkdir` et `rmdir`

Syntaxe :

`mkdir [-p] directory ...`	(make directory)
`rmdir directory ...`	(remove directory)

La commande `mkdir` permet de créer des répertoires. Lorsqu'un répertoire est créé, il possède automatiquement deux sous répertoires : `"."` et `".."` qui seront examinés plus loin. La commande `rmdir` permet de supprimer des répertoires. Les répertoires à supprimer doivent impérativement être vides (ils ne doivent contenir que les répertoires `"."` et `".."`). D'autre part, il est impossible de supprimer des répertoires qui se trouvent entre la racine et le répertoire courant. Chacune de ces deux commandes peut avoir plusieurs arguments. Les arguments de `mkdir` sont les noms de répertoires à créer. Les arguments de `rmdir` doivent être des noms de répertoires déjà existants. Dans les deux cas, on peut utiliser des chemins d'accès relatifs ou absolus.

Remarque 2.1 :

Lorsqu'on utilise la commande `rmdir` *avec plusieurs arguments,* `rmdir` *détruit les répertoires dans l'ordre dans lequel ils ont été précisés sur la ligne de commande. Par conséquent, si l'on veut en détruire plusieurs, l'ordre sur la ligne de commande a une importance.*

Exemple 2.3 :

```
% mkdir mondir
% mkdir mondir/sd1 mondir/sd2 mondir/sd1/sd11
% mkdir mondir/sd3/sd31
% rmdir mondir/sd2
% rmdir mondir/sd3/sd31 mondir/sd3
% rmdir mondir/sd1/sd11 mondir/sd1 mondir
```

Le tableau 2.3 montre les équivalences entre les commandes des systèmes Unix, OpenVms et MS-DOS pour la gestion des répertoires.

Unix	OpenVms	MS-DOS
`mkdir`	`CREATE/DIRECTORY`	`MD` ou `MKDIR`
`rmdir`	`DELETE`	`RD` ou `RMDIR`

Tab. 2.3 – *Équivalences entre systèmes pour la gestion des répertoires*

2.1.4 Répertoires `"."` et `".."`

Quand un répertoire est créé, le système génère automatiquement deux sous-répertoires représentant des liens vers le répertoire créé et le répertoire père :

- le répertoire `"."`= répertoire courant,
- le répertoire `".."` = répertoire père.

Le répertoire `".."` est très utile pour référencer ce qui se trouve au dessus du répertoire courant dans l'arborescence du file system. Ainsi il suffira d'utiliser `".."` dans un chemin d'accès relatif pour référencer le répertoire père.

Par exemple "`cd ..`" remonte d'un cran dans l'arborescence et "`more ../../fic`" liste le contenu d'un fichier deux niveaux au dessus dans l'arborescence. La figure 2.1 montre les liens entre les répertoires "`.`" et "`..`" et les répertoires fils et pères.

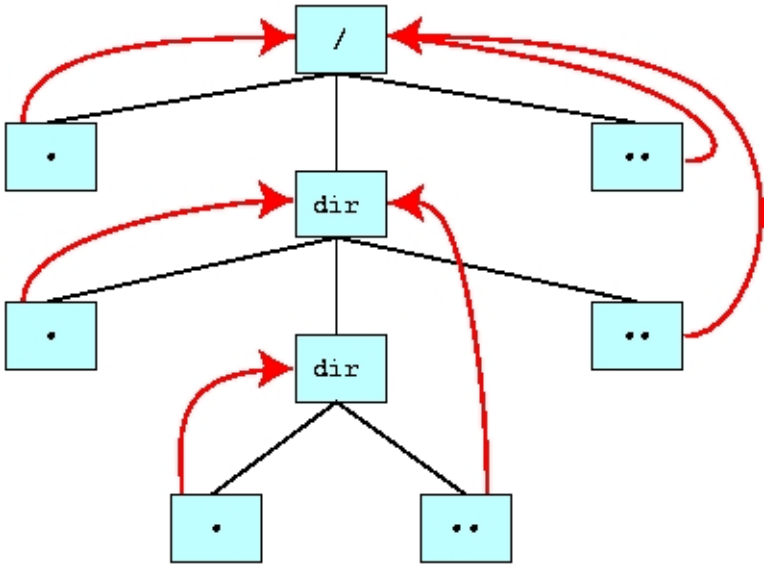


FIG. 2.1 – *Liens des répertoires "." et ".."*

Le tableau 2.4 montre des exemples d'équivalences entre les systèmes UNIX, OpenVMS et MS-DOS de manipulation des répertoires "`.`" et "`..`".

UNIX	OpenVMS	MS-DOS
`.`	`[]`	`.`
`..`	`[-]`	`..`
`../..`	`[-]`	`..\..`

TAB. 2.4 – *Exemples d'equivalences entre systèmes pour la manipulation des répertoires "." et ".."*

2.2 Commandes de manipulation de fichiers

2.2.1 Attributs d'un fichier

Par définition, un fichier est une suite d'octets possédant les attributs suivants :

- un type,
- un masque de protection,
- un nombre de liens avec d'autres fichiers,
- un propriétaire et groupe,

- une taille,
- une date de création et de dernière modification,
- un nom.

Les différents types de fichiers sont :

type	code
standard	`-`
répertoire	`d`
lien symbolique	`l`
fichier spécial mode block	`b`
fichier spécial mode caractère	`c`
fichier spécial mode réseau	`n`
pipe nommé	`p`

Le type "lien symbolique" correspond à un fichier spécial pointant physiquement sur un autre.

Les types "fichier spécial mode block" et "fichier spécial mode caractère" servent à communiquer avec les périphériques (disques, terminaux, etc.).

Le type "fichier spécial mode réseau" sert de canal de communication entre processes sur différentes machines.

Le type "pipe nommé" sert de canal de communication entre différents processes sur une même machine.

2.2.2 Affichage du contenu d'un fichier - Commandes `cat` et `more`

Syntaxe :

```
cat fichier...
more fichier...
```

La commande `cat` concatène le contenu des fichiers en arguments et affiche le contenu sur la sortie standard.

La commande `more` visualise le contenu des fichiers page écran par page écran. Pour visualiser la page suivante, il suffit de frapper sur SPACE , ou de frapper sur RETURN afin de visualiser une ligne supplémentaire. Pour terminer la visualisation avant la fin du fichier, taper sur la touche "`Q`" ou "`q`". D'autres commandes sont disponibles, pour cela taper sur la touche "`h`" ou "`H`" lorsque more a terminé d'afficher une page écran.

Le tableau 2.5 montre l'équivalence entre les commandes `cat` et `more` et d'autres systèmes (OpenVMS et MS-DOS).

2.2.3 Manipulation de fichiers - Commandes `cp`, `mv` et `ln`

Définition :

UNIX	OpenVMS	MS-DOS
`cat`	`type`	`TYPE`
`more`	`type/page`	`TYPE ... | MORE`

TAB. 2.5 – *Équivalence entre* `cat` *et* `more` *et d'autres systèmes*

`cp` copie les fichiers.
`mv` renomme les fichiers et répertoires et/ou déplace les fichiers.
`ln` crée un lien sur un fichier ou un répertoire.

Syntaxe :

```
cp [-i] fichier-source ··· destination
mv [-i] fichier-source ··· destination
ln [-s] fichier-source lien-destination
```

Lorsque le commande "`cp`" ne possède que deux ("`cp fichier1 fichier2`"), elle effectue une copie du fichier source vers le fichier destination. Si celui-ci existait déjà il est supprimé pour être remplacer par le nouveau.

Lorsque la commande "`cp`" possède plus de deux arguments (plusieurs fichiers source), la destination est obligatoirement un répertoire ("`cp fichier1 fichier2 repertoire`"). Dans ce cas, elle duplique ces fichiers dans le répertoire spécifié. S'il en existait déjà sous le même nom , ils sont supprimés pour être remplacé par les copies.

La commande "`mv`" réagit de façon similaire :

- si elle ne possède que deux arguments, elle renomme le fichier source sous le nouveau nom,
- si elle possède plusieurs arguments, la destination est obligatoirement un répertoire. Dans ce cas, elle déplace les fichiers sources dans le répertoire spécifié.

De même, si des fichiers existaient déjà sous le même nom, ils seront supprimés. Dans le cas particulier où la commande "`mv`" ne possède que deux arguments et que la destination est un nom de répertoire, le fichier source est déplacé à ce nouveau point de l'arborescence.

Les fichiers source de la commande "`cp`" ne peuvent pas être des répertoires.

Les fichiers source de la commande "`mv`" peuvent être de n'importe quel type.

La commande "`ln`" autorise l'accès à un fichier via plusieurs noms, ce sont des créations de liens entre fichiers. La syntaxe est :

```
ln fichier1 fichier2
```

- `fichier1` qui existe déjà, va pouvoir être accédé via le nouveau nom `fichier2`.
- `fichier2` est alors lié avec `fichier1`.

On distingue deux types de liens : les liens symboliques et les liens logiques.

– Dans le cas du lien symbolique, `fichier2` (fichier lien) pointe sur `fichier1` (fichier source) permettant d'accéder aux informations sur le disque. Par conséquent, si `fichier1` est effacé, le contenu est perdu et `fichier2` pointe sur quelque chose d'inexistant. **L'information est donc perdue**.

– Dans le cas du lien logique, `fichier1` (fichier source) et `fichier2` (fichier lien) pointent directement sur les données résidant sur le disque. Par conséquent,si `fichier1` est effacé, **le contenu n'est pas perdu et est toujours accessible par** `fichier2`.

La figure 2.2 montre les différences entre les liens symboliques et les liens logiques vis à vis de leur liaisons avec les informations sur le système de fichiers.

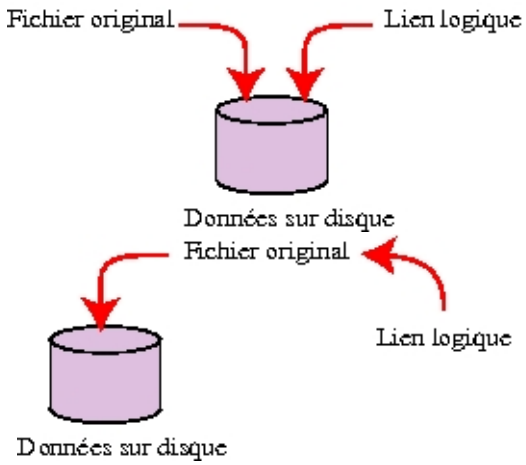


FIG. 2.2 – *Différence entre les liens symboliques et les liens logiques*

Remarque 2.2 :

Deux ou plusieurs fichiers liés par un lien logique doivent résider sur le même système de fichiers.

2.2.3.1 Visualisation du nombre de liens avec la commande `ls`

Exemple 2.4 :

Si l'on prend l'exemple des commandes suivantes :

```
% ls -l
arthur lancelot merlin
% ln -s lancelot dulac
% ln merlin enchanteur
% ls -l
-rw-r--r-- 1 schmoll nobody 37 Jul 25 12:00 arthur
lrw-r--r-- 1 schmoll nobody 37 Jul 25 12:03 dulac -> lancelot
-rwxr-xr-x 2 schmoll nobody 37 Jul 25 12:01 enchanteur
-rw-r--r-- 1 schmoll nobody 37 Jul 25 12:02 lancelot
-rwxr-xr-x 2 schmoll nobody 37 Jul 25 12:01 merlin
```

donc

- `dulac` est un lien symbolique sur `lancelot`,
- `enchanteur` est un lien logique vers les mêmes informations que celles pointées par le fichier `merlin`.

On remarque que le fichier `dulac` est de type "*lien*" et pointe vers le fichier `lancelot`. Le nombre de liens pour `dulac` est **1** (un lien vers lancelot).

En conclusion, pour les liens symboliques :

L'information sur le disque est accédée uniquement par le fichier "`lancelot`".
Lorsqu'on accède au fichier "`dulac`", le système, renvoie l'identifiant du fichier "`lancelot`". Par conséquent, si le fichier "`lancelot`" est détruit, "`dulac`" perdra les références aux données.

En conclusion, pour les liens logiques :

Par contre, les types des fichiers "`merlin`" et "`enchanteur`" correspondent à un "*fichier régulier*" ou "*standard*". De plus, la commande "`ls -l`" indique **deux liens**. En effet, les *secteurs* sur le disque physique correspondent à deux noms différents (fichiers `merlin` et `enchanteur`).
L'information ne réside q'une seule fois sur le disque mais elle peut être accédée par deux noms de fichiers différents. Par conséquent, si le fichier "`merlin`" est détruit, le système a toujours accès aux données via le fichier "`enchanteur`".
Le seul moyen de connaître les liaisons entre deux liens logiques est de connaître l'identifiant sur le disque ("*i-node*") et de rechercher les fichiers le référençant.

Le tableau 2.6 montre les correspondances des commandes `cp`, `mv` et `ln` entre les systèmes d'exploitations UNIX, OpenVMS et MS-DOS.

UNIX	OpenVMS	MS-DOS
`cp`	`COPY`	`COPY`
`mv`	`RENAME`	`REN`
`ln`	`SET FILE/ENTER`	Pas d'équivalence

TAB. 2.6 – *Équivalence des commandes* `cp`, `mv` *et* `ln` *entre* UNIX, *Open*VMS *et MS-DOS*

2.2.4 Effacement d'un fichier - Commande `rm`

Syntaxe :

```
rm [-irf] fichier...
```

La commande "`rm`" est utilisée pour effacer des fichiers. Une fois effacés, les fichiers ne peuvent plus être récupérés[1]. La commande `rm` exige au moins un

1. à moins, biensûr, de diposer d'un système de sauvegarde

argument. Si plusieurs arguments sont fournis à la commande, tous les fichiers spécifiés seront effacés en fonction des modes de protections.

L'option "`-r`" (récursif) indique la récursivité et permet d'effacer un répertoire et tout son contenu.

L'option "`-i`" (interactive) demande une confirmation (y ou n) sur chaque fichier à effacer.

L'option "`-f`" (force) ne fait plus tenir compte à "`rm`" des protections du fichier, mais uniquement du propriétaire. Vous pouvez donc effacer vos fichiers, même s'ils sont protégés.

ATTENTION AUX CATASTROPHES AVEC LES OPTIONS "`-f`" ET SURTOUT "`-r`" !!!

Le tableau 2.7 donne les correspondances des différents comportements de la commande `rm` entre les systèmes d'exploitations Unix, OpenVms et MS-DOS.

Unix	OpenVms	MS-DOS
`rm`	`DELETE`	`DEL`
`rm -i`	`DELETE/CONFIRM`	Pas d'équivalence

Tab. 2.7 – *Équivalences de la commande* `rm` *entre* Unix, *Open*Vms *et MS-DOS.*

2.3 Protections sur les fichiers

2.3.1 Notion d'identité sous Unix

Comme tout système multi-utilisateurs, multi-tâches, vous devez vous identifer avant de pouvoir travailler sous Unix par :

- votre nom de login ou *logname*,
- votre mot de passe.

Dès que vous êtes authentifié, le système lance l'interpéteur de commande votre numéro d'utilisateur : l'*UID*[2]. Il lui associe aussi un ou plusieurs groupes (en fonction du profil utilisateur) : le *GID*[3].

A partir de ce moment, tous les sous-processes générés seront lancés sous la même identité, c'est à dire avec les mêmes *UID*/*GID*.

Chaque fichier sous Unix possède un propriétaire (associé à un *UID*) et un groupe (associé à un *GID*). De façon général, le propriétaire et le groupe du fichier correspondent à ceux du process qui l'a créé.

Le propriétaire du fichier peut en modifier la paternité.

Le propriétaire du fichier peut en modifier le groupe.

2. *UID* = User Identifier
3. *GID* = Group Identifier

Au niveau du mécanisme des protections, il n'y a pas de liaison entre le *GID* et l'*UID*. Si un utilisateur a un *UID* donné identique à celui du fichier et un GID différent, il se placera au niveau du propriétaire. Le numéro d'*UID* est unique par utilisateur.

Il n'y a donc pas de notion de couple (*UID*,*GID*) au même titre que sur OpenVMS avec les UIC `"[group,member]"`.

L'administrateur du système (`root` ou *super-user* équivalent à `SYSTEM` sur OpenVMS) est vu comme le propriétaire de tous les fichiers.

Le tableau 2.8 donne les équivalences entre UNIX et OpenVMS pour les notions d'identités sur le système.

UNIX	OpenVMS
`logname`	`Username`
`UID`	*UIC*
`GID`	groupe dans l'*UIC*

TAB. 2.8 – *Équivalences pour les notions d'identités entre* UNIX *et Open*VMS

2.3.2 Permissions

Sous UNIX, on distingue trois modes d'accès :

- l'accès en lecture,
- l'accès en écriture,
- l'accès en exécution.

En fonction du type de fichier (un répertoire ou un fichier standard), le mode de protection permet de faire les actions décrites dans le tableau 2.9.

Accès	Fichier	Répertoire
Lecture	Le contenu du fichier est visible.	le contenu du répertoire est visible [a].
Écriture	Le contenu du fichier peut être modifié.	Le contenu du répertoire peut être modifié [b].
Exécution	Le fichier peut être utilisé en commande [c].	Le répertoire peut devenir le répertoire courant.

TAB. 2.9 – *Actions possibles en fonction du masque de protection*

[a] Visible par la commande `"ls"` par exemple.
[b] Création, suppression de fichiers, etc.
[c] Reportez vous à la partie III.

Remarque 2.3 :

Remarques sur les protections d'un répertoire
Si le répertoire est accessible en lecture, alors on peut voir les fichiers qui s'y trouvent par la commande `ls` *par exemple.*

Si le répertoire est accessible en écriture, il est alors possible de faire des manipulations sur les fichiers qui s'y trouvent avec les commandes `mv`, `cp` *et* `rm`. *Il faut donc bien faire attention à ce mode de protection.*
Si le répertoire est accessible en exécution, il peut devenir le répertoire courant avec la commande `cd`.
Par conséquent, pour qu'un fichier puisse être effacé, **il faut avoir les droits d'écriture sur le répertoire qui le contient**.

Les protections d'un fichier se situent à trois niveaux :

- le niveau utilisateur (`user`),
- le niveau groupe (`group`),
- le niveau autre (`other`).

La figure 2.3 décrit la méthode d'accès d'un processus à un fichier.

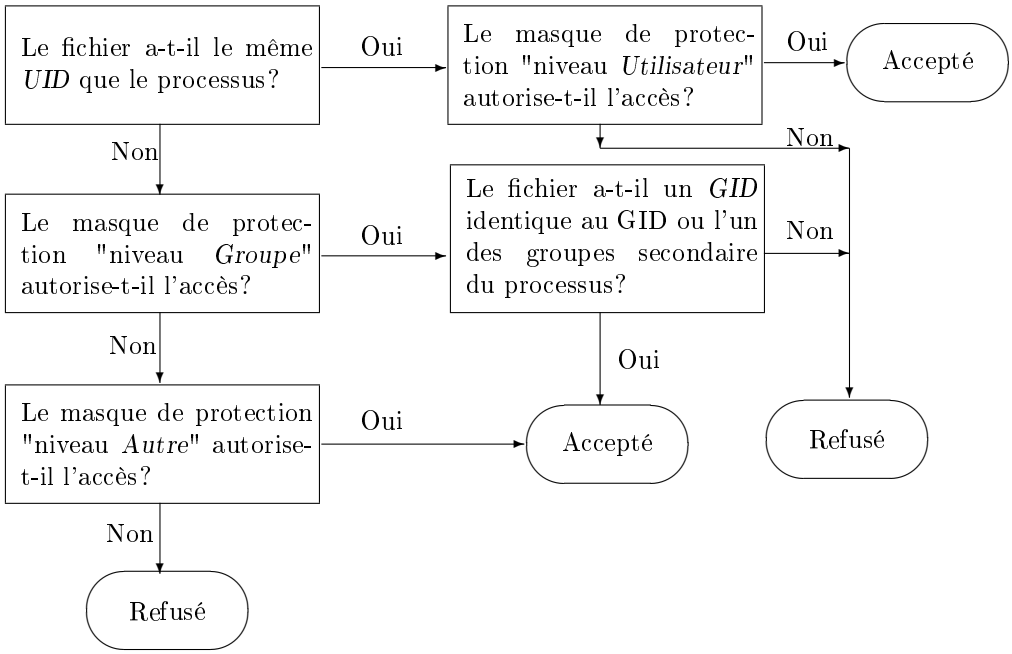


FIG. 2.3 – *Algorithme de vérification des droits d'accès sous* UNIX

2.3.3 Changement de protection - Commande `chmod`

Syntaxe :

```
chmod mode fichier...
```

avec :

`mode`=masque de protections

ou bien `mode`=`<u|g|o><+|-><r|w|x>`

Les permissions peuvent être modifiées pour un fichier ou un répertoire par le propriétaire (ou l'administrateur) en utilisant la commande "`chmod`".

Il est possible de spécifier le nouveau masque de protection de deux façons :

- préciser la totalité du masque de protection en **octal**,
- changer le masque de protection niveau par niveau.

Le masque de protection en octal s'interprète de la façon suivante :

- le droit de l'accès en lecture correspond à $2^2 = 4$,
- le droit de l'accès en écriture correspond à $2^1 = 2$,
- le droit de l'accès en exécution correspond à $2^0 = 1$.

Le tableau 2.10 résume les différentes valeurs associées aux différents droits d'accès.

droits d'accès	lecture 2^2 4	écriture 2^1 2	exécution 2^0 1
abréviation utilisée	`r`	`w`	`x`

TAB. 2.10 – *Valeurs associées aux différents droits d'accès*

Pour affecter les droits d'accès à un fichier ou un répertoire, il suffit de procéder de la façon suivante :

- on additionne entre elles toutes les autorisations d'accès,
- on effectue cette opération pour chaque niveau d'accès (*utilisateur*, *groupe*, *autre*).

Exemple 2.5 :

Le tableau 2.11 donne un exemple de la démarche à suivre pour affecter protéger un fichier avec un masque en octal.

Utilisateur			*Groupe*			*Autre*		
`r`	`w`	`x`	`r`	`-`	`x`	`-`	`-`	`-`
1	1	1	1	0	1	0	0	0
$2^2 \times 1$	$2^1 \times 1$	$2^0 \times 1$	$2^2 \times 1$	$2^1 \times 0$	$2^0 \times 1$	$2^2 \times 0$	$2^1 \times 0$	$2^0 \times 0$
	7			5			0	

TAB. 2.11 – *Exemple d'affectation d'un masque en octal*

Une autre façon de préciser le masque de protection est de dire, pour chaque niveau, quels sont les accès que l'on autorise ou que l'on interdit par rapport au masque de protection courant. Les abréviations utilisées dans ce cas par la commande "`chmod`" sont décrites dans le tableau 2.12.

Pour chaque niveau, la commande "`chmod`" attend un masque de protection du type :

`<protectionlevel>+<access permisssion>`	pour autoriser un accès,
`<protectionlevel>-<access permisssion>`	pour supprimer un accès.

Abréviation utilisée par `chmod`	Signification pour `chmod`	Signification
`u`	`user`	niveau utilisateur
`g`	`group`	niveau groupe
`o`	`other`	niveau autre
`r`	`read`	accès en lecture
`w`	`write`	accès en écriture
`x`	`execute`	accès en exécution

TAB. 2.12 – *Abréviations utilisées par la commande "`chmod`"*

Exemple 2.6 :

Les exemples donnés dans le tableau 2.13 montrent comment modifier les protections d'un fichiers par rapport à celles qui sont déjàs affectées.

Exemple	Signification
`u+rwx`	Rajoute les droits de lecture, d'écriture et d'exécution au niveau de l'utilisateur.
`g+rx`	Rajoute les droits de lecture et d'exécution au niveau du groupe.
`g-w`	Retire les droits en écriture au niveau du groupe.
`o-rwx`	Retire tous les accès pour les autres utilisateurs (ni le propriétaire, ni un utilisateur du même groupe).

TAB. 2.13 – *Exemples de modifications des protections par rapport à celles déjà actives*

Remarque 2.4 :

Il est possible d'avoir des équivalences entre les deux fonctionnements. Par exemple, les deux commandes suivantes sont équivalentes :

```
% chmod 750 fichier
% chmod u+rwx g+rx g-w o-rwx fichier
```

2.3.4 Remarques sur les protections

Tous les répertoires inclus dans le chemin d'accès d'un fichier doivent être accessibles en exécution pour qu'il puisse être atteind.

Pour protéger un fichier, supprimez le droit d'accès en écriture sur ce fichier ainsi que sur le répertoire dans lequel il se trouve.

2.4 Les filtres

2.4.1 Rappels, Propriétés

Rappel :

Un filtre est une commande devant effectuer une lecture à partir de l'entrée standard et une écriture sur la sortie standard. La figure 2.4 en montre le fonctionnement.

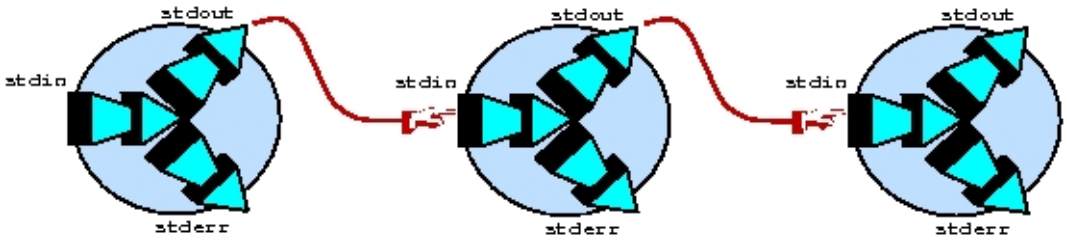


FIG. 2.4 – *Rappel sur les filtres*

Propriétés :

Par défaut, les filtres lisent sur leur entrée standard et affichent le résultat sur leur sortie standard.
Si un fichier est spécifié en argument, l'entrée standard est redirigée automatiquement sur celui-ci. S'il y en a plusieurs, ils sont mis bout à bout et le filtre redirige son entrée standard sur le résultat obtenu.

2.4.2 Filtres déjà vus

La commande "`cat`" fonctionne comme un filtre si elle n'a pas d'arguments. Elle lit sur son entrée standard et la réaffiche sur sa sortie standard. La figure 2.5 en illustre son fonctionnement.

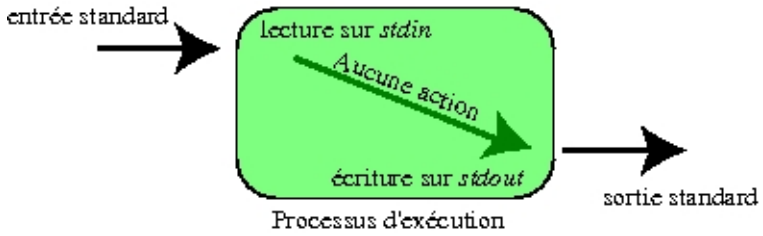


FIG. 2.5 – *Fonctionnement de la commande "`cat`"*

Remarque 2.5 :

La commande "`more`" n'est pas un filtre. Elle lit les informations sur son entrée standard ou bien dans les fichiers passés en argument, et redirige sur l'écran. Elle fait appel aux informations que le système

connait sur le terminal pour connaître sa taille, son mode d'émulation, etc...

2.4.3 Filtre `sort`

Syntaxe :

```
sort [-nd] [-tcaractère] [+numéro-champ>] [-numéro-champ]
            [fichier...]
```

Le filtre "`sort`" permet de trier les lignes de caractères (suite d'octets délimitée par le caractère `<CR>`) envoyées sur l'entrée standard selon un ensemble de critères.

Il est possible de définir un caractère séparateur de champ afin d'effectuer des tris sur une zone particulière. Le tri peut se faire :

- soit numériquement en spécifiant l'option "`-n`",
- soit selon l'ordre `ASCII` standard (mode par défaut),
- soit selon l'ordre défini dans un dictionnaire avec l'option "`-d`".

Les champs sont délimités par défaut par une tabulation ou de façon explicite par le caractère spécifié avec l'option -t<caractère>.

La commande sort lit sur son entrée standard, effectue le tri et affiche le résultat sur sa sortie standard.

Comme la plupart des filtres, la commande "`sort`" accepte des fichiers en arguments. S'ils sont précisés sur la ligne de commande, "`sort`" redirige son entrée standard sur leur contenu. Il est également possible de trier sur un champ particulier en utilisant le symbole "`+`" suivi du numéro du champ.

Remarque 2.6 :

"`sort`" numérote les champs à partir de zéro.

Si la commande est simple à utiliser pour effectuer des tris simples, pour des tris plus complexes, plusieurs tentatives sont bien souvent nécessaires avant de trouver la bonne syntaxe. Ne soyez pas frustrés : la puissance de la commande pallie cet inconvénient. Pour plus de renseignements, consultez le manuel de référence "`sort(1)`".

Exemple 2.7 :

```
% sort -nt: +2 /etc/passwd
% ls -R | sort
```

2.4.4 Filtre `grep`

Syntaxe :

```
grep [-inv] expression [fichier...]
```

Le filtre "`grep`" recherche l'expression précisée sur son entrée standard et l'affiche sur sa sortie standard. Cette expression obéit aux lois des expressions

régulières UNIX (cf. chapitre 8). De façon générale, on spécifie une chaîne de caractères.

Les options les plus courantes de la commande "grep" sont :

- l'option "-i" indique à "grep" qu'il ne faut pas tenir compte des majuscules/minuscules,
- l'option "-v" indique à "grep" qu'il faut afficher les lignes ne contenant pas l'expression précisée en argument,
- l'option "-n" permet de voir afficher les numéros de lignes courantes.

Comme la plupart des filtres, la commande "grep" accepte des fichiers en arguments. S'ils sont précisés sur la ligne de commande, "grep" redirige son entrée standard sur leur contenu.

Exemple 2.8 :

```
% grep user /etc/passwd
% ls -l | grep rwxrwxrwx
```

2.4.5 Filtre wc

Syntaxe :

```
wc [-lwc] [fichier{...}]
```

Le filtre "wc" lit sur son entrée standard, compte le nombre de lignes (enregistrements séparés par <CR>), le nombre de mots (enregistrements séparés par SPACE ou TAB), le nombre de caractères et affiche le résultat sur sa sortie standard.

Les options sont :

-l	compte le nombre de lignes,
-w	compte le nombre de mots,
-c	compte le nombre de caractères.

L'ordre des options dans la ligne de commande détermine l'ordre de sortie. Comme la plupart des filtres, la commande "wc" accepte des fichiers en arguments. S'ils sont précisés sur la ligne de commande, "wc" redirige son entrée standard sur leur contenu.

Exemple 2.9 :

```
% wc -l /etc/passwd
% ls -l | wc -l
```

2.4.6 Filtre cut

Syntaxe :

```
cut -cliste [fichier...]
cut -fliste [-dcaractère] [-s] [fichier...]
```

Le filtre "cut" a deux modes de fonctionnement :

Mode	Option
Extraire des colonnes à partir de l'entrée standard.	option "-c"
Extraire des champs à partir de l'entrée standard.	option "-f"

Dans les deux modes, "liste" est une séquence de numéros pour indiquer à cut quels sont les champs ou les colonnes à retenir. Il y a plusieurs formats possibles pour cette liste :

"A-B"	champs ou colonnes A à B inclus
"A-"	du champ ou colonne A jusqu'à la fin de la ligne
"A,B"	champ ou colonnes A et B
"-B"	du début jusqu'au champ ou colonne B

Toute combinaison des formats précédents est également possible.

Exemple 2.10 :

```
% cut -f1,4,6-9 /tmp/fictest
```

extrait du fichier "/tmp/fictest" les champs 1, 4 et de 6 à 9.

Dans le cas d'un découpage par champ, il existe une option particulière, "-d", pour spécifier le caractère séparateur de champs. Par défaut, ce caractère est la tabulation "TAB". De même, l'option "-s", lors d'un découpage par champ, indique à cut d'écarter toutes les lignes qui ne contiennent pas le séparateur.

Remarque 2.7 :

La commande "cut" commence la numérotation des champs à 1 alors que "sort" commence à 0. Il y a un moyen facile de s'en rappeler en notant que sort contient un zéro dans son nom (en fait un "o") contrairement à "cut".

Comme la plupart des filtres, la commande "cut" accepte des fichiers en arguments. S'ils sont précisés sur la ligne de commande, "cut" redirige son entrée standard sur leur contenu.

Exemple 2.11 :

```
% cut -f3,7 -d: /etc/passwd
% date | cut -c1-3
% ps -ef | cut -c48- | sort
```

Chapitre 3

Commandes usuelles de communication réseau

3.1 Connexion à une autre machine – commande `telnet`

Syntaxe :

```
telnet [host [port]]
```

La commande "`telnet`" permet de communiquer avec une autre machine en utilisant le protocole *Telnet*. Si "`telnet`" est invoqué sans argument, il passe en mode commande, indiqué par l'invite "`telnet>`". Dans ce mode, il accepte et exécute des commandes. Si "`telnet`" est invoqué avec des arguments, il ouvre une connexion sur ce qui lui a été spécifié.

Une fois qu'une connexion a été ouverte, "`telnet`" passe en mode saisie. Dans ce mode, le texte tapé au clavier est envoyé à la machine cible.

Voici quelques commandes accessibles au niveau du prompt "`telnet>`" :

`open [host [port]]`
: Ouvre une connexion sur le nœud spécifié au port indiqué. Si aucun port n'est spécifié, "`telnet`" essaie de contacter un serveur *Telnet* au port standard de *Telnet*. Le nom de nœud peut être soit le nom officiel, soit un *alias*, soit une adresse Internet en notation décimale.

`close`
: Ferme une session "`telnet`". Si la session a commencé en mode commande, alors "`telnet`" repasse en mode commande; autrement, `telnet` se termine.

`?`
: Fournit de l'aide. Sans argument "`?`" affiche le menu d'aide. Si une commande est précisée après "`?`", le message affiché s'applique uniquement à la commande spécifiée.

Pour plus d'informations, consultez `telnet(1)`.

Remarque 3.1 :

"`telnet`" est seulement un service interactif. Vous ne pouvez pas exécuter "`telnet`" en arrière-plan ou à partir d'un programme shell.

3.2 Transfert de fichiers - commande `ftp`

Syntaxe :

```
ftp [-g] [-i] [-n] [v] [host]
```

"`ftp`" est une famille de commandes pour les opérations de manipulation de fichiers ou de répertoires à travers le réseau.

Vous pouvez importer ou exporter des fichiers à partir d'une machine distante (sous UNIX ou non), en utilisant soit le mode de transfert `ASCII`, soit le mode de transfert *binaire*.

Vous pouvez :

- mettre à jour, renommer et supprimer des fichiers,
- lister le contenu de répertoires,
- changer, créer et supprimer des répertoires,
- vérifier l'état, changer les options,
- demander de l'aide.

"`ftp`" admet quatre options :

`-g` : inhibition des métacaractères.
Lorsque cette option n'est pas précisée, par défaut, les métacaractères sont interprétés pour l'importation ou l'exportation de fichiers. Pour plus de précisions sur les métacaractères, reportez vous à la section 1.5.

`-i` : inhibition du mode interactif sur les manipulations de fichiers.
Le mode interactif est actif par défaut.

`-n` : désactivation de la connexion automatique.
La connexion automatique est autorisée.

`-v` : mode *verbose*.
En l'absence de cette option, "`ftp`" utilise le mode *verbose* uniquement si la sortie standard est associée à un terminal.

Quelques-unes des commandes de "`ftp`" sont expliquées ci-dessous. Dans les explications, "`server-host`" désigne la machine sur laquelle on se connecte avec "`ftp`". "`local-host`" désigne la machine sur laquelle la commande "`ftp`" a été lancée. La figure 3.1 illustre les terminologies utilisées.

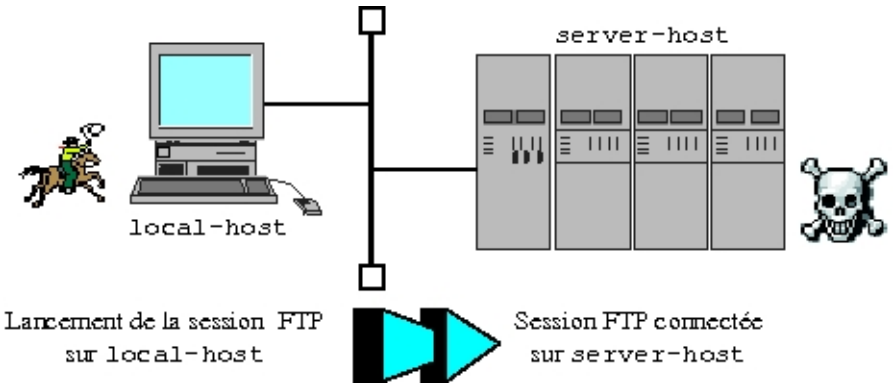


FIG. 3.1 – *Terminologie pour la description de "`ftp`".*

Apercu des commandes `ftp` :

`open server-host [port-number]`
: Etablit une connexion avec "`server-host`" en utilisant un numéro de port si spécifié. Si aucun port n'est précisé, "`ftp`" essaie de contacter un serveur avec le numéro de port standard.

`user user_name [password] [account]`
: Connexion sous l'identité "`user_name`" sur "`server-host`" ouverte avec la commande "`open`". La figure 3.2 illustre l'établissement d'une connexion.

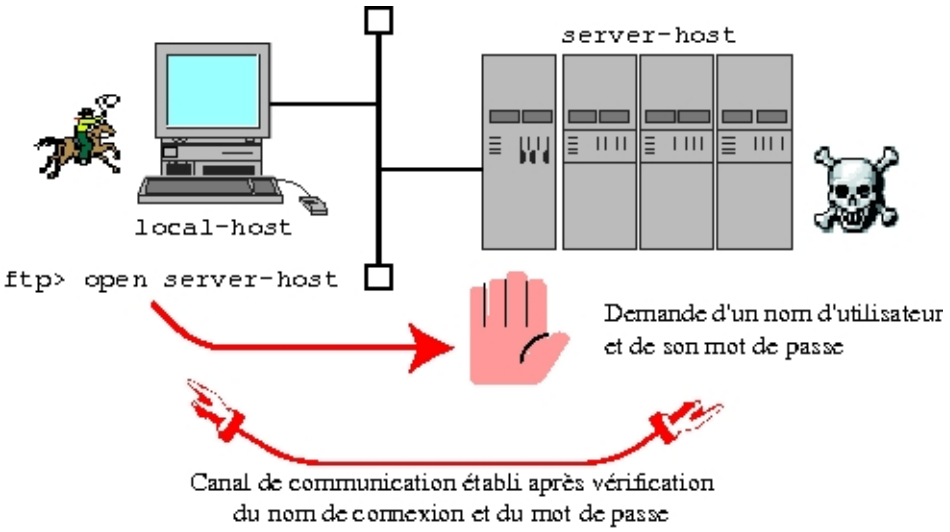


FIG. 3.2 – *Etablissement d'une connexion* `ftp`

`glob`
: Autorise l'usage des métacaractères. Si cette option est activée, "`ftp`" envoie les métacaractères à "`server-host`" pour qu'il puisse les interpréter au niveau des noms de fichiers et des répertoires existants. La substitution

des métacaractères est toujours faite avec la commande "`ls`". Pour plus de précisions sur les métacaractères, preportez-vous à la section 1.5.

`binary`
: Positionne l'option binaire pour le type de transfert de fichiers.

`ls [remote_dir [local_file]]`
: Affiche les noms des fichiers du site distant "`remote_dir`" à l'écran, ou éventuellement la redirige dans un fichier local "`local_file`". Si "`remote_dir`" et "`local_file`" ne sont pas précisés,alors le répertoire de travail distant est affiché sur la sortie standard.

`put local_file [remote_file]`
: Copie un fichier local "`local_file`" vers le site distant sous le nom "`remote_file`". Si "`remote_file`" n'est pas spécifié, "`ftp`" copie le fichier sous le même nom. La figure 3.3 illustre l'envoi et la réception de fichiers entre un serveur et un client "`ftp`".

`mput local_file local_file ...`
: Copie plusieurs fichiers du site local vers le site distant. Les fichiers de destination ont les mêmes noms que les fichiers locaux d'origine. Si l'option "`glob`" est activée, les métacaractères sont interprétés. La figure 3.3 illustre l'envoi et la réception de fichiers entre un serveur et un client "`ftp`".

`get remote_file [local_file]`
: Copie un fichier distant "`remote_file`" sur le système local sous le nom "`local_file`". Si "`local_file`" n'est pas précisé, "`ftp`" copie le fichier avec le même nom. La figure 3.3 illustre l'envoi et la réception de fichiers entre un serveur et un client "`ftp`".

`mget remote_file remote_file ...`
: Copie plusieurs fichiers distants vers le système local. Si l'option "`glob`" est activée, les métacaractères sont interprétés. La figure 3.3 illustre l'envoi et la réception de fichiers entre un serveur et un client "`ftp`".

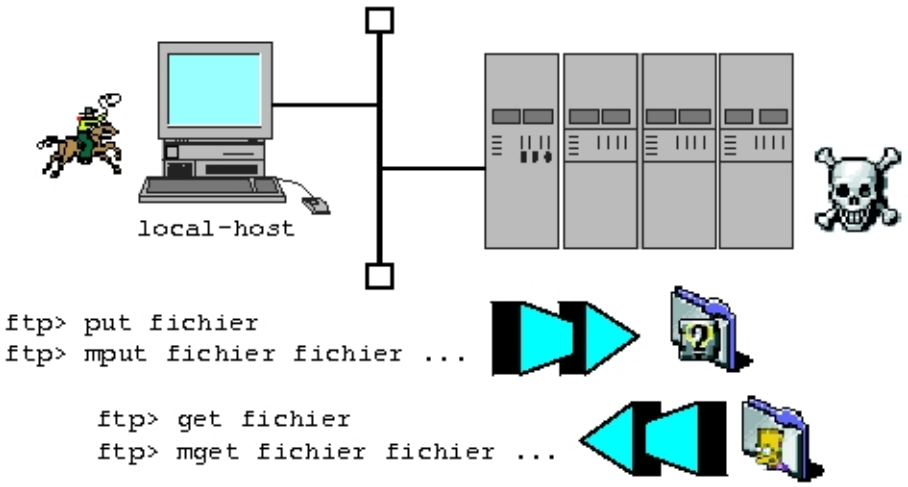


FIG. 3.3 – *Envoi/Réception de fichier(s) de "*`server-host`*" vers "* `local-host` *" avec "*`ftp`*".*

`close`
: Ferme la connexion avec "`server-host`". La commande "`close`" ne permet pas de sortir de "`ftp`" si la connexion a été établie avec une commande "`open`".

`quit` *ou* `bye`
: Les commandes "`quit`" et "`bye`" ont le même effet. Elles ferment la connexion avec "`server-host`" si une connexion était ouverte et sort de "`ftp`".

Exemple 3.1 :

Utilisation de "`ftp`" dans un programme shell

Vous pouvez utiliser "`ftp`" dans un programme shell en respectant certaines règles. Voici un exemple d'utilisation de "`ftp`" dans un programme Bourne Shell.

```
(
    for host in willow arthur merlin
    do
        echo "
            open $host
            user lancelot dulac
            binary
            mput excalibur*
            close
        "
    done
) | ftp -i -n
```

Ce programme shell représente un risque au niveau de la sécurité (mot de passe en clair dans une procédure). Ce type de programme est à utiliser avec parcimonie. Pour plus de renseignements, consultez "`ftp(1)`".

3.3 Connexion automatique – commande `rlogin`

Syntaxe :

`rlogin remote_host [-ec] [-l` *username*`]`

La commande "`rlogin`" connecte votre terminal du site local sur un nœud distant. "`rlogin`" agit comme un terminal virtuel sur le système distant. Le nom de machine distante peut être le nom officiel ou bien un *alias*[1]. Il est possible aussi d'utiliser l'adresse Internet. Le type de terminal (indiqué par la variable "`TERM`") est propagé à travers le réseau et est utilisé pour initialiser la variable "`TERM`" sur le site distant. Pour plus de renseignements sur les variables d'environnement du shell, reportez vous à la section 1.2.

Une fois connecté au site distant, vous pouvez exécuter des commandes sur le site local en utilisant le caractère d'échapement (par défaut "`~`"). Une ligne

1. Nom secondaire au niveau de la configuration réseau.

commençant par "`~!`" crée un shell et vous permet d'exécuter temporairement des commandes sur le site local. Une ligne commençant par "`~`" vous déconnecte du site distant. Plus simplement, pour revenir à votre session de départ, il suffit de taper la commande "`exit`" sur le site distant.

Exemple 3.2 :

```
dragon % rlogin king -l arthur
Password:
Welcome on IRIX 4.5 4D/480 to node king
king% ~!
dragon %            (vous êtes de retour sur votre terminal local)
...
dragon % exit              (vous retournez à la station distante)
king%
```

Les options de rlogin sont décrites ci-dessous :

`-e`*c*	Positionne le caractère d'échappement à "`c`". Il n'y a pas d'espace entre le caractère "`-e`" et le caractère d'échappement.
`-l` *username*	Met le nom de connexion de l'utilisateur sur le site distant à "*username*". Par défaut, le nom de connexion de l'utilisateur sur le site local est utilisé pour se connecter au site distant.

Pour plus de renseigements, consultez "`rlogin(1)`".

3.4 Transfert de fichiers automatique – commande `rcp`

Syntaxe :

`rcp` *source destination*
avec *source* et *destination* de la forme `[[`*user*`@]`*host*`:]`*pathname*

La commande "`rcp`" est similaire à la commane "`cp`" d'Unix. Comme "`cp`", "`rcp`" aura un comportement différent selon le nombre d'arguments :

- si elle ne possède que deux arguments, elle copie le fichier source sous le nouveau nom,
- si elle possède plusieurs arguments, la destination est obligatoirement un répertoire. Dans ce cas, elle copie les fichiers sources dans le répertoire spécifié.

"`rcp`" ne demande pas de mot de passe si une équivalence système ou utilisateur a été configurée.

Dans le cas contraire, elle ne fonctionne pas (message "`Permission denied`"). "`rcp`" autorise des transferts mettant en jeu plusieurs machines. Par exemple, si vous êtes connecté sur une machine, vous pouvez transférer des fichiers entre deux nœuds du réseau totalement différents.

La source et la destination se présentent sous la forme `[[`*user*`@]`*host*`:]`*pathname* avec :

`host`	Spécifie le système sur lequel se trouve le fichier. Si "`host`" n'est pas précisé, le système local est utilisé par défaut.
`user`	Spécifie l'utilisateur sur "`host`". Si "`user`" n'est pas précisé, l'utilisateur sur le site distant est le même que sur le site local (les noms des utilisateurs doivent correspondre d'un système à l'autre).
`pathname`	Donne le chemin du fichier à copier. Il peut être relatif ou absolu. Un chemin relatif est interprété de manière relative par rapport au répertoire de connexion sur le site distant ou par rapport au répertoire courant sur le site local.

Il est possible d'utiliser les métacaractères pour référencer des fichiers. N'oubliez pas que le shell local interprète n'importe quel métacaractère avant d'appeler "`rcp`". Pour empêcher une substitution locale des métacaractères devant être traités par le site distant, mettez les entre simples quotes (cf. section 1.6).

Exemple 3.3 :

On supposera que toutes les équivalences utilisateur et système ont été configurées correctement.
Le processus de cet exemple est décrit au tableau 3.1 et à la figure 3.4.

Dans la configuration décrite dans la figure 3.4,

Exemple	Description
1	L'utilisateur `willow` sur la machine `dragon` copie le fichier `excalibur.c` de l'utilisateur `arthur` sur la machine `king` dans son répertoire courant.
2	L'utilisateur `willow` sur la machine `dragon` copie le fichier local `donjon.c` sur la machine `dragon` vers le répertoire de connexion de l'utilisateur `willow` sur la machine `dulac`.
3	L'utilisateur `willow` sur la machine `dragon` copie tous les fichiers "`.c`" de l'utilisateur `arthur` sur la machine `king` vers son répertoire courant.
4	L'utilisateur `willow` sur la machine `dragon` copie le fichier `excalibur.c` de l'utilisateur `arthur` sur la machine `king` vers le répertoire de connexion de l'utilisateur `lancelot` sur la machine `dulac`.

Description des opérations effectuées.

On obtient l'exemple illustré dans la figure 3.4.

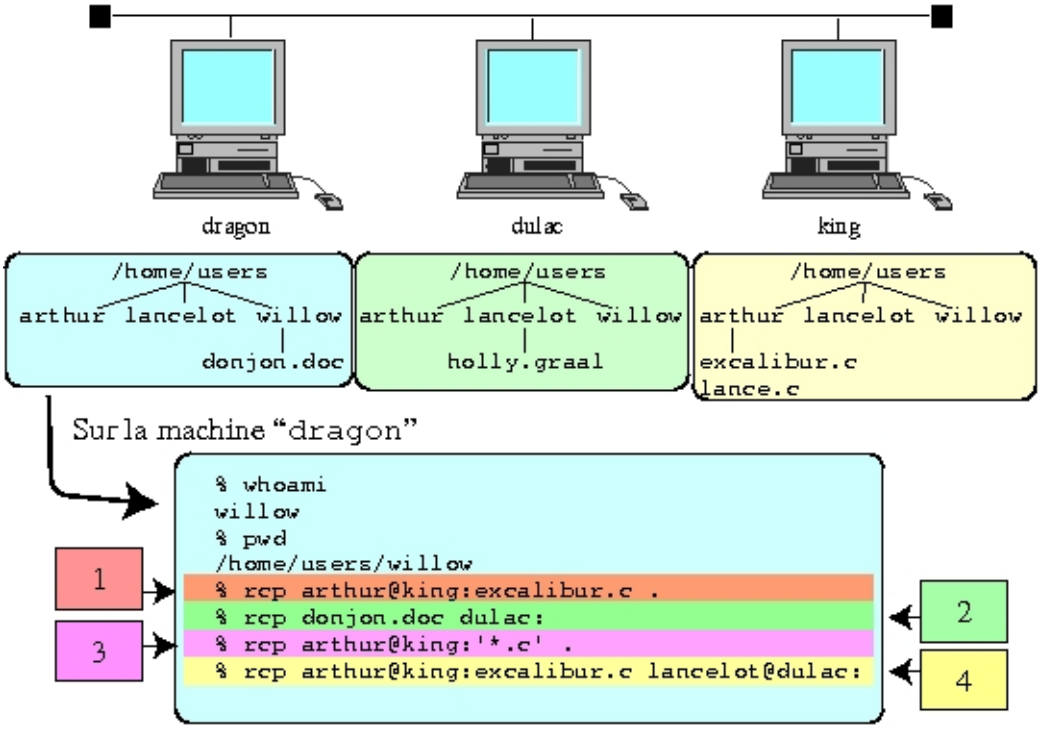


FIG. 3.4 – *Exemple d'utilisation de la commande "*`rcp`*".*

3.5 Exécution d'une commande à distance - commande `rsh` (ou `remsh`)

Syntaxe :

`rsh [-l` *username*`] [-n]` *commande*
`remsh [-l` *username*`] [-n]` *commande*

Les commandes "`rsh`" et "`remsh`" sont équivalentes. Sur certains sytèmes il n'exite que la commande "`rsh`"[2], sur d'autres que seule la commande "`remsh`" sera disponible[3]), sur d'autres les deux cohabiteront[4]. Dans toute la suite de ce paragraphe, seulement "`rsh`" sera cité.

La commande "`rsh`" exécute une commande non interactive sur un système distant. Le nom du compte est le même que le nom du compte local à moins que vous ne spécifiez l'option "`-l`".

Comme "`rcp`", "`rsh`" ne demande pas de mot de passe si une équivalence système ou utilisateur a été configurée. Dans le cas contraire, elle ne fonctionne pas (message "`Permission denied`"). La commande "`rsh`" transmet les signaux

2. *SunOS* et *Solaris* sur les systèmes de Sun Microsystems, *Irix* sur les machines de Silicon Graphics, *Digital* UNIX sur les machines de Compacq - ex Digital Equipment Corp.
3. *UTekV* sur les anciens systèmes UNIX de Tektronix, HP–UX sur les systèmes de Hewlett-Packard
4. AIX, l'UNIX d'IBM

"`INTERRUPT`", "`QUIT`" et "`HUP`" à la commande distante.

Pour plus de précisions, reportez vous à "`signal(2)`" et au chapitre 2.

Vous pouvez utiliser les métacaractères avec "`rsh`". Si vous voulez qu'ils soient interprétés sur le site distant, assurez-vous qu'ils sont bien entre simples quotes (cf. sections 1.5 et 1.6).

Remarque 3.2 :

"`rsh`" ne peut pas exécuter des commandes en mode interactif comme "`vi`", "`emacs`", etc.

3.5.1 Comparaisons `telnet`/`rlogin` et `ftp`/`rcp`

Fonctionnalité	`telnet`	rlogin
Ensemble de commandes.	Oui	Non
Peut se connecter à des systèmes non UNIX.	Oui	Non [5]
Peut être configuré en connexion automatique.	Non	Oui (fichiers `/etc/hosts.equiv` et `~/.rhosts`)
Peut utiliser l'adresse IP pour la connexion.	Oui	Oui
Sortie autorisée.	Oui (vers "`telnet`" en mode commande)	Oui (vers le terminal local)
Nombre de modes.	2 (commande et connecté)	1 (connecté seulement)
Peut être lacé depuis un programme Shell.	Non	Non

TAB. 3.2 – Comparaisons `telnet`/`rlogin`

Remarque 3.3 :

Notez que "`telnet`" et "`rlogin`" peuvent être appelés depuis un programme Shell, mais vous ne pouvez pas leur transmettre des informations au clavier (comme vous pouvez le faire avec "`ftp`").

Fonctionnalité	`ftp`	`rcp`
Ensemble de commandes	Oui	Non
Peut transférer des fichiers sur un système non UNIX	Oui	Non [6]
Peut être exécuté dans un programme Shell	Oui	Oui
Peut mettre en jeu 3 nœuds	Non	Oui
Autorise les métacaractères	Oui (commande "`glob`")	Oui
Peut faire une copie récursive	Non	Oui (option "`-r`")

Suite page suivante ···

5. Dépend de l'implantation sur le système non-UNIX.
6. Sauf implémentation d'un serveur supportant ce type de connexion.

Suite de la page précédente.

Fonctionnalité	`ftp`	`rcp`
Peut configurer une équivalence utilisateur	Oui (fichier "~/.netrc")	Oui (fichiers "/etc/hosts.equiv", "~/.rhosts")
Equivalence utilisateur requise	Non	Oui
Peut utiliser une adresse IP pour la connexion	Oui	Oui

TAB. 3.3 – Comparaisons `ftp`/`rlogin`

Deuxième partie

Introduction au shell

Chapitre 1

Notions élémentaires du Bourne Shell

1.1 Introduction

Le shell est un interpréteur de commandes qui :

- initialise l'environnement,
- génère le prompt.

Quand une commande est validée, le shell

1. effectue les substitutions de variables,
2. interprète les métacaractères,
3. gère les redirections et les pipes,
4. effectue les substitutions de commandes,
5. exécute la commande.

C'est le **mécanisme d'évaluation du shell**. Ces étapes sont à garder en émoire pour toute commande saisie au clavier ou bien enregistrée dans un script. Ce n'est pas ce qui est saisi qui sera exécuté **mais le résultat de l'évaluation de l'expression**.

Il existe plusieurs shells sous UNIX :

- le Bourne Shell (noté "`sh`") ancêtre de tous les shells, utilisés seulement pour l'écriture de procédures. Il n'offre aucune facilité pour l'emploi en mode interactif (pas d'historique de commandes, pas de rappels avec les flèches, etc.).
- le C Shell (noté "`csh`") plutôt concu pour une interface avec les utilisateurs. Il permet le rappel des commandes avec les flèches, de gérer une historique des commandes, etc. Sa syntaxe se rapproche de celle du langage C même si le "*C*" veut dire "*California*" [1].

1. Ce shell a été développé à l'université "*UCB*", University of California – Berkeley.

- le Korn Shell (noté "`ksh`") est une extension du Bourne Shell avec une partie des possibilités du C Shell.
- le Bourne Again Shell (noté "`bash`") est une variante du Bourne Shell, disponible dans le domaine public.
- le TC Shell (noté "`tcsh`") est une extension du C Shell. Il vient comme un remplaçant naturel du C Shell pour faire face à la *concurrence* du Korn Shell.

Le Bourne Shell, le Korn Shell et le Bash Shell sont compatibles entre eux (compatibilité Bourne Shell vers Korn Shell). Le C Shell et le TC Shell sont compatibles entre eux. Par contre, ces deux familles ne sont pas compatibles entre elles. Il est toutefois possible d'exécuter des procédures Bourne Shell alors que le shell de login est le C Shell (sous certaines restrictions quant au mode de lancement).

1.2 Zones mémoire code, variables locales, variables d'environnement du shell

1.2.1 Description

Lors de la création d'un processus, trois zones mémoires lui sont affectées :

la zone "`CODE`"
: Celle-ci représente la zone mémoire allouée au code exécutable qui doit être déroulé par le processus.

la zone "`DATA`"
: Celle-ci représente la zone mémoire réservée pour les données propres au code exécutable.

la zone "`ENV`"
: Celle-ci représente une zone mémoire réservée pour les données propres au code exécutable. Elle est aussi appelée "*zone d'environnement*".

En faisant l'analogie avec un programme source, la zone "`CODE`" correspond aux instructions du programme, tandis que les zones "`DATA`" et "`ENV`" correspondent aux zones associées aux déclarations de variables, la zone "`ENV`" référençant les variables globales.

Lors de la création d'un sous processus, UNIX duplique l'environnement en ne gardant que les variables globales. Pour exécuter une commande, le shell crée un sous processus dans lequel il substitue le code par le code de la commande à exécuter. La méthode suivie est illustrée à la figure 1.1.

L'appel système "`fork()`" crée le processus et ne garde que la zone mémoire "`ENV`". L'appel système "`exec()`" exécute la commande dans le processus créé.

Remarque 1.1 :

"`exec`" est aussi une commande du shell qui a la même fonctionnalité.

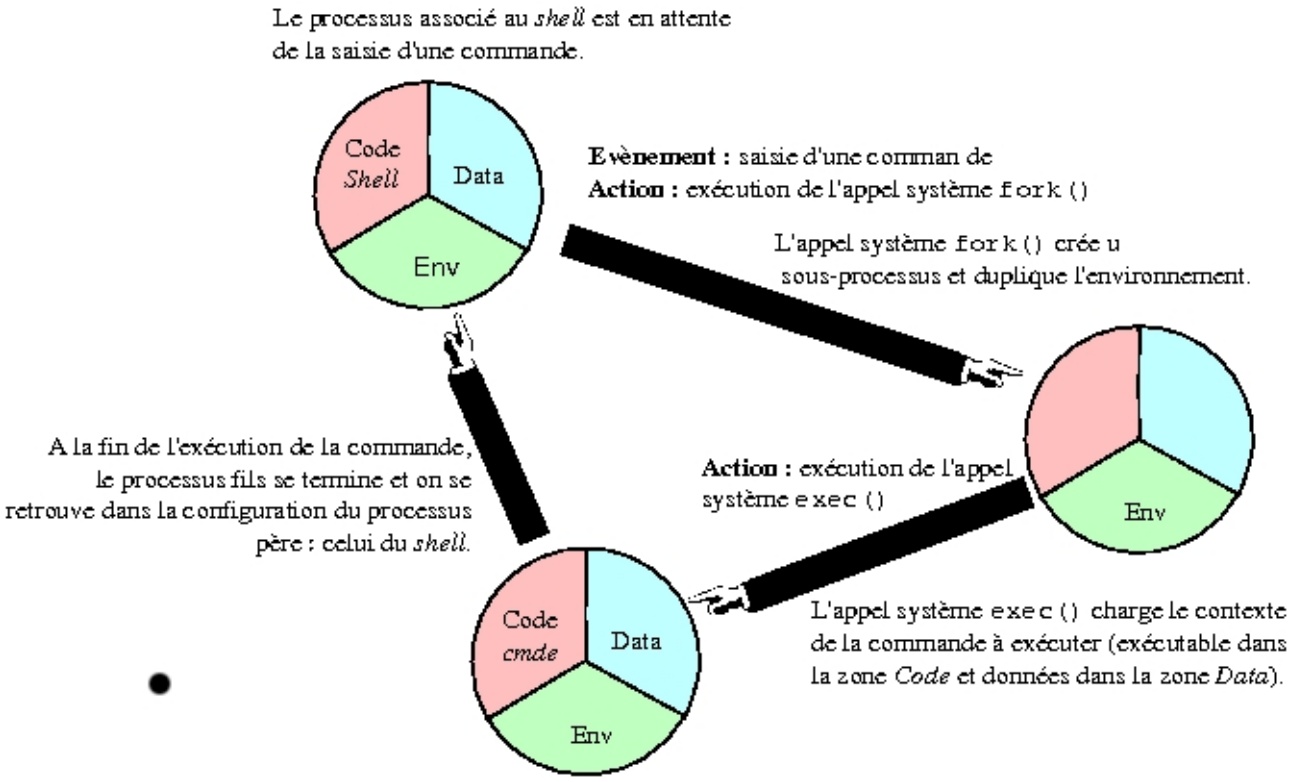


FIG. 1.1 – *Exécution d'une commande sous* UNIX

La commande "`% exec ls`" exécute "`ls`" dans le même processus du shell (substitue le code du shell par le code de la commande "`ls`" dans la zone mémoire "`CODE`" et s'arrête dès que son exécution est terminée. **On est donc déloggé**.

1.2.2 Les commandes de gestion des variables du shell

Syntaxe :

```
set

variable=valeur

unset variable

export variable

printenv

env
```

La commande "`set`" sans argurments affiche la liste des variables locales au shell

La commande "`unset`" suivi d'un nom de variable permet d'effacer celle-ci de la zone des variables locales du shell.

La commande "`export`" suivie du nom d'une variable, permet de placer une variable définie de la zone locale au shell vers la zone d'environnement (exportation de la variable).

Les commandes "`env`" et "`printenv`" listent les variables de la zone d'environnement et les valeurs qui leur sont affectées.

1.2.3 Variables usuelles

Le tableau 1.1 donne la liste des variables les plus usuelles du shell UNIX.

Variable	*Signification*
`PATH`	Référence les chemins d'accès scrutés lors de l'exécution d'une commande.
`HOME`	Référence le répertoire de *login*. C'est le répertoire par défaut de la commande "`cd`".
`PS1`	Invite du shell.
`PS2`	Invite secondaire du shell. Lorsque vous demandez à ce qu'une commande se poursuive après un retour chariot, c'est le contenu de cette de cette variable qui sera affiché.
`LANG`	Langue utilisée pour les messages.
`HISTSIZE`	Nombre de commandes à mémoriser dans l'historique (Korn Shell uniquement).

TAB. 1.1 – Liste des variables les plus usuelles.

1.2.4 Visualisation d'une variable

Pour rappeler le contenu d'une variable (locale ou d'environnement), il suffit de faire précéder son nom par le caractère "$". Par conséquent :

- pour référencer la variable, il suffit de préciser son nom.
- pour référencer son contenu, il suffit de préciser son nom précédé du caractère "$".

Exemple 1.1 :

```
sh_ksh% my_var=schmoll
sh_ksh% echo $my_var
schmoll
```

1.3 Exécution d'une commande

La plupart des commandes sont des exécutables placés dans `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, etc. Ce sont des commandes UNIX ou **commandes externes au shell**. D'autres commandes comme "`cd`", "`echo`", "`pwd`" font partie intgrante de l'interpréteur de commandes. Ce sont des **commandes internes au shell**.

Les commandes UNIX sont exécutées dans un sous processus. Comme les exécutables peuvent se trouver dans différents répertoires, le shell doit savoir où les chercher. C'est la variable "`PATH`" qui définit la liste des répertoires à scruter et l'ordre dans lequel le shell doit faire la recherche.

Remarque 1.2 :

Les commandes internes au shell ne sont pas exécutées dans un processus fils.

1.4 Redirection des entrées/sorties

1.4.1 Introduction

Chaque processus sous UNIX posséde trois canaux de communication :

Canal de communication	Fichier	Numéro logique	Analogie OpenVMS
Entrée standard	`stdin`	0	`SYS$INPUT`
Sortie standard	`stdout`	1	`SYS$OUTPUT`
Sortie d'erreurs standard	`stderr`	2	`SYS$ERROR`

La redirection de ces canaux est très utilisée sous UNIX. En effet, beaucoup de commandes écrivent leur résultat par défaut sur la sortie standard (comme les filtres, par exemple). Le seul moyen de l'avoir dans un fichier est de rediriger la sortie standard. D'autres commandes lisent systématiquement sur leur entrée standard (comme les filtres). Si l'on veut qu'elles prennent un fichier comme argument, il faudra rediriger l'entrée standard.

Les syntaxes utilisées pour les redirections sont explicitées aux sections 1.4.2, 1.4.3 et 1.4.4.

1.4.2 Redirection de l'entrée standard (`stdin`)

Syntaxe :

```
commande < nouvelle-entree-standard
```

Il est possible de rediriger l'entrée de toute commande devant lire des données depuis l'entrée standard, afin que la lecture se fasse sur un fichier grâce au symbole "<".

Exemple 1.2 :

```
% mail machin < fichier
```

1.4.3 Redirection de la sortie standard (`stdout`)

Syntaxe :

`commande > nouvelle-sortie`	(Création/Réécriture)
`commande >> nouvelle-sortie`	(Ajout)

Il est possible de rediriger la sortie de toute commande devant écrire sur la sortie standard afin que l'écriture se fasse sur un fichier. L'écriture peut se faire de deux façons :

- écriture dans un fichier et écrasement si le fichier existe déjà,
- écriture à la suite d'un fichier déjà existant.

Si une ligne de commande contient le symbole de redirection de la sortie standard ">" suivi d'un nom de fichier, celle-ci sera redirigée dans le fichier spécifié au lieu du terminal. Deux cas peuvent se présenter :

- Si le fichier n'existe pas au moment où la commande est exécutée, il est créé.
- Si le fichier existait, alors son contenu est écrasé par la sortie standard de la commande. Si on souhaite que celle-ci vienne s'ajouter à la suite, afin de préserver son contenu initial, il suffit d'utiliser le double symbole ">>". Dans le cas où le fichier n'existait pas, il sera créé.

Exemple 1.3 :

```
% ls > fic
% date > who.log
% who >> who.log
```

1.4.4 Redirection de la sortie d'erreurs standard (`stderr`)

Syntaxe :

`commande 2>fichier`	(Création/Réécriture)
`commande 2>>fichier`	(Ajout)

La plupart des commandes UNIX produisent des messages de diagnostic si un problème survient en cours d'exécution. La sortie des messages d'erreur se fait sur la sortie d'erreurs standard, qui, par défaut, est associée à l'écran.

La sortie de messages d'erreur peut être redirigée indépendamment de la sortie standard. Ceci évite d'avoir les messages d'exécution normale et les messages de diagnostic entrelacés dans un même fichier.

Pour rediriger la sortie d'erreurs standard dans un fichier, on utilise les chaînes "2>" et "2>>" suivie du nom du fichier. Il ne doit pas y avoir d'espace entre le "2" et le ">". Comme pour la redirection de la sortie standard, si le fichier n'existe pas, il est créé, sinon il est écrasé. Si l'on veut que les messages de diagnostics viennent s'ajouter en fin de fichier, il faut utiliser le double symbole de redirection "2>>".

Exemple 1.4 :

```
sh% cp fic1 fic2 2>fic
sh% cp fic1 fic2 2>>fic
```

1.4.5 Redirection d'une sortie standard vers une autre sortie standard

Le principe reste identique. Il suffit de rediriger une sortie vers un fichier (ou canal logique). Lorsque l'on veut référencer le canal associé à une sortie standard (idem pour l'entrée et la sortie d'erreurs standard) comme un fichier, il suffit de faire précédé son numéro logique par le caractère «&".

Par exemple, si la sortie d'erreurs standard doit être redirigée vers la sortie standard, il suffira d'écrire :

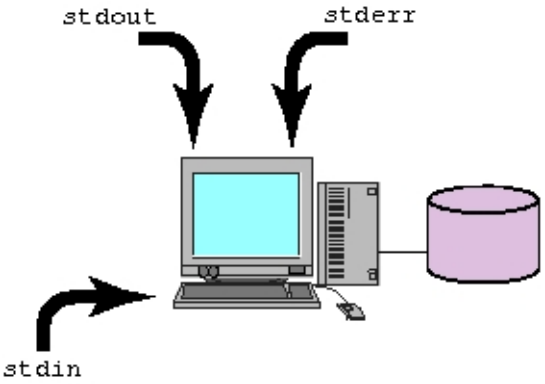
```
commande 2>&1
```

Dans le cas où la sortie standard et la sortie d'erreurs standard doivent être redirigées vers un même fichier, il faudra bien analyser le processus à mettre en jeu.

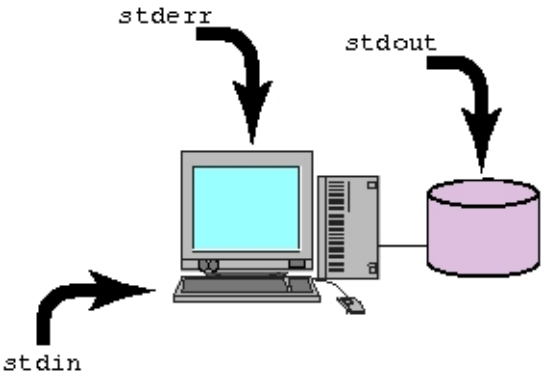
Dans le premier cas :

```
commande 2>&1 >fichier
```

le shell va exécuter les étapes suivantes :



la sortie d'erreurs standard est redirigée vers la valeur courante de la sortie standard, **donc l'écran**.

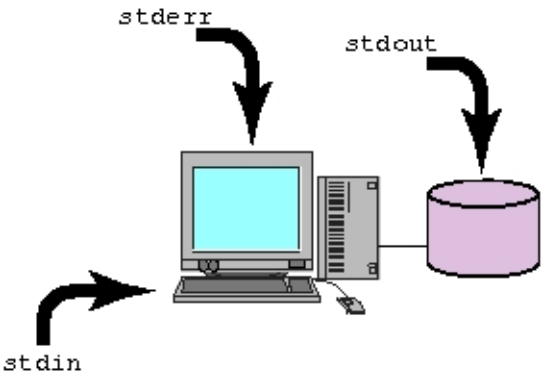


la sortie standard vers un fichier. **Donc seule la sortie standard a été redirigée vers un fichier**.

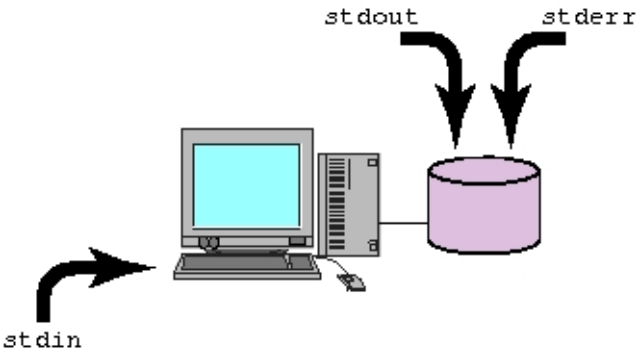
Dans le deuxième cas :

```
commande >fichier 2>&1
```

le shell va exécuter les étapes suivantes :



la sortie standard est redirigée vers un fichier,



la sortie d'erreurs standard est redirigée vers la valeur courante sur laquelle pointe la sortie standard, **donc le fichier**. En conséquence, **la sortie standard et la sortie d'erreurs standard ont bien été redirigées vers un même fichier**.

Le second modèle est donc celui à retenir.

1.4.6 Redirection de la sortie standard d'une commande dans l'entrée standard d'une autre

Syntaxe :

Commande A		*Commande B*
Doit écrire sur `stdout`		Doit lire sur `stdin`

Le symbole "|" appelé "*pipe*", est utilisé pour relier deux commandes entre elles. La sortie standard de la commande à gauche du symbole "|" est utilisée comme entrée standard de la commande de droite. La figure 1.2 illustre le principe utilisé pour ce type de redirection.

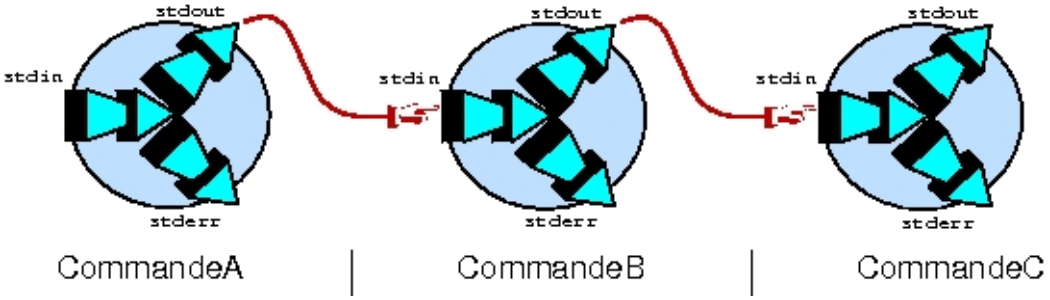


FIG. 1.2 – *Enchainement de commandes, mécanisme du pipe.*

C'est pourquoi :

- toute commande située à gauche du symbole "|" doit produire une sortie sur "`stdout`",
- toute commande située à droite du symbole "|" doit effectuer une lecture depuis "`stdin`",
- toute commande entre les deux symboles "|" doit écrire sur "`stdout`" et lire sur "`stdin`". **C'est donc un filtre**.

La redirection d'entrée/sortie permet le passage d'informations entre un processus et un fichier. Les *pipes* permettent le passage d'informations entre processus. Ils constituent une solution pour utiliser la sortie d'une commande en entrée d'une autre commande sans passer par un fichier intermédiaire.

UNIX repose sur le principe suivant. Chaque commande doit réaliser une seule chose. C'est la combinaison de ces commandes élémentaires, grâce au mécanisme des *pipes* qui permet l'obtention de résultats élaborés.

1.5 Génération de noms de fichiers - Les métacaractères

1.5.1 Introduction

Les métacaractères ne sont pas des "*wildcards*". Un wildcard est interprété par une commande pour générer certains noms de fichiers. Par contre les métacaractères sont interprétés directement par le shell **avant l'exécution de la commande**. Celle-ci reçoit des noms de fichiers, comme si vous les aviez tapés au clavier.

Les métacaractères ne sont pas une aide à la frappe au clavier. Les métacaractères disponibles sont :

`?`	Masque un caractère, sauf le point en première position.
`[]`	Définit une classe de caractères : `-` pour définir une suite, `!` pour exprimer une exclusion séparateur n'est utilisé pour exprimer une liste.
`*`	Masque toutes chaînes de caractères, sauf le point en première position.

Aucun

Remarque 1.3 :

Les expressions utilisant les métacaractères suivent les règles des expressions régulières UNIX.

Les métacaractères ne masquent jamais les fichiers cachés. Le point en début du nom d'un fichier doit être tapé explicitement.

Le tableau 1.2 donne les équivalences entre UNIX OpenVMS et MS-DOS pour l'utilisation des métacaractères sous UNIX.

1.5.2 Utilisation du métacaractère "?"

Le point d'interrogation "?" masque un et un seul caractère sauf le point en première position.

Exemple 1.5 :

UNIX	OpenVMS	MS-DOS
*	*	*
?	%	?
[]	pas d'équivalence	pas d'équivalence

TAB. 1.2 – *Équivalence pour l'utilisation des métacaractères entre* UNIX, *OpenVMS et MS-DOS.*

`echo ???`	Masque tous les noms de fichiers à trois caractères.
`ls A?C`	Masque tous les noms de fichiers à trois caractères, dont le premier est "A" et le dernier est "C".

1.5.3 Utilisation des métacaractères "[]"

Les crochets ouvrants et fermants []"[]" définissent une classe de caractères, dont un et un seul sera masqué.

Exemple 1.6 :

`echo [abc]??`	Tous les fichiers dont le nom fait 3 caractères et dont la première lettre est soit "a", soit "b", soit "c".
`echo ?[a-zA-Z]`	Tous les fichiers dont le nom fait 2 caractères et dont le dernier caractère est une lettre (majuscule ou minuscule).
`echo [!a-zA-Z]??`	Tous les fichiers dont le nom fait 3 caractères et dont le premier n'est pas une lettre.

1.5.4 Utilisation du métacaractère "*"

L'étoile "*" masque toute chaîne de caractères, même vide, sauf le point en première position.

Exemple 1.7 :

```
echo *
echo .*
echo *a*
```
etc.

1.6 Les quotes et les caractères spéciaux

1.6.1 Introduction

Nous avons vu dans les paragraphes précédents, les caractères spéciaux suivants :

`$`	utilisé pour la substitution d'une variable,
`? [] *`	utilisés pour la génération des noms de fichiers par le Shell,
`< > >>`	utilisés pour la redirection des entrées/sorties,
SPACE	utilisé comme séparateur par le Shell,
`|`	utilisé dans les *pipes*.

Le contexte ne suffit pas toujours à déterminer quel sens donner au caractère. C'est pourquoi il est nécessaire d'avoir un moyen permettant d'échapper au sens spécial du caractère et forcer celui-ci à être considéré comme tel. **C'est le mécanisme de recours aux quotes**.

1.6.2 Les caractères d'échappements

back slash "\"

Annule la signification particulière du caractère immédiatement après.

Exemple :

`echo \$ABC`	renvoie la chaîne "`$ABC`".
`echo abc \` RETURN	annule l'interprétation du "`<RETURN>`", il n'y a donc pas d'interprétation de la commande d'où l'apparition d'un "*prompt secondaire*" pour continuer la commande.

simple quote "'"

Annule l'interprétation de tous les caractères à l'intérieur des quotes à l'exception de la simple quote elle-même.

Exemple :

`echo '$ABC > Bonjour'`	renvoie la chaîne "`$ABC > Bonjour`" à l'écran.

doubles quotes """

Annule l'interprétation de tous les caractères entre les doubles quotes, à l'exception des caractères \, ", `$` et ` (back quote).

Exemple :

```
ABC=Salut
echo "$ABC > Bonjour"
```

renvoie la chaîne "`Salut > Bonjour`" à l'écran.

1.6.3 Résumé

Symbole	Type	Action	Exception
`$`	Caractère spécial	Substitue la valeur d'une variable.	*N.A.*
`>`, `>>`	Caractère spécial	Redirige la sortie standard vers un fichier.	*N.A.*

Suite page suivante ···

Suite de la page précédente.			
Symbole	Type	Action	Exception
<	Caractère spécial	Redirige l'entrée standard à partir d'un fichier.	*N.A.*
|	Caractère spécial	Combine plusieurs commandes dans un *pipe*.	*N.A.*
SPACE	Caractère spécial	Délimiteur dans une commande.	*N.A.*
?	Métacaractère	Masque un caractère, sauf le point en première position.	*N.A.*
[]	Métacaractère	Définit une classe de caractères à masquer.	*N.A.*
*	Métacaractère	Masque toutes chaînes de caractères, sauf le point en première position.	*N.A.*
\ (*back slash*)	Caractère d'échappement	Annule l'interprétation du caractère suivant.	*Aucune.*
' (*simple quote*)	Caractère d'échappement	Annule l'interprétation de tous les caractères entre les "'".	*Aucune.*
" (*double quotes*)	Caractère d'échappement	Annule l'interprétation de tous les caractères entre les """.	\ (*back slash*), "$" (symbole dollar), """ (double quotes), "`" (back quote).

Chapitre 2

Le mode multi-tâche

2.1 Tâche de fond – le *background*

Syntaxe :

```
% ligne-de-commande &
% jobs
```

Le caractère "`&`" permet d'exécuter la commande en arrière plan (*background*) et permet de lancer pendant ce temps là d'autres commandes en avant plan. Lors de la déconnexion, tous les processes en arrière plan meurent. En effet, ils ne peuvent qu'exister que si leur père existe.

Lorsqu'une commande est lancée en arrière plan, le shell reporte à l'utilisateur le numéro de PID (*Process IDentifier*) identifiant le processus lancé en arrière plan avant de renvoyer le prompt. Une commande s'exécutant en arrière plan **ne peut pas** être arrêtée avec les touches BREAK, CTRL–C, etc. Par contre, une sortie de session (*logout*) tuera tous les processus s'exécutant en arrière plan.

Il est important qu'un processus lancé en arrière plan ait ses entrées/sorties redirigées explicitement. Il est possible de voir les processes lancés en arrière plan avec la commande "`jobs`".

Remarque 2.1 :

CTRL–*Z* **ne sert pas à interrompre un process mais à le mettre en attente en arrière plan**. *Si vous voulez interrompre une commande, utilisez "*CTRL–*C". Pour plus de renseignements sur les touches d'interruption, reportez vous à la section 2.3.3.*

2.2 Substitution de commande - caractère *back quote*

Syntaxe :

```
‘commande‘
```

La substitution de commande se fait en utilisant le caractère back quote ("‘") pour lancer des commandes à insérer dans une chaîne de caractères.

La substitution de commandes dans une chaîne de caractères est une autre facilité offerte par le shell. Elle permet de capturer la sortie d'une commande et de l'assigner à une variable ou de l'utiliser comme un argument d'une autre commande. Comme beaucoup de commandes UNIX génèrent une sortie, la substitution de commandes peut être très intéressante.

A propos de l'efficacité, il est plus rapide, dans certains cas, d'effectuer une substitution de commande plutot que de passer par une redirection.

Exemple 2.1 :

```
% echo "my current shell is ‘basename $SHELL‘"
% current_dir=‘pwd‘
% cd /etc
% cd $current_dir
```

2.3 Commandes associées

2.3.1 Commande "kill"

Syntaxe :

```
kill [-sig_no] PID [PID ...]
```

La commande "kill" peut être utilisée pour tuer n'importe quelle commande, y compris des commandes lancées en arrière plan. "kill", plus précisément, envoie un signal aux processus désignés. L'action effectuée par défaut pour un process est de mourir sur réception de certains signaux. Pour transmettre un signal à un processus il faut en être le propriétaire; "kill" ne peut être utilisé pour tuer le process d'un autre utilisateur à moins d'être le *super user* (utilisateur "root").

Par défaut "kill" transmet le signal numéro 15 au process spécifié. Dans le monde UNIX, il n'est pas possible de véritablement tuer un process comme on pourrait le faire avec Windows–NT et OpenVMS. UNIX, en fait, envoie une requête au processus pour qu'il se termine de lui-même. Il est possible d'immuniser certaines commandes contre les signaux de terminaison. La méthode la plus efficace pour tuer un process est l'utilisation du signal 9.

Pour avoir la liste des signaux et leur action, reportez-vous à "signal(2)".

Exemple 2.2 :

```
% cat /usr/man/man1/* >big.file &
[[1] 1234
% kill 1234
[1] + Terminated cat /usr/man/man1/* >big.file
```

2.3.2 Commande "wait"

Syntaxe :

```
wait [PID]
```

La commande "wait" permet de suspendre l'exécution d'un shell script jusqu'à ce que le processus, dont le *PID* est spécifié en argument, se termine. Si aucun *PID* n'est spécifié, on attendra, dans ce cas, que tous les processus lancés en arrière plan soient terminés.

Exemple 2.3 :

```
% cat /usr/man1/* >big.file &
% find / -print >big.file2 &
% wait
% echo "cat et find sont termin{\'e}s ..."
```

2.3.3 Les commandes "fg" et "bg"

Syntaxe :

`bg PID`	`bg` = back ground (arrière plan)
`bg % numéro`	
`fg PID`	`fg` = foreground (avant plan)
`fg % numéro`	

Nous avons vu qu'avec la touche "CTRL-Z", il était possible de faire passer une commande **en attente** en arrière plan. L'exécution est suspendue mais le process est toujours actif. Si vous désirez qu'elle continue à s'exécuter, toujours en arrière plan, utilisez la commande "bg".

Pour refaire passer en avant plan une commande qui s'exécute en arrière plan utiliser la commande "fg". "fg" permet aussi de réactivé une commande suspendue par "CTRL-Z". Le processus associé devient alors le processus courant jusqu'à ce que cette commande soit achevée.

Pour plus de renseignements, reportez vous à "sh(1)".

Exemple 2.4 :

```
% find / -print >/dev/null
CTRL-Z
[1] 1234 Suspended
% bg %1
[1] 1234 find / -print >/dev/null &
% ls
```

```
...

%fg %1
find / -print >/dev/null
```

Équivalence :

UNIX	OpenVMS
CTRL-Z	SET PROCESS/SUSPEND
fg	RESUME
bg	Pas d'équivalence.

2.3.4 Commandes "at"

Syntaxe :

```
at [-c | -s | -k] [-m] [-q queuename] [-f file]
      date [increment] [command | file]

at -r job_number ...

at -l [-q queuename] [user ...]
```

La commande "at" permet de différer l'exécution de travaux. Elle lit sur son entrée standard les commandes qu'elle doit lancer à la date spécifiée.

Au lieu d'envoyer les ordres à exécuter sur l'entrée standard, vous pouvez :

- spécifier directement la commande à la suite. Dans ce cas, l'entrée standard de la commande qui aura été saisie, devra être prise à partir d'un fichier avec les mécanismes décrits à la section 1.4.2.
- donner le nom d'un fichier contenant les commandes à exécuter. Celui-ci n'est pas lu à la place de l'entrée standard mais est traité comme une procédure de commande. Il doit donc être accessible en exécution par le Shell (cf. sections 2.3 et 1.3).

La commande " batch" exécute les travaux seulement lorsque le niveau de la charge du système le permet. La commande "at" redirige automatiquement sa sortie standard et sa sortie d'erreurs standard dans la boite aux lettres du courrier électronique. Par défaut, la commande "at" utilise le Bourne Shell comme interpréteur de commandes. Si vous désirez changer l'interpréteur de commande par défaut, les options suivantes devront être spécifiées :

Option	Shell
"-c"	C Shell
"-k"	Korn Shell

Le paramètre "date" a le format suivant :

```
[[CC]AA]MMJJhhmm[.ss]
```

avec :

`CC` les deux chiffres des centaines de l'année,
`AA` les deux derniers chiffres de l'année,
`MM` le mois (01-12),
`JJ` le jour (01-31),
`hh` l'heure (00-23),
`mm` les minutes (00-59),
`ss` les secondes (00-59).

`"CC"` et `"AA"` sont optionnels, l'année en cours est prise par défaut.

Le paramètre `"date"` admet aussi les formats suivants :

- La commande `"at"` interprète les valeurs composées d'un ou deux chiffres comme étant des heures. Elle interprète les valeurs composées de quatre chiffres comme étant des heures et des minutes. Vous pouvez éventuellement séparer les heures des minutes par le caractère deux points `":"`. Dans ce cas, le format sera `"hh:mm"`.
- Vous pouvez indiquer le suffixe `"am"`, `"pm"` ou `"zulu"`. Si vous ne spécifiez ni `"am"` ni `"pm"`, la commande `"at"` utilise une horloge de 24 heures. Si vous spécifiez `"zulu"`, le temps universel[1] est utilisé.
- L'un des mots clefs suivants :
  - ⋆ `"noon"` pour *midi*,
  - ⋆ `"midnight"` pour *minuit*,
  - ⋆ `"now"` pour représenter l'instant présent,
  - ⋆ `"A"` en abréviation de `"am"`,
  - ⋆ `"P"` en abréviation de `"pm"`,
  - ⋆ `"N"` en abréviation de `"noon"`,
  - ⋆ `"M"` en abréviation de `"midnight"`.

Remarque 2.2 :

Le mot clef "`now`" peut être utilisé uniquement si vous indiquez le paramètre "`date`" ou "`increment`". Dans le cas contraire, le message "`too late`" s'affiche.

Le paramètre `"date"` permet d'indiquer un nom de mois et un numéro de jour (éventuellement un numéro d'année précédé d'une virgule), ou un jour de la semaine. La commande `"at"` reconnaît deux jours particuliers : il s'agit de `"today"` et de `"tomorrow"`. Si l'heure indiquée est postérieure à l'heure d'entrée de la commande, `"today"` est utilisée par défaut en tant que paramètre `"date"`. Dans le cas contraire, celui-ci prend la valeur `"tomorrow"`. Si vous indiquez un mois antérieur à celui en cours sans préciser d'année, l'année suivante est utilisée par défaut.

Le paramètre facultatif `"increment"` peut prendre l'une des valeurs suivantes :

- le signe plus `"+"` suivi d'un nombre et de l'un des mots suivants : `"minute[s]"`, `"hour[s]"`, `"day[s]"`, `"week[s]"`, `"month[s]"` ou `"year[s]"` (ou tout équivalent en anglais).

1. Le temps universel correspond au fuseau horaire de Greenwich.

– le mot "`next`" suivi de l'un des mots suivants : "`minute[s]`", "`hour[s]`", "`day[s]`", "`week[s]`", "`month[s]`" ou "`year[s]`" (ou tout équivalent en anglais).

Options :

`-c`	Indique que le C Shell sera utilisé pour exécuter le travail.
`-k`	Indique que le Korn Shell sera utilisé pour exécuter le travail.
`-l`	Liste les travaux à exécuter.
`-m`	Envoie un message à l'utilisateur après l'exécution de la commande.
`-q` *queuename*	Spécifie la file d'attente dans laquelle on effectue le travail.
`-s`	Indique que le Bourne Shell sera utilisé pour exécuter le travail.
`-r` *job*	Supprime des travaux programmés par la commande "`at`" (le paramètre "`job`" correspond à un numéro affecté par la commande).

Par défaut, l'attribution des files d'attente se fait de la façon suivante :

file d'attente	type de travaux
`a`	travaux soumis avec la commande "`at`".
`b`	travaux "*batch*".
`c`	travaux soumis par le "*cron*".
`d`	travaux "*sync*".
`e`	travaux Korn Shell.
`f`	travaux C Shell.

Remarque 2.3 :

Les travaux "batch" sous Unix *ont un sens différent de celui habituellement entendu sous Open*Vms*. Ceux-ci ne sont exécutés seulement si le système dispose de suffisamment de ressources. Avec Open*Vms*, il est possible d'affecter un niveau de priorité des processus pris en compte par l'ordonnanceur du système (scheduler). Il est donc possible d'avoir une file d'attente offrant un niveau bas de ressource (priorité faible), les travaux ne seront donc exécutés seulement si le système dispose de suffisamment de ressources. Par contre, il est aussi possible d'avoir une file d'attent avec un niveau élevé. Dans ce cas, les travaux passeront en priorité. Ceci peut être utile en fonction de l'environnement d'exploitation.*

Pour plus de renseignements, reportez-vous à "`at(1)`".

Exemple 2.5 :

Pour programmer l'exécution d'une commande à partir du terminal, entrez une commande semblable à l'un des exemples suivants.

Si "uuclean" se trouve dans le répertoire courrant (ou dans un répertoire spécifié dans la variable "PATH") :

```
% at 5 pm Friday uuclean
% at now next week uuclean
```

Si "uuclean" se trouve dans le répertoire "$HOME/bin/uuclean" :

```
% at now + 2 days $HOME/bin/uuclean
% at now + 2 days
$HOME/bin/uuclean
```

CTRL-D

Remarque 2.4 :

Lorsque vous indiquez, sur la ligne de commande, un nom de commande comme dernier élément, vous devez spécifier le chemin d'accès complet si celle-ci ne se trouve pas dans le répertoire courrant. En outre, la commande "at" ne prend en compte aucun argument pour les commandes à lancer.

Exemple 2.6 :

Pour exécuter la commande "uuclean" le 24 Janvier à 15 heures, entrez l'une des commandes ci-dessous :

```
% echo uuclean | at 3:00 pm January 24
% echo uuclean | at 3pm Jan 24
% echo uuclean | at 1500 jan 24
```

Équivalence :

UNIX	OpenVMS
at	submit

2.4 Répétition de tâches : crontab

2.4.1 Introduction – Syntaxe

crontab est un utilitaire utilisé pour exécuter un certain nombre de tâches répétitives. Chaque utilisateur possède sa propre table de tâches. Celle-ci est enregistrée dans un répertoire du système d'exploitation (/var/spool/crontab").

Cette fonctionnalité du système est associé à :

- la commande "crontab",
- un fichier de description des tâches à effectuer.

La commande crontab obéit à la syntaxe explicitée ci-après.

Syntaxe :

```
crontab [ -u user ] file
crontab [ -u user ] { -l | -r | -e }
```

Le premier format de la commande permet d'enregistrer les commandes à effectuer . Le second format permet de consulter et de maintenir la table de

description des tâches. Les options disponibles sont :

`-u user`
: L'option "`-u` *user*" permet de préciser pour quel utilisateur, défini sur le système, les opérations sont à effectuer. Par défaut, la commande `crontab` concernera votre propre table de description.

`-l`
: L'option "`-l`" affiche le contenu de la table de description de l'utilisateur concerné sur la sortie standard.

`-r`
: L'option "`-r`" réinitialise la table de description de l'utilisateur concerné : toutes les tâches qui y sont définies seront supprimées.

`-e`
: L'option "`-e`" permet d'éditer la table de description de l'utilisateur concerné. L'éditeur utilisé sera celui précisé par le contenu de la variable d'environnement "`EDITOR`" ou bien la variable d'environnement "`VISUAL`". Les modifications prendont effet immédiatement après la sortie de l'éditeur de texte.

Pour plus de précisions, reportez-vous à `crontab(1)`.

2.4.2 Fichier de description de tâches

Le fichier de description des tâches va contenir les instructions nécessaires au processus de supervision (processus "`cron(8)`"). Celui décrira les opérations de la façon suivante "*exécute cette commande à cette heure là ce jour-ci*".

Dans ce fichier,

- toute ligne commençant par le caractère "`#`" est ignorée (ligne de commentaires),
- les lignes blanches sont ignorées,
- les espaces et les tabulations en début de ligne sont ignorés.

Remarque 2.5 :

Il n'est pas possible de mettre des commentaires au milieu d'une ligne. Le signe de commentaire doit donc être le premier caractère de la ligne.

Chaque ligne de description, hormis les lignes de commentaires, peuvent , soit définir des variables d'environnement pour l'ensemble des tâches à effectuer, soit des commandes à exécuter aux instants spécifiés. La définition d'une variable d'environnement dans le fichier obéit à la syntaxe suivante :

nom `=` *valeur*

où "*nom*" représente le nom de la variable d'environnement et "*valeur*" représente la valeur qui lui est affectée. Les espaces de part et d'autre du signe *égual* "`=`" sont optionnels. Si la valeur à affecter à la variable est une chaine de caractères, il est préférable de la mettre entre simple quotes ("`'`" ou double quotes ("`"`").

Pour toutes les commandes qui seront exécutées :

- **leur sortie standard et leur sortie d'erreur standard redirigées vers la boite aux lettres de l'utilisateur,**
- **leur entrée standard devra être explicitement précisée si cela est necéssaire.**

Certaines variables d'environnement sont automatiquement positionnées par le gestionnaire de tâches `cron(8)` :

Variable	Description
`SHELL`	représente l'interpréteur de commande par défaut. Cette variable est initialisée à `"/bin/sh"`, donc le Bourne Shell.
`LOGNAME`	représente le nom de connexion de l'utilisateur. Par défaut, elle correspond à votre entrée dans la base des utilisateurs.
`USER`	a exactement les mêmes fonctionnalités que la variable `"LOGNAME"`. Cependant, certains utilitaires issus des systèmes BSD[2] utilisent `"USER"` plutôt que `"LOGNAME"`.
`HOME`	représente votre répertoire de connexion. Par défaut, il correspond à celui défini dans les caractéristiques de l'utilisateur.
`MAILTO`	En plus des variables `LOGNAME`, `HOME` et `SHELL`, le processus `"cron(8)"` examine le contenu de la variable d'environnement `"MAILTO"` afin de savoir s'il est nécessaire d'envoyer un message donnant le résultat de l'exécution des commandes. Si cette variable est définie et non vide, elle doit contenir l'adresse courrier de la personne à qui envoyer les messages. Si elle est définie **mais vide**[3], aucun message ne sera envoyé. Au cas où aucune option se serait précisé grâce à la variable `"MAILTO"`, le résultat produit sur la sortie standard est envoyé dans la boite aux lettres de l'utilisateur possédant cette liste de tâches à exécuter.

Remarque 2.6 :

Cette fonctionnanlité offerte grâce à la variable `"MAILTO"` peut être très utile si vous décidez d'utiliser un autre serveur de messagerie que `"/usr/lib/sendmail"`. Dans ce cas, `"/bin/mail"` sera employé pour assurer l'acheminement des messages. `"/bin/mail"` n'utilisera pas la notion d'aliases de `"/usr/lib/sendmail"`.

Les lignes de description de tâches se composent de plusieurs champs séparés par des espaces ou tabulations. Chaque ligne possède :

- cinq champs précisant la date et l'heure à laquelle la commande doit être lancée,
- dans le cas où la table de description est celle de l'administrateur, le nom de l'utilisateur pour lequel on lance la commande[4],

4. Cette fonctionnalité n'est pas disponible sur tous les Unix. On la trouvera par exemple

– la commande à lancer.

Le service "`cron(8)`"[5] examine toutes les minutes, les tables de description. Lorsque la date précisée dans une entrée d'une table de description de tâches, le service "`cron(8)`" crée un sous processus sous l'identité de la personne et lance la commande à l'intérieur de ce contexte. La spécification de la date et de l'heure se compose des cinq informations suivantes :

Champ	Valeurs autorisées
minute	0-59
heure	0-23
jour du mois	1-31
mois	1-12
jour de la semaine	0-7[6]

Si l'un de ces champs contient le caractère "`*`", celui-ci ne doit pas être pris en compte, ou bien l'une des bornes inférieure ou supérieure doit être considérée dans la détermination de la date. Par exemple :

`0 1 * * *`	indique : "*tous les jours à 01:00*"
`0 0 * * 1`	indique : "*tous les lundis à minuit*"
`* * * 2 3`	indique : "*tous les mardis du mois de février à minuit*"

Pour chaque champ, il est possible de donner :

– une séquence de nombres séparés par des virgules,
– des plages de nombre, en spécifiant les bornes inférieurs et supérieures séparées par le caractère "`-`",
– de combiner les deux possibilités.

Exemple 2.7 :

`1,5`	correspond à la séquence : "*1, 5*"
`8-10`	correspond à la séquence : "*8, 9, 10*"
`0-5,7,9`	correspond à la séquence : "*0, 1, 2, 3, 4, 5, 7, 9*"
`0-4,8-12`	correspond à la séquence : "*0, 1, 2, 3, 4, 8, 9, 10, 11, 12*"

Remarque 2.7 :

Aucun espace ne doit être introduit dans les cinq champs associés à la date et à l'heure de lancement d'une commande.

Remarque 2.8 :

Certaines extensions du service "`cron(8)`" permettent de modifier l'incrément des plages de nombres. Une plage de nombres suivie de "`/<number>`" indique le pas entre les valeurs à prendre à l'intérieur de l'intervalle. Par exemple, "`0-23/2`", dans le champ "heure" spécifie toutes les deux heures d'une journée (c'est-à-dire

sous Linux.

5. "`cron(8)`" est un processus système s'exécutant en permanence. Dans le jargon Unix, un tel processus est appelé "*démon*" ou "*daemon*".

"0,2,4,6,8,10,12,14,16,18,20,22"). Cette option est aussi autorisée à la suite du caractère "". Par exemple, "*/2" au niveau du champ "heure", sera identique à l'exmple précédent, c'est-à-dire qu'il signifiera "toutes les deux heures".*

Le dernier champ, c'est-à-dire le reste de la ligne, spécifie la commande à exécuter. **Par conséquent, la spécification doit être inscrite sur une seule ligne**. La commande sera exécutée dans le contexte du Bourne Shell (`/bin/sh`) ou celui précisé grâce à la variable d'environnement `"SHELL"`.

Remarque 2.9 :

Le jour de l'exécution d'une commande peut être précisé dans deux champs :

- *le jour du mois (3^e champ),*
- *le jour de la semaine (5^e champ).*

Si ces deux champs sont renseignés, c'est-à-dire qu'ils ne contiennent pas le caractère "", la commande sera exécutée dès que la date courrante correspondra à l'une des deux possibilités. Par exemple, "30 4 1,15 * 5" indiquera que la commande devra être exécutée à 04:30 le 1^{er} et le quinze de chaque mois, mais aussi tous les vendredi.*

2.4.3 Exemple de fichier de description

Exemple 2.8 :

```
# Utiliser /bin/sh pour executer les commandes et non pas celui
# precise dans le fichier /etc/passwd.
SHELL=/bin/sh
# Envoyer un message a "bart" pour rediriger la sortie standard
# toutes les commandes.
MAILTO=bart
#
# Commande a lancer 5 minutes apres minuit chaque jour.
5  0  *  *  *    $HOME/bin/daily.job >> $HOME/tmp/out 2>&1
# Commande a lancer a 14:15 le premier de chaque mois -- la sortie
# standard de la commande arrive dans la boite aux lettres de "bart".
15 14 1  *  *    $HOME/bin/monthly
# Commande a lancer a 22:00 tous les jours de la semaine (hors week-end)
0  22 *  *  1-5  if [ -d /var/adm/acct ]; then chacct; else echo "no acct"; fi
# Autres commandes
23 0-23/2 * * *  echo "execute tous les jours 23 minutes apres 0h, 2h, 4h ..."
5  4  *  *  0    echo "execute a 04:05 chaque dimanche"
```

Troisième partie

Introduction à la programmation Bourne Shell

Chapitre 1

Introduction

1.1 Définition

Un programme shell est un fichier régulier contenant des commandes UNIX ou bien des commandes "intrinsèques" du shell. Les permissions de ce fichier doivent au moins être lecture et exécution. Pour exécuter un programme shell, taper le nom du programme après le prompt à condition que le programme shell se trouve dans un des répertoires spécifiés dans la variable `PATH`. Nous verrons ce point plus en détail à la sectioncall-shell.

1.2 Historique des shells

Le premier Shell développé pour UNIX fut appelé le Bourne Shell, nom provenant du responsable de l'équipe de développement (*Steve Bourne*). Les personnes de l'université de Berkeley en Californie[1] trouvèrent des lacunes au Bourne Shell (d'après eux, trop de données devaient être saisies) et développèrent un nouveau Shell : le "*C Shell*"[2]. Le "*C Shell*" offrit de nouvelles caractéristiques que n'avait pas le Bourne Shell :

- Premièrement, l'utilisateur pouvait rappeler les commandes et les éditer. La procédure n'est pas très aisée[3] mais fonctionnait et avait, au moins, le mérite d'exister contrairement au Bourne Shell.
- En second, le "*C shell*" permettait de définir différents noms pour une commande : les "*aliases*".
- Troisièmement, le "*C shell*" permit la génération automatique de fin de nom de fichier. Il permettait de terminer le nom du fichier en tapant sur la touche "*Escape*" du clavier.

1. University of California Berkeley = UCB.
2. C comme Californie et non pas comme le langage.
3. Les rappels des commandes avec le "*C Shell*" consiste à rappeler la dernière commande commençcant par une chaine de caractères ou bien la n^e commande saisie. Il suffit alors de faire précédent la chaine ou bien le numéro de la commande par le caractère "`!`".

- En fin, l'initialisation et la terminaision du contexte d'une session utilisateur appelle un ou plusieurs fichiers de configuration, en fonction du mode d'appel. Il a donc été distingué :
  - ⋆ l'initialisation d'une session interactive,
  - ⋆ l'initialisation de l'environnement lié à une procédure ou un *sous-shell*[4],
  - ⋆ la fin d'une session interactive.

  . Par conséquent, il est possible de paramètrer son environnement plus finement.

Il présenta donc beaucoup d'autres avantages par rapport au Bourne Shell mais avait deux inconvénients majeurs :

- il n'était pas standard,
- il était incompatible avec les programmes Bourne Shell.

Finalement, David KORN des laboratoires BELL d'AT&T eût le dernier mot. Il écrivit le "*Korn Shell*" (appelé K Shell) avec son équipe de développeurs. Celui-ci possède les meilleures caractéristiques du Bourne Shell, inclut celles du C Shell tout en étant 100% compatible avec le Bourne Shell. Il rajouta aussi un certain nombre de fonctionnalités pour la programmation de procédures de commandes. Enfin, pour *concurencer* le "*Korn Shell*", son équivalent fut développé avec les syntaxes du "*C Shell*" : le "`tcsh`". Ce dernier est un sur-ensemble du "*C Shell*" avec un effort accrue sur la souplesse d'utilisation de l'interpréteur de commande (utilisation plus *conviviale*, fonctionnalités pour les procédures de commandes accrues, etc.).

Toutefois, l'équipe du "*Bourne Shell*" n'en resta pas là. Ses fonctionnalités furent accrues avec le "`bash`" signifiant "*Bourne another Shell*".

En conclusion :

Il existe deux grandes familles pour les syntaxes :

- celle du "*Bourne Shell*", regroupant le Bourne Shell, le Korn Shell et le `bash`,
- celle du "*C Shell*", regroupant le C Shell et le `tcsh`.

Nous ne parlerons dans ce document que du "*Bourne Shell*", première étape vers la création de procédures sous UNIX. La correspondance entre les syntaxes du "*Bourne Shell*" et du "*C Shell*" seront explicitées à l'annexe B.

Remarque 1.1 :

Le Shell POSIX s'appuie sur le "Korn Shell". Donc si vous voulez rester dans la norme POSIX, exigez le "Korn Shell".

4. Un *sous-shell* correspond à un sous-processus associé à un interpréteur de commande (ou *shell*), lancé à partir d'une session interactive.

1.3 Quelques règles et recommandations pour l'écriture des shells-scripts

1.3.1 Règles à observer

Règle 1 : Commentez abondamment et documentez systématiquement vos scripts avec une en-tête comprenant :
- le nom de l'auteur,
- la date,
- une présentation des fonctionnalités.

Règle 2 : Vérifier les paramètres d'appel, et indiquer l'usage si l'appel est incorrect.

Règle 3 : Assortir toute sortie anormale d'un message d'erreur et d'un appel à `"exit"` avec un numéro d'erreur afin de pouvoir être testé dans tout script appelant.

Règle 4 : Adopter une approche modulaire avec usage de :
- fonctions internes,
- scripts externes,
- appel à des scripts existants.

Règle 5 : Utilisez les minuscules pour les noms de variables internes à un script ou une fonction (hors environnement) et les majuscules pour les variables empilées dans ou héritées de l'environnement.

Règle 6 : Si des paramètres d'environnement propres au déroulement ou à l'enchaînement de scripts sont nécessaires, les terminaisons normales et anormales doivent assurer le dépilement de ces paramètres, afin de ne pas polluer inutilement l'environnement utilisateur au fur et à mesure des appels.

Règle 7 : Si un script fait référence à des ressources (fichiers) qu'il ne crée pas lui-même, et/ou s'il atteint un certain volume (supérieur à une page par exemple), rédigez les traces (jalonnement de `"stdout"`) et des messages d'erreurs (`"stderr"`) sur un même fichier de *log*, portant le même nom que le script avec l'extension `".log"`.

Règle 8 : Si des messages et/ou des erreurs doivent être redirigées sur la console[5], alors préfixez les du nom du script qui les génèrent.

Suite page suivante ···

5. La console est un terminal spécial du système chargé d'afficher tous les messages systèmes. En général, ce périphéque est connecté physiquement à la machine sous Unix. Elle peut toutefois être déportée de différentes façons.

Règle 9 : Tout fichier temporaire doit être créé dans un répertoire **réservé et distinct** qui sera identifié par une variable d'environnement ("`D_TEMP`" par exemple). **Ce répertoire sera distinct de "`/tmp`"**. La nécessité de plusieurs répertoires temporaires sera satisfaite par la création de sous-répertoires dans celui-ci. Le "*PID*"[6] (accessible grâce à la variable "`$$`"[7]) du script sera utilisé **systématiquement** comme suffixe des noms de fichiers temporaires générés.
Toutes les ressources temporaires (sous-répertoires et fichiers) seront évacuées de cette zone par destruction (commande "`rm`") ou rangées ailleurs (commande "`mv`") lors des terminaisons.

Règle 10 : Tout script aura comme première ligne, la description du Shell à lancer. On aura donc :

Shell	Première ligne
Bourne Shell	`#!/bin/sh`
C Shell	`#!/bin/csh`
Korn Shell	`#!/bin/ksh`
TC Shell	`#!/bin/tcsh`
Bourne Another Shell	`#!/bin/bash`

La châine de caractère spécifiant le nom de l'exécutable doit être écrite sans espaces et dès la première colonne.

Suite page suivante ···

1.3.2 Recommandations

Le premier souci du développeur de *shell script* doit être la portabilité. En effet, il est dispendieux d'avoir à réadapter des procédures d'exploitation à chaque introduction d'une nouvelle machine. Les scripts développés sur une architecture d'un constructeur sous le même système d'exploitation (UNIX), doivent se comporter de façon identique sur *SunOS*, *Solaris*, HP–UX, *Irix*, *Digital* UNIX, *AIX*, etc. La normalisation POSIX est là pour le permettre.

Les développements ne doivent pas être des opérations où l'on redécouvre et reconstruit la roue. Un usage systématique des utilitaires ad-hoc (de la "*section 1*") est recommandé. Quand cela s'avère nécessaire pour les traitements de données symboliques, on utilisera en priorité les utilitaires de manipulation de fichiers.

On limitera volontairement l'usage des utilitaires "`sed`" et "`awk`" à des séquences simples de traitement, dès l'instant où l'on a recours à des expressions régulières. Décomposez les traitements à effectuer. Exploitez la philosophie et les possibilités d'UNIX :

- chaque commande s'exécute dans un sous process,
- UNIX dispose d'un ensemble de commande très vastes, chacune ayant une fonction bien précise (tâche élémentaire).

6. *PID : Process IDentifier.*
7. cf. section 3.1.

Il est beaucoup plus économique d'utiliser plusieurs commandes qui s'enchaînent (avec les mécanismes des *pipes*) pour réaliser une tâche complexe. Vous y gagnerez en simplicité de raisonnement et de mise au point.

N'oubliez pas que le *C Shell*, le *TC Shell*, le *Bourne Another Shell* (ou `bash`) et le Korn Shell exécutent, à chaque lancement, les fichiers suivants :

Shell	Fichier
C Shell	`~/.cshrc`
TC Shell	`~/.tcshrc`
Bourne Another Shell	`~/.bashrc`
Korn Shell	`~/.kshrc`

Par conséquent, à chaque lancement d'un script *C Shell*, *TC Shell*, *Bourne Another Shell* (ou `bash`) ou Korn Shell, le fichier d'initialisation est exécuté par défaut.

Chapitre 2

Les arguments des programmes shell

Tout programme shell peut être invoqué depuis la ligne de commande. Les arguments de la ligne de commande sont référencés par le programme shell lui-même en fonction de leur position relative dans la ligne de commande. Ces arguments sont appelés paramètres ou variables positionnels. Ils prennent la valeur correspondante à l'argument de la ligne de commande. Les paramètres positionnels peuvent être utilisés dans les programmes shell de la même manière que les variables shell, c'est-à-dire que pour les référencer, il suffit d'utiliser le symbole `"$"`. On ne référencer que jusqu'à neuf arguments de la ligne de commande (`$1` à `$9`). Nous verrons qu'il est possible de récupérer tous les arguments de la ligne de commande.

L'argument zéro de la ligne de commande (variable `"$0"` correspond au nom de la commande.

En résumé :

Ligne de commande :
`$ nom_prog arg1 arg2 arg3 arg4 ...`
Dans le programme Shell :

		Équivalences	
Commande shell	Sortie	OpenVMS	Langage C
`echo $0`	`nom_prog`	*pas d'équivalent*	`argv[0]`
`echo $1`	`arg1`	`P1`	`argv[1]`
`echo $2`	`arg2`	`P2`	`argv[2]`
...			
`echo $8`	`arg8`	`P8`	`argv[8]`
`echo $9`	`arg9`	*pas d'équivalent*	`argv[9]`

Il est possible de référencer la totalité des arguments à l'intérieur d'un script shell. Pour cela, la variable `"$#"` et la commande `"shift"` seront utilisées. Reportez-vous aux sections 3.1 et 4.1.

Chapitre 3

Les variables

3.1 Les variables spéciales

La variable "`#`" renvoie le nombre d'arguments passés à la procédure.

La variable "`*`" renvoie la liste de tous les arguments passés à la procédure même s'ils sont plus de neuf.

Le fait de lancer un programme shell crée un process dans lequel un shell s'exécute pour interpréter les commandes du Shell Script. La variable "`$`" renvoie le PID du nouveau shell qui est lancé.

La variable "`!`" renvoie le numéro de process (PID) de la dernière commande lancée en arrière plan grâce au caractère de terminaison "`&`". Cette variable peut être utile pour synchroniser un script grâce à la commande interne "`wait`".

La variable "`-`" renvoie la liste des options positionnées pour le shell courant grâce à la commande interne "`set`" (cf. "`sh(1)`").

La variable "`?`" contient le code de retour de la dernière commande qui a été exécutée. De façon générale, une valeur nulle indique *VRAI*, un retour non nul indique une erreur.

En résumé :

Variable	Valeur retournée
`#`	Nombre d'arguments.
`*`	Liste des arguments du script.
`$`	Numéro du process shell dans lequel s'exécute le script.
`!`	Numéro du process de la dernière commande lancée en arrière plan (grâce au caractère "`&`")
`-`	Liste des options du shell positionnées avec la commande `set`.
`?`	Code de retour de la dernière commande exécutée.

Exemple 3.1 :

```
% nom_prog arg1 arg2 arg3
```

`echo $#` affiche "3" dans le programme shell.

`echo $*` affiche la chaine "arg1␣arg2␣arg3" dans le programme shell, "␣" représentant SPACE .

`echo $$` affiche le numéro du processus dans le programme shell s'exécute.

3.2 Manipulation sur les variables

Il est possible de combiner des appels à une variable avec une expression qui sera évaluée par le shell et, ainsi, renverra une valeur suivant le contexte.

Expression	Résultat de l'évaluation
`${variable-chaine}`	Si la variable est initialisée, l'expression renvoie son contenu. Dans le cas contraire, elle renvoie la chaîne de caractères spécifiée.
`${variable:-chaine}`	Si la variable est initialisée et est non vide (différente de la chaîne vide «"), l'expression renvoie son contenu. Dans le cas contraire, elle renvoie la chaîne de caractères spécifiée.
`${variable=chaine}`	Si la variable est initialisée, l'expression renvoie son contenu. Dans le cas contraire, la variable est initialisée avec la chaîne spécifiée dans l'expression. La valeur finale retournée au shell est le nouveau contenu de la variable (donc la chaîne spécifiée dans l'expression).
`${variable:=chaine}`	Si la variable est initialisée et est non vide (différente de la chaîne vide «"), l'expression renvoie son contenu. Dans le cas contraire, la variable est initialisée avec la chaîne spécifiée dans l'expression. La valeur finale retournée au shell est le nouveau contenu de la variable (donc la chaîne spécifiée dans l'expression).
`${variable?chaine}`	Si la variable est initialisée, l'expression renvoie son contenu. Dans le cas contraire, le shell affiche un message d'erreur dont la forme est : "`variable: chaîne`".
`${variable:?chaine}`	Si la variable est initialisée et est non vide (différente de la chaîne vide «"), l'expression renvoie son contenu. Dans le cas contraire, le shell affiche un message d'erreur dont la forme est : "`variable: chaîne`".
`${variable+chaine}`	Si la variable est initialisée, l'expression renvoie la valeur de la chaîne. Dans le cas contraire, on renvoie une chaîne nulle.

Suite page suivante ···

	Suite de la page précédente.
Expression	Résultat de l'évaluation
`${variable:+chaine}`	Si la variable est initialisée et est non vide (différente de la chaîne vide «"), l'expression renvoie la valeur de la chaîne. Dans le cas contraire, on renvoie une chaîne nulle.

Exemple 3.2 :

```
echo ${MY_VAR-"non defini"}
echo ${HOME_ADM=/home/adm}
echo ${MY_VAR?"Variable non initialis{\'e}e dans `basename $0`"}
echo ${MY_VAR+"Ca roule, ma poule"}
```

Remarque 3.1 :

Attention, *le contenu de la variable "`*`" est le résultat de l'évaulation de la ligne de commande. Par conséquent, les espaces ou tabulations supplémentaires, délimitant les arguments du script seront éliminés.*

Par conséquent, le programme appelé de la façon suivante sur la ligne de commandes :

`%␣nom_prog␣arg1␣␣arg2␣␣␣␣arg3␣␣␣␣␣␣arg4`

contiendra la valeur suivante dans la variable "`*`" :

`arg1␣arg2␣arg3␣arg4`

et non pas *:*

`arg1␣␣arg2␣␣␣␣arg3␣␣␣␣␣␣arg4`

Chapitre 4

Commandes évoluées pour les scripts

4.1 La commande "shift"

Syntaxe :

```
shift [n]
```

La commande "shift" permet de gérer les arguments passés à un script au niveau de la ligne de commandes. Les actions effectuées sont :

- décalage de toutes les chaînes de la variable "*" de "n" positions vers la gauche,
- décrémentation de la variable "#" de "n".

Par défaut, "n" est positionné à 1.

La commande "shift" est une commande **interne du Shell**.

4.2 La commande "read"

Syntaxe :

```
read variable [variable...]
```

La commande "read" est utilisée pour lire les informations sur l'entrée standard. S'il y a plus de variables dans la commande "read" que de valeurs réellement saisies[1], les variables les plus à droites sont assignées à "NULL" (chaîne vide).

Si l'utilisateur saisit plus de mots qu'il n'y a de variables, toutes les données de droite sont affectées à la dernière variable de la liste. Une fois positionnées, celles-ci sont accédées comme les autres.

1. N'oubliez pas l'espace ou " SPACE " est séparateur au niveau de la ligne de commandes.

La commande "`read`" est une commande **interne du Shell**.

Exemple 4.1 :

On suppose qu'un script contient la commande suivante :

```
read var1 var2 var3
```

***Premier cas* :**

L'utilisateur saisit la chaîne suivante :

`chaine1␣chaine2␣chaine3`

Dans ce cas, les variables contiendront :

Variable	Valeur
var1	chaine1
var2	chaine2
var3	chaine3

***Deuxième cas* :**

L'utilisateur saisit la chaîne suivante :

`chaine1␣chaine2`

Dans ce cas, les variables contiendront :

Variable	Valeur
var1	chaine1
var2	chaine2
var3	NULL

***Troisième cas* :**

L'utilisateur saisit la chaîne suivante :

`chaine1␣chaine2␣chaine3␣chaine4␣chaine␣5`

Dans ce cas, les variables contiendront :

Variable	Valeur
var1	chaine1
var2	chaine2
var3	chaine3␣chaine4␣chaine␣5

4.3 La commande "expr"

Syntaxe :

```
expr expression
```

Les arguments de la commande "`expr`" sont considérés comme une expression. La commande "`expr`" évalue ses arguments et écrit le résultat sur la sortie standard. "`expr`", comme toutes les commandes UNIX, doit avoir ses arguments **séparés par des espaces**. La première utilisation de "`expr`" concerne les opérations arithmétiques simples. Les opérateurs `+`, `-`, `*` et `/` correspondent respectivement à l'addition, à la soustraction, à la multiplication et à la division.

La seconde utilisation de la commande "`expr`" concerne la comparaison et la gestion des chaînes de caractères grâce à l'opérateur "`:`". Celui-ci permet de compter combien de fois apparaît, dans l'expression placée en premier membre,

l'expression régulière spécifiée en second membre. Pour plus de renseignements, reportez vous au chapitre 8.

Il est possible d'effectuer des regroupements grâce à des parenthèses, comme dans toute opération arithmétique.

Attention :

Ne pas oublier que le processus d'évaluation du shell effectue ses opérations avant de passer le contrôle à la commande "expr". Par conséquent, les caractères "*", "(" et ")" risquent d'être évalués par le shell et non pas passé en argument à la commande "expr". Il faudra donc les faire précéder du caractère "\" (cf. sections 1.5 et 5).

Exemple 4.2 :

Exemples valides :

```
% X=3; Y=5
% Z=‘expr $X + 4 ‘
% echo $Z
7
% Z=‘expr \( $Z + $X \) \* $Y‘
% echo $Z
50
% X=abcdef
% Z=‘expr $X : ’.*’ ‘
% echo $Z
6
% Z=‘expr \( $X : ’.*’ \) + $Y‘
% echo $Z
11
```

Exemples non valides :

`Z=‘expr␣$X+4‘`
Tout d'abord, le shell cherche une variable dont le nom est "X+4". En supposant qu'elle n'existe pas, la commande "expr" ne reçoit qu'un seul argument :l'expression "X+4", ce qui n'a aucun sens.

`Z=‘expr␣$X␣*␣$Y‘`
Les variables "X" et "Y" existent bien et contiennent des valeurs numériques. Par conséquent, le résultat de l'évaluation de "$X" et "$Y" est valide. Par contre, "*" est un métacaractère valide du shell (cf. section 1.5). Il doit donc être remplacé par les fichiers correspondants, c'est-à-dire tous les fichiers du répertoire courant. L'expression devient alors :
`Z=‘expr␣3␣fic1␣fic2␣fic3␣fic4␣fic5␣fic6␣4‘`
en supposant que les variables "X" et "Y" contiennent respectivement les valeurs "3 " et "4" et que le répertoire courant contient les fichiers "fic1", "fic2", "fic3", "fic4", "fic5" et "fic6".

`Z=‘expr␣(␣$X␣\*␣$Y␣)␣+␣$Z‘`

Ici, les caractères "`(`" et "`)`" sont pris en compte par le mécanisme d'évaluation du shell : ils indiquent une expression à exécuter dans un processus séparé (cf. section 5). Il apparaît que, ne serait-ce que pour la syntaxe, cette expression est invalide pour le shell. C'est donc le shell qui renverra une erreur et non pas la commande "`expr`".

Chapitre 5

Les listes de commandes et les fonctions

5.1 Les listes de commandes

Une liste de commandes est une séquence d'un ou plusieurs pipes ou commandes séparés par l'un des caractères suivants :

Caractère	Signification
`;`	Le shell attend que l'expression précédant le caractère `";"` soit terminé pour passer à la suivante.
`&`	Le shell exécute l'expression précédent en arrière plan et passe à la suivante.
`&&`	Le shell ne passe que l'expression suivant seulement si le premier a renvoyer une valeur nulle comme code de retour (code `"TRUE"`).
`||`	Le shell ne passe qu'au *pipe* suivant seulement si le premier a renvoyé une valeur non nulle comme code de retour (code `"FALSE"`).

Exemple 5.1 :

```
ls -lR | wc -l; more /usr/lib/sendmail/sendmail.cf
ls /usr/bin; cat /etc/passwd
find / -print >big.file & ls /usr/bin
[ -f /etc/passwd] && cat /etc/passwd
```

Chaque liste de commandes peut être regroupée de deux façons :

Syntaxe	Signification
`(liste)`	Exécute la liste dans un sous shell sans perturber le contexte du shell courant.
`{liste␣;}`	Exécute la liste dans le shell courant. Aucun sous shell n'est créé.

Exemple 5.2 :

L'exemple ci-après permet de recopier le contenu d'un répertoire vers un autre. Pour cela, on utilise la commande "`tar(1)`". L'option "`-c`" permet de créer une archive, l'option "`-x`" permet d'extraire les informations. Lorsque l'option "`-f`" est précisiée sur la ligne de commande, le premier argument de la commande "`tar`" est le nom du fichier où l'archive doit être écrite ou lue. Le fichier spécifié ici est "`-`" indiquant l'entrée ou la sortie standart. Lors de la création d'une archive, la commande "`tar`" admet au moins un argument : le fichier à sauvegarder ou bien le nom d'un répertoire contenant le point de l'arborescence à sauvegarder.

```
cd /home/users/schmoll; tar -cvf - . | (cd /tmp/trf; tar -xvf -)
```

L'expression précédente peut se décomposer en trois étapes :

1. Changement de répertoire.
2. Création d'un processus dans lequel la commande "`tar`" doit s'exécuter. Celle-ci crée une archive contenant l'arborescence du répertoire courant (le répertoire "`/home/users/schmoll`") et l'envoie sur sa sortie standard.
3. Création d'un nouveau processus dont l'entrée standard est redirigée sur la sortie standard de la commande "`tar`" précédente. Ce sous processus se déplace dans l'arborescence (répertoire "`/tmp/trf`") et crée un nouveau sous processus pour exécuter une autre commande "`tar`" lisant le contenu de l'archive à extraire sur l'entrée standard.

FIG. 5.1 – *Enchaînement de commandes et modèle d'exécution*

En conclusion, le premier membre de l'expression (à gauche du symbole "`|`" prend le contenu du répertoire "`/home/users/schmoll`" et l'envoie sur sa sortie standard via le format d'archive "`tar`". Le second membre change de répertoire et restaure le contenu de le répertoire courant. Cette expression se résume donc au schéma 5.1.

5.2 Les fonctions

5.2.1 Déclaration et utilisation des fonctions

Syntaxe :

```
nom_fonction ()
```

```
{
    liste
}
```

Une fonction doit être décrite avant tout appel. Elle gère ses paramètres de la même façon qu'un shell script (variables "0" à "9", commande "shift", etc.). Elle n'est connu seulement que **du process dans lequel elle a été définie**.

L'utilisation des fonctions en Shell diffère de celle que l'on peut en faire avec les langages évolués. Elles servent à effectuer un traitement élémentaire à l'intérieur d'un script (connu de lui seul). Si le traitement doit être commun à un ensemble de procédure, il est préférable d'utiliser un script externe.

Une fonction en Shell diffère de celles que l'on peut trouver dans un langage évolué comme le C ou le Fortran par le fait qu'elle ne sait renvoyer qu'un statut d'exécution. Elle ne permet donc pas de remonter de valeurs au programme appelant, voire-même modifier des arguments qui lui seraient passés.

Remarque 5.1 :

Pour plus de clarté, il est préférable de faire précédé le nom de chaque fonction par le caractère "_" (underscore).

Une fonction s'exécute dans le shell courant, c'est-à-dire dans le même processus. Par conséquent, tout appel à la commande "exit" entraîne la fin du shell script.

Exemple 5.3 :

La fonction ci-après va nous permettre de demander à l'utilisateur de répondre par "O" pour *oui* ou "N" pour *non*. A chaque fois que le script aura besoin de poser ce type de question, cette fonction sera appelée. Elle va donc admettre les agurments suivants :

- la question,
- la réponse par défaut.

Sachant qu'une fonction ne peut renvoyer qu'un statut d'exécution, il faudra utiliser d'autres mécanismes si l'on veut retourner la réponse effectivement saisie : *le mécanisme des redirections d'entrées/sorties.* Dans le cas qui nous interresse, tous les messages de la fonction seront redirigés sur la sortie d'erreurs standard et la réponse saisie par l'utilisateur, sur la sortie standard.
Nous aurons donc :

```
_ask ()
{
    # Arg1 = message
    # Arg2 = reponse par defaut

    if [ $# -ne 2 ]; then
        echo "Missing arguments in function \"ask\"." >&2
        exit
    fi
    case "$2" in
        o|O)
```

```
            message="$1 ([o]/n):"
            ;;
        n|N)
            message="$1 (o/[n]):"
            ;;
        *)
            echo "Invalid default answer in function \"ask\"." >&2
            exit
            ;;
    esac
    while
        echo "$message \c" >&2
        read answer
        [ "$answer" = "" ] && answer=$2
        answer=`echo $answer | tr '[A-Z]' '[a-z]'`
        [ "$answer" != "o" -a "$answer" != "n" ]
    do
        echo "Reponse invalide. Recommencez." >&2
    done
    echo $answer
}

rep=`_ask "Voulez-vous  vraiment poursuivre ce programme" "o"`

if [ "$rep" = "n" ]; then
    exit
fi

echo "Bonjour."
echo "Voici un programme Shell."

rep=`_ask "Est-ce que l'on continue" "o"`

...
```

5.2.2 Commande "return"

Syntaxe :

```
return [n]
```

La commande interne "**return**" permet de renvoyer un code de fin d'exécution au programme appelant dans la variable "?".

Si aucune valeur n'est spécifiée après la commande "**return**", la valeur par défaut sera le statut de la dernière commande exécutée à l'intérieur de la fonction.

Chapitre 6

Les commentaires et les techniques d'exécution

6.1 Les commentaires

Syntaxe :

```
# <ligne de commentaire>
commande # commentaire
```

Le signe "#" marque le début d'une zone de commentaire du script. Elle se termine à la fin de la ligne. Tout ce qui suit ce caractère est donc ignoré par le shell.

6.2 Interprétation spéciale du signe "#" sur la première ligne d'un shell script

Si l'on satisfait les conditions suivantes :

- le signe "#" se trouve en première ligne du fichier shell script **et** sur la première colonne,
- il est suivi du caractère "!",
- la séquence "#!" est suivie du nom de l'exécutable d'un shell (chemin absolu),

alors, cette ligne permettra d'avoir la description du shell à lancer au moment de l'exécution.

Le processus mis en place est le suivant lorsqu'une telle ligne est trouvée :

- lancement du shell spécifié sur la première ligne,
- lecture du fichier et évaluation/exécution des commandes par le shell que l'on vient le lancer.

Cette démarche est obligatoire pour tout shell script (cf. règle 10 de la section 1.3.1).

Cette technique est donc utilisé pour l'ensemble des langages interprétés sous UNIX comme :

- les différentes variantes du shell (Bourne Shell, C-Shell, Korn Shell, etc.),
- des utilitaires admettant des fichiers de requêtes comme "`egrep`" (cf. section 9.3), "`sed`" (cf. section 10) et "`awk`" (cf. section 11),
- des langages interprétés comme "`perl`"[5, 6, 7].

Nous pourrons donc avoir les cas de figure suivants :

Syntaxe	Signification
`#!/bin/sh`	Tout ce qui va suivre obéit à la syntaxe Bourne Shell. Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/bin/sh`".
`#!/bin/csh`	Tout ce qui va suivre obéit à la syntaxe C Shell. Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/bin/csh`".
`#!/bin/ksh`	Tout ce qui va suivre obéit à la syntaxe Korn Shell. Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/bin/ksh`".
`#!/bin/egrep`	Tout ce qui va suivre obéit à la syntaxe de la commande "/bin/egrep". Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/bin/egrep`".
`#!/bin/sed`	Tout ce qui va suivre obéit à la syntaxe de la commande "`/bin/sed`". Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/bin/sh`".
`#!/bin/awk`	Tout ce qui va suivre obéit à la syntaxe de l'utilitaire "`awk`". Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/bin/awk`"
`#!/usr/bin/perl`	Tout ce qui va suivre obéit à la syntaxe de Perl. Le processus à lancer pour exécuter et évaluer les instructions suivantes exécutera le programme "`/usr/bin/perl`"
etc.	

6.2.1 Les appels d'un shell script au niveau de la ligne de commandes

La notion exposée ici s'applique à tout type de programmes interprétés sous UNIX. Nous ne traiterons ici que l'exemple du Bourne Shell ("`/bin/sh`").

Syntaxes :

Appel au niveau de la ligne de commande	Descriptions – Restrictions
Suite de la page précédente.	

Suite page suivante ···

Appel au niveau de la ligne de commande	Descriptions – Restrictions
`shell_script [arguments-programme]`	"`shell_script`" représente le nom du fichier. Celui-ci doit être accessible en lecture **et en exécution**.
`/bin/sh shell_cript [arguments-programme]`	"`shell_script`" représente le nom du fichier. Celui-ci doit être accessible en lecture **uniquement**.
`/bin/sh -vx shell_script [arguments-programme]`	"`shell_script`" représente le nom du fichier. Celui-ci doit être accessible en lecture. Chaque ligne du fichier est interprétée et affichée avant son exécution sur la sortie d'erreurs standard.

Remarque 6.1 :

L'option "`-vx`" est propre au Bourne Shell, C Shell et Korn Shell. Pour d'autres outils comme "`sed`", "`awk`" ou "`perl`", reportez-vous à la page du manuel associée.

Chapitre 7

Les tests et les boucles

7.1 Les tests

7.1.1 La commande « `test` »

La commande « `test` » évalue l'expression spécifiée en argument et positionne la valeur de retour :

- à 0 si l'expression est vraie,
- à une valeur non nulle si l'expression est fausse.

Syntaxe :

`[`␣*expression*␣`]`
ou
`test`␣*expression*

Les espaces doivent être respectés avant et après les « `[` », « `]` ».

Remarque 7.1 :

La commande « `test` » est une commande externe au Shell.

La commande « `test` » est utilisée pour évaluer si une expression est vraie ou fausse. Elle sera alors utilisée par d'autres commandes du shell permettant d'effectuer :

- des branchements conditionnels (« `if` »),
- des boucles,
- etc.

7.1.2 Tests sur les fichiers

Syntaxe :

```
[␣option␣fichier␣]
ou
test␣option␣fichier
```

Les options usuelles sont :

Options	Actions
`-f`	Vrai si le fichier est de type ordinaire (type « - »).
`-r`	Vrai si le fichier existe et est accessible en lecture.
`-w`	Vrai si le fichier existe et est accessible en écriture.
`-x`	Vrai si le fichier existe et est accessible en exécution.
`-d`	Vrai si le fichier existe et est un répertoire.
`-s`	Vrai si le fichier existe et est de taille non nulle.
`-c`	Vrai si le fichier existe et est un fichier périphérique de type « *raw device* ».
`-b`	Vrai si le fichier existe et est un fichier périphérique de type « *block device* ».

7.1.3 Tests sur les chaînes de caractères

Syntaxe :

Est Égal	N'est pas Égal
`[␣chaine1␣=␣chaine2␣]`	`[␣chaine1␣!=␣chaine2␣]`
ou	
Est Égal	N'est pas Égal
`test␣chaine1␣=␣chaine2`	`test␣chaine1␣!=␣chaine2`

Quand un test est effectué sur une variable shell, il est judicieux de penser que celle-ci peut très bien ne rien contenir. Par exemple, considérons le test suivant :

```
[ $XX = oui ]
```

Si la variable « `XX` » n'est pas initialisée, c'est à dire si « `XX` » est nulle, le shell réalisera la substitution de variable et tentera d'exécuter le test suivant :

```
[  = oui ]
```

qui est incorrect au niveau syntaxe et promet un message d'erreur. Le moyen simple pour pallier à ce problème est de préciser le nom de variable entre double quotes (« " ») ce qui assure l'affectation de la variable même si celle-ci est `NULL`. Soit :

```
[ "$XX" = oui ]
```

ce qui donne après évaluation du shell :

```
[ "" = oui ]
```

Remarquez également que si la variable est susceptible de contenir des caractères blancs, il est intéressant d'entourer celle-ci de double quotes.

7.1.4 Les test numériques

Syntaxe :

`test␣nombre␣relation␣nombre`
ou
`[␣nombre␣relation␣nombre␣]`

Les relations supportées sont :

Relation	Signification
`-lt`	Strictement inférieur à
`-le`	Inférieur ou égal à
`-gt`	Strictement supérieur à
`-ge`	Supérieur ou égal à
`-eq`	Égal à
`-ne`	Différent de

7.1.5 Autres opérations

Opération	Signification
`-o`	OU logique
`-a`	ET logique
`!`	Négation logique
`\(␣\)`	Regroupement (les parenthèses doivent être précédées de back slash « \ »)

7.2 La construction « if »

Syntaxe :

Syntaxe	*Exécution*

`if`
liste A
`then`
	« *liste A* » est exécutée.

liste B	Si la valeur de retour de la dernière commande de « *liste A* » est nulle (vrai), le shell exécute « *liste B* », puis saute à la première instruction après « `fi` ».

Suite page suivante ···

Suite de la page précédente ···

Syntaxe	*Exécution*
`elif`	Optionnel.
liste C	Si le code retour de la dernière commande de « *liste A* » est non nul (faux), on exécute « *liste C* ».
`then`	
liste D	Si le code retour de la dernière commande de « *liste C* » est nul, le shell exécute « *liste D* », puis saute à la première instruction après le « `fi` ».
etc.	
`else`	Optionnel.
liste E	Si le code retour de la dernière commande de « *liste C* » est non nul, le shell exécute « *liste E* » puis saute à la première instruction après le « `fi` ».
`fi`	

Une liste de commandes (*liste A*, *liste B*, etc.) est une séquence de commandes shell séparées par des points virgule (« `;` ») ou bien par un retour chariot (RETURN).

7.3 La construction « `case` »

La construction « `case` » est utilisée pour des branchements multiples.

Syntaxe :

Syntaxe	*Exécution*
`case` *mot* `in`	
choix1`)`	Le mot est comparé à « *choix1* ».
liste A	S'il correspond, « *liste A* » est exécutée puis le shell se branche à la première instruction suivant le « `esac` ».
`;;`	
choix2`)`	
liste B	Si « *choix1* » ne correspond pas, le mot est comparé à « *choix2* » et ainsi de suite. S'il n'y a aucune correspondance, on continue.
`;;`	
`esac`	

Les choix sont construits à partir des caractères du shell de génération de nom de fichiers. Il est également possible d'utiliser le caractère « | » signifiant

« *OU logique* ».

7.4 La boucle « `while` »

Syntaxe :

```
while
   liste A
do
   liste B
done
```

Exécution :

étape 1 : « *liste A* » est exécutée.

étape 2 : Si la valeur de retour de la dernière commande de « *liste A* » est nulle (vrai), « *"liste B* » est exécutée et retourne à l'étape 1.

étape 3 : Si la valeur de retour de la dernière commande de « *liste A* » est non nulle (faux), le shell se branche à la première instruction suivant « `done` ».

7.5 La boucle « `until` »

Syntaxe :

```
until
   liste A
do
   liste B
done
```

Exécution :

étape 1 : « *liste A* » est exécutée.

étape 2 : Si la valeur de retour de la dernière commande de « *liste A* » est non nul (faux), « *"liste B* » est exécutée et retourne à l'étape 1.

étape 3 : Si la valeur de retour de la dernière commande de « *liste A* » est nulle (vrai), le shell se branche à la première instruction suivant « `done` ».

7.6 La boucle « `for` »

Syntaxe :

```
for var in liste
do
   liste A
done
```

« *var* » est une variable du shell et « *liste* » est une liste de chaînes de caractères délimitées par des espaces ou des tabulations. **Le nombre de chaînes de caractères de la liste détermine le nombre d'occurrence de la boucle**.

Exécution :

étape 1 : « *var* » est positionné à la valeur de la première chaîne de caractères de la liste.

étape 2 : « *liste A* » est exécutée.

étape 3 : « *var* » est positionnée à la valeur de la seconde chaîne de caractères de la liste, puis reprise à l'étape 2.

étape 4 : Répétition jusqu'à ce que toutes les chaînes de caractères aient été utilisées.

Remarque 7.2 :

Toute chaîne de caractères comprises entre les caractères « " » et « ' » comptent pour une itération pour la boucle « for ». Par conséquent :

```
for␣var␣in␣"Ceci␣est␣une␣cha{\^\i}ne"      compte 1 itération.
do
...
done
```

```
for␣var␣in␣Ceci␣est␣une␣cha{\^\i}ne        compte 4 itération.
do
...
done
```

7.7 Les commandes « `break` », « `continue` » et « `exit` »

`break` [*n*]	Provoque la fin de la n^e boucle pour les boucles imbriquées.
`continue` [*n*]	Reprise à la n^e boucle pour les boucles imbriquées.
`exit` [*n*]	Stoppe l'exécution du programme shell et positionne le code retour à la valeur « *n* ».

Pour l'une quelconque des trois boucles (« `while` », « `until` », et « `for` »), vous pouvez utiliser les commandes « `break` » et « `continue` » pour en modifier le déroulement.

La commande « `break` » provoquera la fin de la boucle et le saut à la première commande suivant « `done` ».

La commande « `continue` » est légèrement différente. Les commandes suivantes du corps de la boucle seront ignorées et l'exécution reprendra au début de la liste d'initialisation dans le cas des boucles « `while` » et « `until` ». Utilisée dans une boucle « `for` », les commandes suivantes du corps de la boucle seront ignorées et la variable sera positionnée à la valeur suivante de la liste.

Par défaut, le nombre de boucles considérées par les commandes « `break` »

et « `continue` » est 1.

La commande « `exit` » provoquera **l'arrêt définitif du programme shell** et le positionnement du code de retour de ce programme shell à la valeur de l'argument s'il est spécifié. Si aucun n'est spécifié, le code de retour du programme shell sera positionné à la valeur de retour de la dernière commande exécutée juste avant « `exit` ».

Remarque 7.3 :

Comme nous l'avons vu, la commande « `exit` » provoque l'arrêt définitif du programme shell. Par extension, elle provoque l'arrêt de l'interpréteur de commandes courant. Elle permet donc de se déconnecter (cf. section 1.2).

7.8 Signaux et traps

7.8.1 Définition des signaux et des traps

7.8.1.1 Les signaux

Certains événement génèrent des signaux qui sont envoyés au processus.

Par exemple :

- lors d'une déconnexion, le signal 1 est envoyé aux processes lancés en arrière plan,
- la pression de CTRL-C envoie un signal 2 aux processes en avant plan,
- la commande « `kill` » envoie, par défaut, le signal 15 aux processus spécifiés en argument (cf. section 2.3.1).

7.8.1.2 Les *traps*

Un *trap* est un « piège » pour « attraper » un signal. Il est alors possible de lui associer l'exécution de quelques actions.

Un *trap* est une façon d'interrompre le déroulement normal d'un processus pour répondre à un signal en exécutant une action prévue à cet effet. Cette action est appelée « *action de gestion d'interruption* »[1]. Elle ne sera exécutée que dans le cas où une interruption spécifique surviendrait. Il est possible d'avoir plusieurs actions distinctes en réponse à plusieurs signaux distincts.

Le signal 9 correspond à l'interruption absolue. Il ne pourra pas être ignoré ni *trappé* comme les autres.

1. Interrupt Service Routine.

7.8.1.3 La commande « trap »

Syntaxe :

```
trap␣'commande'␣signo␣[signo␣...]
```

« `trap` » permettra d'exécuter des commandes si le signal « *signo* » survient.

La plupart du temps, la réception d'un signal quelconque provoquera l'arrêt du process qui le reçoit. La commande « `trap` » pourra être utilisée dans des programmes shell pour piéger des signaux avant qu'ils n'interrompent le process généré par l'exécution du programme. Ceci permet au programmeur de prévoir une réponse pour certains signaux.

Les commandes « `trap` » sont, en général, placées au début des programmes shell, mais elles peuvent être placées n'importe où de façon à contrôler au mieux le déroulement d'un processus. À la lecture des commandes « `trap` », le shell positionne des pièges à signaux qui seront activés lors de la venue des dits signaux.

Les signaux à piéger spécifiés dans la commande « `trap` » le sont par leur numéro (*signo*). Pour ignorer des signaux, il suffit de taper la commande (ou de l'insérer dans un shell script) :

```
$␣trap␣''␣signo␣[signo␣...]
```

« `trap` » ne définit les pièges que pour le process courant et tous les sous processus. « `trap` » est une commande interne au shell.

Chapitre 8

Les expressions régulières

8.1 Introduction - Définition

Une expression régulière est une séquence de symboles qui répond à une syntaxe précise permettant de d'identifier des chaînes de caractères existantes mais ne permet jamais d'en créer. Dans une expression régulière, on peut trouver tous les caractères sauf le caractère de saut de ligne (fin de ligne, `\n`, `ASCII`(10), *line feed*).

Les expressions régulières sont utilisées dans beaucoup de commandes UNIX ("`sed`", "`awk`", "`grep`", etc.). Elles sont aussi à la base d'outils mutli plateforme pour le WEB comme "`perl`"[5, 6, 7]. Leur syntaxe et leur lisibilité ne sont pas des plus claires mais elles constituent, en grande partie, la puissance des Shells.

8.2 Spécification de caractères particuliers

Symbole	Signification
`x`	un caractère précis (ici le caractère "`x`").
`.`	un caractère quelconque sauf le caractère *line-feed*.
`^`	début de ligne.
`$`	fin de ligne.
`\`	inhibition de l'interprétation d'un caractère spécial.
`[liste]`	un caractère quelconque dans la liste.
`[^liste]`	un caractère quelconque absent de la liste.

Le caractère "`\`" permet de désigner les caractères particuliers, utilisés dans la définition d'une expression régulière, qui ne doivent pas être interprétés.

L'expression régulière "`[liste]`" désigne n'importe quel caractère de la liste, mais un et un seul seulement. L'expression régulière "`[^liste]`" désigne n'importe quel caractère sauf les caractères contenus dans la liste. La liste de ces

deux expressions régulières peut avoir deux formats :

- une suite de caractères à prendre en compte. Par exemple: "[abcABC]" désigne l'un de ces six caractères. De même "[␣abcABC]" désigne les même caractères que précédemment, **l'espace en plus**. Ce type d'expression régulière s'appelle *classe de caractères* (*character class*).
- une séquence de caractères en ne précisant que les bornes séparées par le caractère "-". Par exemple: «[r-v]" désigne tous les caractères compris entre "r" et "v"). Ce type d'expression régulière s'appelle *character range* (intervalle de caractères).

8.3 Exemples d'application

Expression régulière	Signification
abc	désigne la chaîne "abc".
\\ * \$	désigne le caractère qui suit "\" comme la constante caractère et non comme un caractère à interpréter (ici "\", "*" et "$").
c$	désigne le caractère "c" en fin de ligne.
c\$	désigne la chaîne "c$".
^abc	désigne la chaîne "abc" en début de ligne.
abc$	désigne la chaîne "abc" en fin de ligne.
^abc$	désigne la chaîne "abc". C'est la seule chaine de la ligne.
^abc.	désigne la chaîne "abc" en début de ligne suivi de n'importe quel caractère.
[Aa]	désigne une chaîne contenant le caractère "A" ou "a".
x[Aa][Bb]	désigne une chaîne contenant le caractère "x" suivi de "A" ou "a" puis de "B" ou "b".
[^Aa]	désigne une chaîne ne contenant ni caractère "A" ni le caractère "a".
[r-v]	désigne tous les caractères compris entre "r" et "v" ("r","s","t", "u", "v").
[^r-v]	désigne tous les caractères sauf "r", "s", "t", "u" et "v".

8.4 Recherche de caractères suivant un nombre d'occurrences

Symboles : :

Symbole	Signification
*	recherche du caractère pointé de 0 à n fois.
+	recherche du caractère pointé de 1 à n fois.
?	recherche du caractère pointé 0 ou 1 fois.
expr1|expr2	désigne une chaîne correspondant à l'une ou l'autre des deux expressions régulières.

Suite page suivante · · ·

	Suite de la page précédente ···
Symbole	Signification
(···)	partie d'expression régulière.

Une fois le caractère spécifié, on peut sélectionner le nombre d'occurrences d'apparition dans l'expression régulière.

- Le symbole "`*`" désigne que le caractère pointé peut apparaître de 0 à n fois dans la chaîne.
- Le symbole "`+`" désigne que le caractère pointé doit apparaître au moins une fois dans la chaîne.
- Le symbole "`?`" désigne que le caractère pointé doit apparaître au plus une fois dans la chaîne.
- Le symbole "`|`" permet de combiner plusieurs expressions régulières. La condition est satisfaite si au moins l'une des deux expressions est satisfaite.

Les parenthèses permettent de regrouper des expressions régulières, afin de fixer des priorités.

Exemple 8.1 :

Expression régulière	Signification
`56*`	désigne une chaîne contenant un "`5`" suivi de 0 ou plusieurs "`6`".
`56+`	désigne une chaîne contenant un "`5`" suivi d'au moins un "`5`".
`56?7`	désigne une chaîne contenant un "`5`" suivi d'un "`6`" ou pas puis d'un "`7`".
`123|321`	désigne une chaîne contenant la séquence "`123`" ou la séquence "`321`".

Chapitre 9

Utilisation avancées de certains filtres

9.1 Introduction

Dans la section 1.7, nous avons déjà examiner la syntaxe de quelques filtres. Nous allons maintenant regarder comment certains d'entre eux utilisent les expressions régulières. Nous examinerons les filtres suivants :

- "grep",
- "egrep",
- "fgrep".

9.2 Utilisation avancée de "grep", "egrep" et "fgrep"

9.2.1 Introduction - Rappels

Le filtre "grep" est utilisé pour rechercher, dans un fichier, les lignes qui contiennent une chaîne de caractères précise.

Syntaxe :

grep [*options*] *expression-régulière* [*liste-de-fichiers*]

"grep" compare les lignes sur son entrée standard ou dans le (les) fichier(s) à une expression régulière placée sur la ligne de commande. Il envoie ensuite sur la sortie standard toutes les lignes en entrée correspondant à l'expression régulière. Si l'expression régulière contient des caractères pouvant être interprétés par le Shell, elle doit être placée dans ce cas entre simple quotes «'» ou doubles quotes """.

Principales options :

- `-c` donne uniquement le nombre d'occurrences trouvées,
- `-i` ignore les majuscules/minuscules,
- `-l` donne uniquement le nom des fichiers en entrée dans lesquels au moins une occurrence a été trouvée,
- `-n` affiche le numéro de la ligne où se trouve l'occurrence,
- `-v` donne les lignes dans lesquelles aucune occurrence n'a pas été trouvée.

Remarque 9.1 :

Les options "`-c`", "`-v`" et "`-n`" ne peuvent pas se combiner entre elles.

Exemple 9.1 :

`grep dulac /etc/passwd`	Recherche la chaine "`dulac`" dans le fichier "`/etc/passwd`".
`grep '\\' albert`	Recherche la chaine "\\" dans le fichier "`albert`" se trouvant dans le répertoire courant.
`grep \' arthur`	Recherche la chaine "'" dans le fichier "`albert`" se trouvant dans le répertoire courant.
`grep "''" the.king`	Recherche la chaine "''" dans le fichier "`the.king`" se trouvant dans le répertoire courant.

9.2.2 Utilisation de "grep"

"`grep`" accepte un ensemble restreint d'expressions régulières, en particulier, il n'accepte pas les symboles suivants :

```
+      ?
|      ( )
```

"`grep`" renvoie au shell un statut sur le résultat de sa recherche. Les différentes valeurs possibles sont :

Code	Description
0	une occurrence au moins a été trouvée.
1	aucune occurrence n'a été trouvée.
2	il y a une erreur de syntaxe ou bien un des fichiers en entrée n'est pas accessible

Remarque 9.2 :

Un statut "0" est équivalent à "Vrai".
Un statut non nul équivaut à "Faux".

On peut donc utiliser ce code dans les scripts lors de tests. Par exemple, il est possible d'afficher un message lorsque la recherche a été fructueuse ou non. Pour cela, la sortie standard de la commande "grep" sera redirigée dans un fichier spécial et seul le statut d'exécution sera utilisé.

Exemple 9.2 :

```
if grep expression fichier >/dev/null 2>&1; then
    echo "Gagné"
else
    echo "Perdu"
fi
```

Dans cet exemple, la commande "grep" effectue sa recherche dans "fichier". Tout message d'erreur ou résultat d'affichage n'apparaîtra pas à l'écran. Si la recherche donne un résultat ("$? = 0"), le message "Gagné" sera affiché. Si la recherche est infructueuse ou s'il se produit une erreur, "Perdu" sera affiché.

Exemple 9.3 :

```
grep '^[0-9][0-9]' fichier
```

Dans cet exemple, on fait appel à la notion d'expression régulière (cf. 8). L'expression "^[0-9][0-9]" se décompose de la façon suivante :

- `^` Début de ligne.
- `[0-9]` Caractère quelconque compris entre les codes ASCII de 0 et de 9. Ces caractères sont : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Par conséquent "[0-9]" représente un chiffre quelconque. Celui-ci doit apparaître une seule fois juste après le début de ligne
- `[0-9]` Le second "[0-9]" représente lui-aussi un chiffre quelconque. Celui-ci doit apparaître une seule fois juste après le précédent chiffre.

Cette expression régulière permet de décrire toute chaine de caractères commencant par deux chiffres quelconque. La commande "grep" va donc extraire de "fichier", toutes les lignes commencant par deux chiffres.

9.3 Utilisation de "egrep"

La commande "egrep" fonctionne comme la commande "grep". Par contre :

- elle accepte la définition complète des expressions régulières y compris "+", "?", "|" et "( )".
- elle admet une option supplémentaire "-f". Celle-ci permet de spécifier le nom d'un fichier contenant l'ensemble des expressions régulières à rechercher dans les fichiers présents sur la ligne de commande.

Exemple 9.4 :

```
egrep -f fic.expressions.regulieres fichier1 fichier2
```

Cet exemple donne un apercu de l'utilisation de l'option "`-f`". Toutes les lignes de "`fichier1`" et "`fichier2`" satisfaisant les expressions régulières contenues dans le fichier "`fic.expressions.regulieres`" seront affichées à l'écran.

Exemple 9.5 :

```
ypcat -k aliases | egrep '^schmoll:.*$' | cut -d@ -f2
```

Cet exemple combine l'utilisation des filtres avec le symbole "`|`". La commande "`ypcat(1)`" permet d'interroger le système d'annuaire réparti "*NIS*[1]". "`ypcat -k aliases`" interroge les entrées décrivant les boites aux lettres du réseau. Les réponses sont envoyées sur la sortie standard et leur format est :

nom-utilisateur: *boite aux lettres*

La description des boites aux lettres obéit à la syntaxe des adresses de courrier électronique SMTP[2] :

nom`@`*domaine*

La commande "`egrep`" récupère la sortie standard de la commande "`ypcat`" sur son entrée standard et recherche les lignes satisfaisant l'expression régulière"`^schmoll:.*$`". Celle-ci correspond à toute chaine commencant par "`schmoll:`" suivit d'un nombre quelconque de n'importe quels caractères.
Le résultat de cette extraction sera donc toutes les lignes de la forme :

`schmoll:` *nom*`@`*domaine*

Ce résultat est redirigé sur l'entrée standard de la commande "`cut`", qui devra extraire le second champ de chaque ligne, le caractère délimiteur étant "`@`".
Nous aurons donc, comme résultat, tous les domaines des adresses électroniques des boites aux lettres de "`schmoll`".

9.4 Utilisation de "`fgrep`"

La commande "`fgrep`" fonctionne comme la commande "`grep`" sauf qu'elle n'admet aucune expression régulière. Par contre elle dispose de l'option "`-f`" identique à la commande "`egrep`".

Le fichier contiendra alors les chaînes de caractères à rechercher sur les fichiers en entrée.

Exemple 9.6 :

1. NIS = Network Information Service [11]
2. SMTP = Simple Mail Transfert Protocol.

`fgrep "^[a-zA-Z]" fichier`
Cette commande permet de recherche la chaîne "[a-zA-Z]" très exactement sans chercher à la traiter comme une expression régulière.

Exemple 9.7 :

`fgrep -f liste fichier1 fichier2`
Cet exemple donne un apercu de l'utilisation de l'option "-f". Toutes les lignes dans "fichier1" et "fichier2" contenant les chaines présentes dans le fichier "liste" seront affichées à l'écran.

Remarque 9.3 :

La commande "fgrep" possède un algorithme plus rapide que les commandes "grep" et "egrep". Elle permet de travailler sur de très gros volumes de données.

9.5 Remarque sur l'utilisation de l'option "-f"

À la règle 10 explicitée à la section 1.3.1, nous avions préciser que la première ligne d'un script devait spécifier le nom de l'exécutable associé au shell utiliser.

Nous avons vu de même, que les commandes "egrep" et "fgrep" disposent de l'option "-f", permettant de préciser un fichier de requêtes de recherche.

Si l'analogie est faite avec le shell, le processus chargé d'évaluer chaque ligne du fichier est l'exécutable spécifié au niveau de la première ligne du fichier contenant les instructions. Par conséquent si le fichier contenant les requêtes de recherche :

- a comme première ligne "#!/usr/bin/egrep" ou "#!/usr/bin/fgrep"[3],
- est accessible en exécution,

il peut être considéré comme un filtre exécutant les requêtes de recherche associé à "egrep" ou "fgrep" qu'il contient.

Si nous prenons tout d'abord l'exemple de "egrep", l'exemple suivant permet de rechercher, sur l'entrée standard (ou bien dans la liste de fichiers spécifiée en argument), toutes les lignes satisfaisant les expressions régulières suivantes :

- "^[A-Z]", c'est-à-dire tout ligne commençant par une lettre majuscule,
- "[0-9]$", c'est-à-dire toute ligne se terminant par un chiffre,

et d'afficher le résultat sur la sortie standard.

```
#!/usr/bin/egrep
^[A-Z]
[0-9]$
```

3. Nous considérons ici que les exécutables de "egrep" et "fgrep" se trouvent dans le répertoire "/usr/bin". Pour en connaître la localisation exacte sur votre système, vous pouvez utiliser la commande "which(1)" ou "whereis(1)".

Le résultat affiché seront toutes les lignes satisfaisant au moins l'un des deux critères. Si cet exemple est enregistré dans le fichier "`localise`", et que celui-ci est accessible en exécution, il suffira de taper la commande suivante pour obtenir le résultat escompté :

`localise` *fichier* ···

ou bien, si la commande est utilisée comme un filtre :

commande_envoyant_sur_la_sortie_standard | `localise` *fichier* ···

Le principe d'utilisation pour la commande "`fgrep`" reste identique à la restriction près que "`fgrep`" recherche des chaînes sans aucune interprétation des expressions spécifiées dans le fichier. Par conséquent, si nous avons :

```
#!/usr/bin/fgrep
^[A-Z]
[0-9]$
```

cette nouvelle commande, en supposant que le fichier soit enregistré sous le nom "`localise2`", recherchera sur les fichiers spécifiés en argument, ou bien sur l'entrée standard, toutes les lignes contenant au moins l'une des deux chaines suivantes :

- "`^[A-Z]`",
- "`[0-9]$`".

Tout comme la commande précédente, l'appel pourra se faire selon les deux méthodes ci-après :

`localise2` *fichier* ···

ou bien, si la commande est utilisée comme un filtre :

commande_envoyant_sur_la_sortie_standard | `localise2` *fichier* ···

Chapitre 10

Utilisation de "sed"

10.1 Introduction

La commande "sed" est un éditeur non interactif (comme peuvent l'être "vi(1)" et "ed(1)"). Elle est utilisée généralement comme filtre.

Syntaxe :

```
sed [-n requ{\^e}te] [-e requ{\^e}te] ··· [-f fichier.requ{\^e}tes] [fichiers] ···
```

Description des options :

`-e` *requête*	Cette option indique que la requête fait partie de la ligne de commandes. Il est possible de faire appel plusieurs fois à cette option pour un même appel à la commande "sed". Les requêtes seront traitées séquentiellement pour chaque ligne lues sur l'entrée standard ou sur les fichiers spécifiés en argument.
`-f` *fichier.requêtes*	Cette option permet d'utiliser un fichier comme texte de requêtes.
`-n` *requête*	Cette option permet d'empêcher l'impression des lignes traitées.

Remarque 10.1 :

Les requêtes spécifiées avec les options "-n" ou "-e" peuvent être protégées grâce aux caractères "" " et "' " pour éviter l'interprétation par le Shell.

Exemple 10.1 :

```
sed -e 'requ1' -e 'req2' fichier.in > fichier.out
```

10.2 Mode de fonctionnement

La commande "sed" permet de copier les fichiers indiqués sur la sortie standard en l'éditant en fonction du texte de requêtes. Si aucun fichier n'est précisé en entrée, "sed" lit sur son entrée standard.

La sortie standard peut être redirigée sur un fichier avec les mécanismes classiques du Shell (cf. section 1.4). Les requêtes de la commande "sed" dont identiques à celles de la commande "ed(1)". La répétition des requêtes globales de substitution est plus efficace avec "sed" qu'avec "ed".

Pendant son exécution, la commande "sed" procède de la façon suivante :

1. Chaque ligne sélectionnée par les adresses est copiée dans un espace de travail ("*pattern space*") et toutes les commandes lui sont appliquées.
2. Lorsque toutes les commandes ont été appliquées, le contenu de l'espace de travail modifié est envoyé sur la sortie standard. Le contenu de l'espace de travail est supprimé afin de recevoir un nouveau contenu.
3. En plus de l'espace de travail, "sed" dispose d'un espace mémoire ("*hold space*") où peut être stocké temporairement le contenu de l'espace de travail. L'utilisation de cet espace est utilisé par les commandes "g", "G", "h" et "H" décrites à la section 10.3.3.

10.3 Formulation des requêtes

10.3.1 Introduction

La forme générale d'une requête est :

```
[adresse1],[adresse2]commande[argument]
```

Les adresses spécifiées dans une requêtes désignent les limites (début et fin) où doit s'appliquer la commande. L'argument représente une chaîne de caractères, ou du texte ou bien encore un fichier suivant les commandes utilisées.

10.3.2 Définition des adresses

Les adresses peuvent être décrites de deux façons :

- soit par des valeurs numériques donnant les numéros de lignes de l'espace de travail ou le caractère "$" indiquant sa dernière ligne,
- soit par des expressions régulières placées entre deux *slash* "/".

Une requête précédée par deux adresses s'applique à toutes les lignes comprises entre ces deux adresses. Dans le cas où deux adresses seraient identiques, il suffit de la spécifier seulement une fois. Si aucune adresse n'est spécifiée, l'espace de travail correspond à la totalité des lignes en entrée. La commande "sed" est donc appliquée à toutes les lignes.

Syntaxes :

Adresse1,*Adresse2Commande*
Adresse1,/*Expression*$_{régulière}$/*Commande*
/*Expression*$_{régulière}$/,*Adresse2Commande*
/*Expression*$^{1}_{régulière}$/,/*Expression*$^{2}_{régulière}$/*Commande*
AdresseCommande
/*Expression*$_{régulière}$/*Commande*
Commande

Exemple 10.2 :

On utilise ici, à titre d'exemple pour les adresses, la commande "`d`", permettant de détruire les lignes correspondantes de l'espace de travail.

`10,20d`	Détruit les lignes comprises entre la dixième et la vingtième ligne .
`/^#/,$d`	Détruit les lignes comprises entre la première ligne commençant par "`#`" jusqu'à la dernière ligne du fichier (numéro de ligne "`$`").
`/^#/,/end$/d`	Détruit les lignes comprises entre la première ligne commençant par "`#`" et la première ligne se terminant par "`end`".
`10d`	Détruit la dixième ligne.
`/10/d`	Détruit toute ligne contenant la chaîne "`10`".
`d`	Détruit l'ensemble de l'espace de travail.

10.3.3 Les Commandes

La liste des commandes citées dans ce paragraphe font parties des plus utilisées. Cette liste n'est pas exhaustive. Pour plus de renseignements, reportez vous au manuel des commandes "`sed(1)`" et "`ed(1)`".

Commande	Description
p	Copie le contenu de l'espace de travail sur la sortie standard.
a*texte*	Ajoute du texte après la position courante.
i*texte*	Ajoute du texte avant la position courante.
c*texte*	Change le texte de la ligne courant par "*texte*".
s/*expr*$_1$/*expr*$_2$/[g]	Substitue la première chaine satisfaisant l'expression régulière 1 (*expr*$_1$) par le texte correspondant à l'expression régulière 2 (*expr*$_2$). L'option "g" signifie que la substitution doit se faire autant de fois que nécessaire sur la ligne traitée. En effet, par défaut, la commande "s" ne s'applique qu'à la première chaine satisfaisant l'expression régulière dans la ligne courante de l'espace de travail. Avec l'option "g", toute chaine satisfaisant l'expression régulière "*expr*$_1$" dans la ligne courante de l'espace de travail sera remplacée.
d	Détruit la ligne courante.
g	Remplace le contenu de l'espace de travail (*pattern space*) par le contenu de l'espace mémoire (*holder space*).
G	Ajoute le contenu de l'espace mémoire (*holder space*) au contenu de l'espace de travail (*pattern space*).
h	Remplace le contenu de l'espace mémoire (*holder space*) par le contenu de l'espace de travail (*pattern space*).
H	Ajoute le contenu de l'espace de travail (*pattern space*) au contenu de l'espace mémoire (*holder space*).
r␣*fichier*	Insère le contenu d'un fichier après la ligne courante.
w␣*fichier*	Met le contenu de l'espace de travail dans un fichier.

Exemple 10.3 :

```
sed␣-n␣"/lancelot/p"␣donjons.dragons
```

Copie toutes les lignes de l'espace de travail contenant la chaine "lancelot".

```
sed␣-e␣'/lancelot/a␣et␣arthur'␣donjons.dragons
```

Insère la chaine "␣et␣arthur"[1] après la chaine "lancelot" sur la totalité de l'espace de travail.

1. Notez les espaces dans la chaine "␣et␣arthur".

```
sed␣-e␣'/dulac/ilancelot␣'␣donjons.dragons
```

Insère la chaine "`lancelot␣`"[2] avant la chaine "`dulac`" sur la totalité de l'espace de travail.

```
sed␣-e␣'s/Salut/Bonjour/g'␣welcome.txt
```

Substitue "`Salut`" par "`Bonjour`" dans la totalité du fichier "`welcome.txt`".

10.3.4 Les symboles particuliers

Les symboles particuliers sont tous ceux des expressions régulières à l'exception des symboles "`+`" et "`|`". A cette liste, on rajoute les symboles de la commande de substitution "`s`". Ces symboles sont :

Symbole	Description
"`\(`" et "`\)`"	Définit une portion d'expression régulière.
"`\`n"	Ne fait pas partie des expressions régulières mais représente la $n^{ième}$ portion d'une expression régulière.
"`&`"	Ne fait pas partie des expressions régulières mais représente la portion de ligne d'entrée qui correspond à l'expression régulière.

Exemple 10.4 :

```
s/\([␣\t]*function[␣\t]*\)\([0-z]*\)\([␣\t]*(.*)\)/#FONCTION␣\2/
```

La commande "`sed`" utilisée ici est la commande "`s`". Sa syntaxe est :

$$[adresse1[,adresse2]\texttt{s/}expr_1\texttt{/}expr_2\texttt{/}[\texttt{g}]$$

On constate donc que les adresses de début et de fin sont absentes. Par conséquent, cette requête s'applique à toutes les lignes en entrée (cf. 10.3.2). Après analyse, il apparaît que cette requête "`sed`" se décompose de la façon suivante :

```
s/
\([␣\t]*function[␣\t]*\)
\([0-z]*\)
\([␣\t]*(.*)\)
/
#FONCTION␣\2
/
```

Nous distinguons donc trois regroupements d'expression régulière :

1. "`[␣\t]*funtion[␣\t]*`"

2. Notez l'espace suivant la chaine "`lancelot`".

2. "[0-z]*"
3. "[␣\t]*(.*)"

Dans le second membre de la commande "s» ("#FONCTION \2") l'expression "\2" référence donc la chaine correspondant à "[0-z]*".
Analysons chaque regroupement du premier membre de cette requête : "[␣\t]*funtion[␣\t]*".
Tout d'abord "[␣\t]*" désigne le caractère SPACE ou bien TAB un nombre quelconque de fois. "function" représente tout simplement cette chaine de caractères. Par conséquent, "[␣\t]*funtion[␣\t]*" désigne la chaine "function" suivie et précédée d'espaces et de tabulations.
La seconde partie, "[0-z]*", désigne un des caractères dont le code ASCII est compris entre celui de "0" et celui de "z"[3], un nombre quelconque de fois. Nous avons donc, approximativement, une chaine de caractère quelconque **sans espaces ni tabulations**.
La troisième partie, "[␣\t]*(.*)" reprend la séquence "[␣\t]*" représentant une séquence d'espaces et de tabulations un nombre quelconque de fois. "(.*)" désigne le caractère "(", suivi de caractères quelconques un nombre indéterminé de fois (".*"), suivi par le caractère ")". En conclusion, cette dernière partie représente une chaine de caractères quelconque (évuentuellement vide) encadrée par des parenthèses et précédée par un nombre quelconque d'espaces et de tabulations.
Le premier membre de cette requête "sed",

\([␣\t]*function[␣\t]*\)\([0-z]*\)\([␣\t]*(.*)\)

représente donc :

1. le mot "function" précédé et suivi d'un nombre quelconque d'espaces et de tabulations,
2. une chaine de caractères quelconques **sans espaces ni tabulations**,
3. une chaine de caractères entre parenthèses précédée te suivie d'un nombre quelconque d'espaces et de tabulations.

La seconde partie de la requête "sed",

#FONCTION␣\2

substitue l'ensemble de la ligne par :

- la chaine "#FONCTION␣",
- la partie correspondant au second membre de l'expression régulière précédente, c'est-à-dire une chaine de caractères quelconques **sans espaces ni tabulations**.

Exemple 10.5 :

s/abcdef1234567890/&ghijkl/

Cette requête "sed" équivaut à :

s/abcdef1234567890/abcdef1234567890ghijkl/

3. Cette plage inclue l'ensemble des chiffres, toutes les lettres minuscules et majuscules, plus un certain nombre de caractères de ponctuation. Pour plus de précisions, reportez-vous à la table du jeu de caractères ASCII.

10.4 Exemples avancés

Pour plus d'exemples sur la commande "`sed`", reportez vous au chapitre 12.

10.4.1 Exemple 1

Le but de cet exemple est de :

But :
L'utilisateur désire connaître la localisation dans l'arborescence du système de diverses commandes.

Réalisation :
On se propose, alors, de développer un script en Bourne Shell, dont le nom est "`locate`", acceptant un nombre quelconque d'arguments représentant les commandes à localiser et d'afficher le chemin absolu de chaque exécutable correspondant. Si, par hasard, aucun fichier n'a pu être trouvé, on affichera un message d'erreur.

Syntaxe :
La syntaxe proposée, pour appeler ce script est :

`locate` *file*···

Le script obtenu est alors :

```
#!/bin/sh
if [ $# -eq 0 ]; then
    echo "‘basename $0‘: missing arguments." >&2
    echo "usage: ‘basename $0‘ file ..." >&2
    exit -1
fi
while
    [ $# -ne 0 ]
do
    find_it=0
    for dir in ‘echo $PATH | sed -e ’s/:/ /g’‘
    do
        if [ -f $dir/$1 ]; then
            echo $dir/$1
            find_it=1
            break
        fi
    done
    [ $find_it -eq 0 ] && echo "$1 not found in \"PATH\" variable." >&2
    shift
done
```

Nous allons maintenant détailler le fonctionnement.

```
#!/bin/sh
if [ $# -eq 0 ]; then
    echo "‘basename $0‘: missing arguments." >&2
    echo "usage: ‘basename $0‘ file ..." >&2
    exit -1
fi
```

Tout d'abord, le script vérifie que le nombre d'arguments est bien non nul grâce à la variable "$#". Si ce n'est pas le cas, on extrait le nom du script, contenu dans la variable "$0" pour afficher le message d'erreur correspondant, ainsi que la façon de l'utiliser.

```
while
    [ $# -ne 0 ]
do

...

    shift
done
```

Afin de pouvoir analyser tous les arguments, le programme va effectuer une boucle tant que le nombre d'arguments est non nul. L'utilisation de la commande "shift" permettra de mettre à jour la valeur qui y est contenue et l'argument qui sera traité sera toujours contenu dans la variable positionnelle "1" (son contenu sera appelé grâce à "$1").

```
find_it=0
for dir in ‘echo $PATH | sed -e 's/:/ /g'‘
do

...

done
```

La variable "PATH" contient la liste des répertoires à examiner, sachant que chaque nom est séparé par le caractère ":". La boucle "for" admet une liste de valeurs séparées par des espaces. Par conséquent, il faut fournir à "for", cette liste à partir du contenu de "PATH" dont il faudra remplacer ":" par SPACE .

Pour cela, la commande "`echo $PATH`" renvoie sur l'entrée standard de la commande "`sed`". La requête "`s/:/␣/g`" substitue chaque occurence (grâce à l'option "`g`") du caractère "`:`" par SPACE .
Le résultat de "`‘echo␣$PATH␣|␣sed␣-e␣’s/:/␣/g’‘`" est évalué par le shell lors de l'étape des "*substitutions de commandes*". Le résultat obtenu est donc :

`for dir in` *rep1 rep2 rep3* ⋯

Par conséquent, la variable "`dir`" va contenir, à chaque itération, l'un des répertoires contenus dans la variable "`PATH`". Il ne restera plus qu'à vérifier l'existance du fichier concerné, dont le nom, lui, est contenu dans la variable "`1`". Cette variable prendra, à tour de rôle, la valeur des différents arguments passé au script, gestion assuré par la boucle "`while`" associé à la commande "`shift`".
La variable "`find_it`" est un *flag* indiquant si le fichier a été trouvé lors de l'exécution de la boucle "`for`".

```
if [ -f $dir/$1 ]; then
    echo $dir/$1
    find_it=1
    break
fi
```

Cette opération est simple. Un test vérifie l'existance du fichier. S'il existe, le nom complet est affiché sur la sortie standard et le *flag* "`find_it`" est positionné à la valeur 1 et on force la sortie de la boucle "`for`". Dans le cas contraire, l'exécution de la boucle se poursuit.

```
[ $find_it -eq 0 ] && echo "$1 not found in \"PATH\" variable." >&2
shift
```

À ce stade de l'exécution, chaque répertoire contenu dans la variable "`PATH`" a été examiné, à moins qu'une sortie n'aie été provoqué lorsque le fichier a été trouvé. Dans ce cas, le script affichera le message d'information sur la sortie d'erreur standard (redirection grâce à "`>&2`") à condition que la variable "`find_it`" soit non nulle.
Il ne restera plus qu'à passer à l'argument suivant grâce à la commande "`shift`".

10.4.2 Exemple 2

Le but de cet exemple est d'afficher toutes les lignes d'un fichier se terminant par "`*/`".

Pour cela, nous allons utiliser la notion des espaces de travail de "`sed`". Nous avons vu, à la section 10.2, que "`sed`" copiait dans son espace de travail, toutes les lignes (ou *enregistrements*), arrivant sur son entrée standard, correspondant

aux critères de sélection spécifiés dans les adresses de début et de fin de la requête.

Il ne nous reste donc qu'à formuler le critère de sélection adéquat et d'afficher sur la sortie standard, le contenu de l'espace de travail.

La chaîne à considérer est donc "*/". Or le caractère "*" a une signification particulière dans une expression régulière. Par conséquent, le caractère d'échappement "\" doit être utilisé. L'expression régulière correspondant à la chaîne "*/" est donc "*/". Il ne reste plus qu'à y ajouter le caractère adéquat pour les expressions régulières caractérisant la fin d'une ligne, c'est-à-dire "$". Nous obtenons donc l'expression suivante :

```
\*\/$
```

Le critère de sélection à appliquer dans la spécification des adresses de début et de fin pour la requête "sed" correspond à toutes les lignes satisfaisant l'expression régulière précédente. Ces deux adresses seront donc identiques et correspondront à cette expression. Il suffira donc de ne la spécifier qu'une seule fois en la délimitant par le caractère "/".

La commande permettant d'afficher l'espace de travail sur la sortie standard, est "p".

Nous obtenons donc :

```
sed -n '/\*\/$/p' fichier
```

10.4.3 Exemple 3

Le but de cet exemple est de :

- supprimer les lignes commençant par "#",
- sur une ligne, tout ce qui suit le caractère "#".

Nous devons donc procéder en deux étapes :

Première étape :
: Le premier traitement à effectuer est celui de supprimer toute ligne commançant par "#". En effet, si notre commande supprime tout d'abord tout ce qui suit le caractère "#", y compris celui-ci, nous obtiendrons une ligne blanche à supprimer, qu'il ne sera pas possible de distinguer d'une ligne déjà blanche qui, elle, ne devra pas être supprimée.

Seconde étape :
: Le traitement doit s'effectuer sur le résultat de la première étape. Nous obtenons une série d'enregistrements (*lignes de fichiers*) dont, aucune ne commence par "#". Il ne restera donc plus qu'à substituer tout ce qui suit le caractère "#" par rien, caractère "#" compris. Ce traitement s'effectuera sur toutes les lignes. L'opération de substitution ne s'exécutera donc seulement si il existe une occurence du caractère "#" sur la ligne.

Deux méthodes sont possibles :

- soit nous utilisons deux commandes "`sed`" disposant chacune de la requête associée à l'étape considérée et reliées entre elles par le mécanisme des "*pipe*" grâce au caractère "`|`",
- soit nous utilisons une seule commande "`sed`" devant exécuter ces deux requêtes.

Dans un but pédagogique, nous montrerons la seconde méthode. En effet, elle mettra en évidence comment combiner les requêtes entre-elles et aussi la façon de les ordonner.

Examinons la requête à écrire pour exécuter la première étape. Tout d'abord, la commande "`sed`" associée à la suppression est "`d`". À cette commande, la spécification des adresses de début et de fin doit correspondre à toute ligne commençant par "`#`". Nous allons donc utiliser une expression régulière et les deux adresses seront identiques. L'expression régulière adéquate correspond à :

1. début de ligne,
2. caractère "`#`",
3. quelque chose éventuellement, c'est-à-dire un caractère quelconque un nombre quelconque de fois (zéro éventuellement).

Les correspondances, au niveau des expressions régulières sont :

Description	Expression
début de ligne	`^`
caractère "`#`"	`#`
quelque chose éventuellement	`.*`

La requête correspondant à la première étape est donc :

```
/^#.*/d
```

Examinons maintenant la requête à écrire pour exécuter la seconde étape. Comme il a été dit précédemment, cette requête devra s'appliquer sur toutes les lignes obtenues par la première. Sachant que les requêtes s'exécutent séquentiellement, lors de l'appel à la seconde, les enregistrements qui lui sont fournis sont ceux qui résultent de la *première* exécution. Par conséquent, les adresses de début et de fin ne sont pas à spécifier car ce sont toutes les lignes qu'il faut traiter.

La commande de substitution de "`sed`" est "`s`". Sa syntaxe est :

$$\texttt{s/}expression_1\texttt{/}expression_2\texttt{/[g]}$$

Ici, "$expression_1$" devra correspondre à la séquence suivante :

- caractère "`#`",
- quelque chose éventuellement, c'est-à-dire une suite de caractères quelconques un nombre quelconque de fois,
- fin de ligne.

Quant à "*expression*$_2$", elle devra représenté ce par quoi "*expression*$_1$" devra être remplacé, c'est-à-dire "**rien**". Par conséquent, "*expression*$_2$" **sera vide**. Il ne nous reste donc plus qu'à formuler l'expression régulère associée à "*expression*$_1$". En fonction de ce que doit décrire "*expression*$_1$", nous avons les correspondances suivantes avec les expressions régulières :

Description	Expression
caractère "#"	`#`
quelque chose éventuellement	`.*`
fin de ligne	`$`

La requête correspondant à la seconde étape est donc :

```
s/#.*$//
```

Nous prenons donc bien en compte tout ce qu'il y a entre la première occurence du caractère "`#`" jusqu'à la fin de la ligne. Cette entité est substituée par "*rien*".

Nous obtenons donc :

```
sed -e '/^#.*/d' -e 's/#.*$//' fichier
```

10.4.4 Exemple 4

Le but de cet exemple est de récupérer le champ "*login*" et "*UID*" du fichier "`/etc/passwd`".

D'après le format utilisé pour ce fichier (explicité dans `passwd(5)`), tous les champs sont séparés par le caractère "`:`". Par conséquent, le contenu de chaque champ interdit l'utilisation de ce caractère. Les champs de ce fichier auront donc comme propriété :

> Les caractères contenus pour chaque champ du fichier `/etc/passwd` sont tous les caractères alphanumériques valides à l'exception du caractère "`:`".

Pour représenter une chaîne de caractères ne contenant pas "`:`" à l'aide des expressions régulières, nous allons utiliser les plages de caractères en excluant les caractères non valides. Nous aurons donc :

```
[^:]*
```

Les champs "*login*" et "*UID*" représentent les premier et troisième champ du fichier "`/etc/passwd`". Par conséquent, la méthode à suivre pour décrire notre requête est la suivante :

- le début de ligne,

- une séquence de caractères ne contenant pas "`:`" permettant d'isoler le premier champ, donc le champ "*login*",

- le caractère ":",
- une séquence de caractères ne contenant pas ":" permettant d'isoler le second champ,
- le caractère ":",
- une séquence de caractères ne contenant pas ":" permettant d'isoler le troisième champ, donc le champ "*UID*",
- une séquence de caractères quelconques (":" pouvant y apparaître),
- la fin de la ligne.

Nous avons donc identifié quatre groupements :

- le premier associé au champ "*login*",
- le second associé à une séquence de caractères ne contenant pas ":", encadrée à gauche et à droite par ":"[4],
- le troisième associé au champ "*UID*",
- le quatrième associé à une séquence de caractères quelconques (":" pouvant y apparaître) jusqu'à la fin de la ligne.

Examinons maintenant l'expression régulière associée :

Description	Expression
début de ligne	`^`
séquence de caractères ne contenant pas ":"	`[^:]*`
le caractère ":"	`:`
séquence de caractères ne contenant pas ":"	`[^:]*`
le caractère ":"	`:`
séquence de caractères ne contenant pas ":"	`[^:]*`
séquence de caractères quelconques	`.*`
fin de ligne	`$`

4. Ce regroupement permet d'isoler le second champ avec les caractères de délimitation.

En fonction des regroupements explicités précédemment, nous obtenons l'expression suivante :

```
^\([^:]*\)\(:[^:]*:\)\([^:]*\)\(.*$\)
```

Maintenant, il ne nous reste plus qu'à prendre tout ce qui correspond au premier et troisième regroupement dans chaque enregistrement (lignes arrivant sur l'entrée standard), et formater la sortie de la façon suivante :

login	TAB	*UID*

donc :

$regroupement_1$	TAB	$regroupement_3$

Nous obtenons donc :

```
sed␣-e␣'s/^\([^:]*\)\(:[^:]*:\)\([^:]*\)\(.*$\)/\1\t\3/'␣/etc/passwd
```

10.4.5 Exemple 5

Le but de cet exemple est de copier de tous les utilisateurs précédent l'entrée du compte "uucp" dans "/etc/passwd" et déplacer de les à la fin, compte "uucp" compris. Le résultat sera affiché sur la sortie standard.

Pour cela on fait appel à un fichier de commandes "sed" que l'on nommera "script.sed". L'appel se fera de la façon suivante :

```
sed -f script.sed /etc/passwd
```

Détaillons les opérations à effectuer :

- *Couper* dans un espace de travail particulier, à partir de la première ligne jusqu'à celle commençant par «uucp".
- Se placer à la fin de l'espace mémoire contenant les enregistrements à traiter par "sed" et *coller* le contenu de l'espace précédent.

Pour cela, nous allons utiliser les commandes liées à la gestion de l'espace mémoire et l'espace de travail de "sed". Comme il l'a été décrit à la section 10.2, "sed" dispose d'un espace mémoire permettant de stocker temporairement, un certain nombre d'informations.

La première étape consiste donc à partir du début du fichier (donc de la première ligne) et parcourir jusqu'à la première ligne commençant par «uucp". Sachant que les nom d'utilisateurs sont uniques, le fichier ne contiendra qu'une et une seule ligne commençant par «uucp". Les adresses de début et de fin sont donc :

Description	Expression
Première ligne	1
Première ligne commençant par "uucp"	/^uucp/

La commande "sed" appropriée pour stocker l'ensemble de ces lignes dans un espace mémoire est "H" (cf. section 10.3.3). La requête associée est donc :

```
1,/^uucp/H
```

À ce stade de l'exécution, l'enregistrement courrant arrivant sur l'entrée de "sed" est la ligne du fichier "/etc/passwd" définissant le compte utilisateur suivant celui de "uucp". Aucune action n'est spécifiée. Les lignes sont donc réécritures sans aucune opération sur la sortie standard.

Lorsque la fin de fichier est rencontrée, "sed" doit afficher ce qu'il a précédemment mémoriser dans son espace mémoire. Par conséquent, l'adresse valide pour exécuter la requête est celle correspondant à la fin de fichier : "$". Ici les adresses de début et de fin sont identiques : elles doivent correspondre toutes les deux à la fin de fichier. La commande "sed" appropriée pour afficher sur la sortie standard l'ensemble de ces lignes précédemment stockées dans un espace mémoire est "G" (cf. section 10.3.3). La requête associée est donc :

```
$G
```

En conclusion, le contenu du fichier "script.sed" est :

```
1,/^uucp/H
$G
```

Remarque 10.2 :

L'espace mémoire utilisé est, bien sûr, propre à "sed". Les opérations "g" et "h" sont similaires mais avec un espace vide au départ.

10.5 Remarque sur l'utilisation de l'option "-f"

À la règle 10 explicitée à la section 1.3.1, nous avions préciser que la première ligne d'un script devait spécifier le nom de l'exécutable associé au shell utiliser.

Nous avons vu de même, que la commande "sed" dispose de l'option "-f", permettant de préciser un fichier de requêtes "sed".

Si l'analogie est faite avec le shell, le processus chargé d'évaluer chaque ligne du fichier est l'exécutable spécifié au niveau de la première ligne du fichier contenant les instructions. Par conséquent si le fichier contenant les requêtes "sed" :

- a comme première ligne "#!/usr/bin/sed"[5],
- est accessible en exécution,

il peut être considéré comme un filtre exécutant les requêtes "sed" qu'il contient.

Par exemple, si cette notion est appliquée au fichier décrit dans l'exemple de la section 10.4.5, le contenu de ce fichier sera :

```
#!/usr/bin/sed
1,/^uucp/H
$G
```

5. Nous considérons ici que l'exécutable de "sed" se trouve dans le répertoire "/usr/bin". Pour en connaître la localisation exacte sur votre système, vous pouvez utiliser la commande "which(1)" ou "whereis(1)".

La commande qui sera saisie deviendra :

```
script.sed /etc/passwd
```

Chapitre 11

Utilisation de "awk"

11.1 Introduction

"awk" est un processeur d'éléments syntaxiques qui :

- sert à traiter, de manière non interactive, un texte au niveau de ses éléments syntaxiques,
- travaille sur la base d'une ligne, et ligne après ligne,
- permet de définir, analyser, transformer les mots ou éléments syntaxiques qui composent chaque ligne.

"awk" traitera chaque ligne en entrée comme un enregistrement. C'est ce terme qui sera employé par la suite. Chaque enregistrement est composé de champs. Ces champs sont séparés, par défaut, par un ou plusieurs espaces ou bien une ou plusieurs tabulations. C'est aussi un langage de programmation dont la fonction première est de rechercher des chaînes de caractères suivant certains critères et d'y appliquer des actions. On aura donc toujours un modèle :

sélection + *action*

Le corps de chaque action est un bloc constitué d'une ou plusieurs commandes, délimité par les caractères "{" et «}". On aura alors :

```
sélection { commande ··· }
```

Pour la petite histoire, le nom "awk" est dérivé des initiales de ses auteurs: Alfred V. AHO, Peter J. WEINBERGER et Brian W. KERNINGHAN.

Syntaxe :

`awk '`*corps du programme awk* `' [`*fichier*`=··· ]`
ou `awk -f` *fichier.programme* `[`*fichier* `···]`

Dans le premier cas de syntaxe, le corps du programme "awk" est appliqué directement aux fichiers ou à l'entrée standard. Dans le second cas de syntaxe, le programme "awk" est contenu dans un fichier et appliqué aux fichiers ou à l'entrée standard.

11.2 Les sélecteurs

11.2.1 Introduction, Définition

On a vu que chaque requête "awk" était constituée d'une sélection et d'une action. Nous allons examiner dans ce paragraphe la syntaxe pour la spécification des sélections. Les sélections des lignes en entrées sont décrites par des sélecteurs.

Un sélecteur est :

- une expression régulière,
- un mot clef.

Chaque fois que l'expression décrite par un sélecteur est vraie, l'action correspondante est exécutée. Les sélecteurs et les actions peuvent faire appel à des variables, à des paramètres et à des expressions logiques. Les actions, elles, peuvent contenir des structures de contrôle ("if", "while", etc.).

11.2.2 Les sélecteurs prédéfini

Les deux seuls sélecteurs prédéfinis dans "awk" sont :

- BEGIN
- END

Le sélecteur "BEGIN" est vrai avant le traitement du premier enregistrement. Ce sélecteur est généralement utilisé pour initialiser des variables, définir des champs de saisie, etc.

Le sélecteur "END" est vrai après le traitement du dernier enregistrement. Ce sélecteur est généralement utilisé pour afficher les résultats finaux.

Exemple 11.1 :

L'exemple suivant permet d'afficher les noms de tous les fichiers du répertoire et leurs tailles.

```
ls -al | awk '
    BEGIN {
        totalsize = 0
        print "Fichier(Taille) du r{\'e}pertoire courant."
    }
    {
        totalsize += $5
        printf ("%s (%d)\n", $9, $5)
    }
    END {
        printf (Taille totale = %d octet(s).\n", totalsize)
        print "Fin de la liste."
    }
'
```

Pour rappel, la commande "ls -al" permet d'obtenir la liste de tous les fichiers dans le répertoire courant, y compris les fichiers cachés.

Le cinquième champ correspond à la taille du fichier, le neuvième à son nom (cf. section 2.1.2).
Lors de l'exécution du sélecteur "`BEGIN`", le programme "`awk`" initialise la variable "`totalsize`" à 0, afin de pouvoir effectuer un cumul sur le cinquième champ, c'est-à-dire la taille des fichiers. Il affiche aussi un message avant exécution.
Pour chaque enregistrement, c'est-à-dire pour chaque fichier, on cumule sur le cinquième champ ("`totalsize += $5`") et on affiche les informations désirées.
Enfin, lors de l'exécution du sélecteur "`END`", le programme "`awk`" affiche la valeur du cumul (variable "`totalsize`") et un message de fin.
Par exemple, si la commande "`ls -al`" produit le résultat suivant :

```
-rw-r----- 1 schmoll esme     102 Jul 12 10:00 .cshrc
-rw-r----- 1 schmoll esme     123 Jul 12 10:00 .login
-rw-r----- 1 schmoll esme     124 Jul 12 10:00 .logout
-rw-r----- 1 schmoll esme      24 Jul 25 10:00 mes
-rw-r--r-- 1 schmoll esme      12 Jul 25 12:00 fichiers
-rwxr-xr-x 1 schmoll esme    1337 Jul 25 12:00 a
-rw-r--r-- 1 schmoll esme 1221337 Jul 25 12:00 moi
```

le programme "`awk`" produira la sortie suivante :

```
Fichier(Taille) du r{\'e}pertoire courant.
.cshrc (102)
.login (123)
.logout (124)
mes (24)
fichiers (12)
a (1337)
moi (1221337)
Taille totale = 1223058 octet(s).
```

11.2.3 Les règles de sélection

Un sélecteur est une expression régulière qui va être comparée à l'enregistrement courant du fichier en entrée. Si une correspondance est trouvée entre l'expression régulière et l'enregistrement, le sélecteur devient vrai et l'action correspondante est exécutée. Cette expression régulière peut aussi s'appliquer seulement sur un ou plusieurs champs.

11.2.4 Les caractères spéciaux pour la sélection

Caractère	Description
/	Délimiteur d'une expression régulière.
"	Délimiteur d'une chaîne de caractères.

Suite page suivante · · ·

Suite de la page précédente ···

Caractère	Description
\	Permet de spécifier un caractère spécial en tant que caractère normal. Par exemple "\/" désigne le caractère "/".
$$n$	Représente le $n^{ième}$ champ de l'enregistrement courant. La valeur de "n" n'est limitée que par le nombre de champs dans l'enregistrement courant.
$0	Représente la totalité de l'enregistrement en entrée.

11.2.5 Les expressions logiques pour la sélection

Opérateur	Description
<	Inférieur à
>	Supérieur à
==	Égalité
!=	Différent
&&	*ET* logique
||	*OU* logique
~	Permet de comparer l'expression régulière à un champ précis.

11.2.6 Syntaxe des sélecteurs

La syntaxe des sélecteur peut s'exprimer de trois façons :

- sélection en fonction d'une expression régulière,
- sélection en fonction d'expression logiques,
- sélection en utilisant les deux formats précédents.

Dans le premier cas de figure, si aucun champ n'est spécifié dans le sélecteur, l'expression régulière s'applique à l'ensemble de l'enregistrement. On aura donc les syntaxes suivantes :

/expression$_{régulière}$/
$champ ~ /expression$_{régulière}$/

Dans le second cas de figure, on peut écrire une expression logique en utilisant les opérateurs logiques précédemment décrits. On pourra donc avoir, par exemple :

```
$1 == $2
$2 < $3
```

```
$2 != $3
( $2 < $3 ) && ($2 > $4)
(($2 > 100) || ( $2 == $3*50)) && ($4 > 10)
```

Il est évidemment possible d'utiliser les opérateurs logiques pour relier les expressions correspondant au premier cas de figure. On pourra donc avoir :

```
($1 ~ /[a-z]/) && ($2 ~ /[0-9]/)
($1 ~ /[a-z]/) && ($2 ~ /[0-9]/) && ($2 < 10)
```

11.3 Les variables

"awk" permet l'utilisation de variables internes. Ces variables peuvent être de type numérique ou bien alphanumérique.

Il possède aussi un certain nombre de variables pré définies permettant de paramètrer l'environnement de "awk" ou bien d'obtenir des informations sur le contexte courant. Ces variables sont :

Variable	Description
FILENAME	nom du fichier courant en entrée.
NR	numéro de l'enregistrement courant.
NF	nombre de champs dans l'enregistrement courant.
FS	caractère séparateur de champs.
RS	caractère de séparation d'enregistrements.
$0	l'enregistrement courant.
$n	le $n^{ième}$ champ de l'enregistrement courant.
IFS	caractère séparateur de champs en entrée.
IRS	caractère de séparation d'enregistrements en entrée.
OFS	caractère séparateur de champs en sortie.
ORS	caractère de séparation d'enregistrements en sortie.
OFMT	format de sortie pour les chiffres.

Exemple 11.2 :

Commande :

```
awk '
    BEGIN {
        print "Liste du fichier ", FILENAME
        print
        printf ("N.L.\tUID"\t|\tNom de login\n")
        FS=":"
        OFS="\t|\t"
    }
    {
        printf ("%03d:\t",NR)
        print $3,$1
    }
' /etc/passwd
```

Le programme "awk" précédent s'applique au fichier "/etc/passwd",

contenant la liste des utilisateurs autorisés à se connecter au système. Ce fichier se compose des champs suivants séparés par le caractère ":":

- le nom de connexion ou "*logname*",
- le mot de passe crypté de l'utilisateur,
- son identifiant numérique unique, l'UID,
- son identifiant numérique de groupe, le GID,
- un champ de commentaires, appelé "GCOS", contenant usuellement le prénom et le nom complet de l'utilisateur,
- son répertoire de connexion, c'est-à-dire le répertoire qu'il aura par défaut lorsqu'il se connectera au système[1]
- le nom de l'exécutable correspondant à son interpréteur de commandes.

Pour plus de renseignements sur ce fichiers, reportez-vous à "passwd(5)".

Ce programme se décompose en deux parties :

1. Le sélecteur "BEGIN" permet d'afficher un texte prélimaire indiquant :
   - la liste des utilisateurs du fichier traité (référencé par la variable "FILENAME",
   - une ligne blanche (commande "print" sans arguments),
   - la première ligne du tableau généré par le programme.

   L'initialisation de la variable "FS" permet de spécifier la nouvelle valeur du séparateur de champ. Celui-ci devient le caractère ":". L'initialisation de la variable "OFS" permet de spécifier le nouveau format d'affichage de la commande "print". Chaque argument de cette commande sera séparé par le caractère "|" précédé et suivi d'une tabulation.

2. Le second sélecteur ne possède aucun critère, les actions qui y sont présentes s'appliquent à chaque enregistrement du fichier c'est-à-dire à chaque ligne. Les deux actions présentes ont les fonctions suivantes :
   - la première affiche le numéro de l'enregistrement courant sur trois chiffres[2] grâce à la variable "NR",
   - la seconde affiche le troisième et le premier champ en utilisant le format de sortie contenu dans la variable "OFS".

Le résultat obtenu sera donc :

```
liste du fichier /etc/passwd

N.L. UID | Nom de login
001: 0 | root
```

etc.

1. Ce répertoire est celui utilisé par défaut par la commande "cd", cf. section 2.1.1.
2. Les formats utilisés par la commande "printf" de "awk" sont identiques à ceux de la fonction "printf(3)".

La définition des variables utilisateur est dynamique. Par conséquent, leur déclaration se fait à la première utilisation et leur type s'adapte en fonction de la valeur à laquelle elle a été initialisée. Le type d'une variable dépend de son utilisation. Il peut être :

- numérique (virgule flottante ou entière)
- alphanumérique

Les variables sont déclarées de façon globale à tous les blocs action de toutes les requêtes "`awk`". Par conséquent, une variable créée dans la requête "`BEGIN`", sera accessible par toutes les requêtes "`awk`".

11.3.1 Les tableaux

"`awk`" sait aussi gérer des tableaux. Les types disponibles pour ces tableaux sont les mêmes que les variables. On a donc les types suivants :

- type numérique (virgule flottante ou entier),
- alphanumérique.

La dimension d'un tableau est dynamique. Il n'y a donc pas besoin de déclarer sa dimension avant de l'utiliser, il suffit de l'initialiser. Les indices peuvent être numériques ou bien alphanumériques, c'est-à-dire qu'un élément du tableau peut être représenté par un indice numérique ou bien par un indice symbolisé par une chaîne de caractères. Ce deuxième cas correspond aux tableaux "*associatifs*". Tout comme les variables, les tableaux sont communs à tous les blocs "*action*" de toutes les requêtes "`awk`" : ils sont déclarés "*en global*".

Exemple 11.3 :

Indiçage de tableaux :

```
array[0]="ABC"
other_array[schmoll]="bidule"
```

11.4 Les actions

Les actions sont des blocs constituant une requête "`awk`". Ils décrivent les opérations à effectuer lorsque la sélection décrite en tête de requête est vérifiée. Les gens connaissant le langage C retrouveront la plupart des fonctions standards avec des syntaxes identiques. On trouvera aussi un ensemble de fonctions spécifiques.

Les opérateurs arithmétiques autorisés dans ces blocs sont explicités dans le tableau suivant :

Opérateur	Description
=	affectation
+	addition binaire
++	incrémentation de 1
	Suite page suivante · · ·

Suite de la page précédente ···

Opérateur	Description
+=	addition unaire
-	soustraction binaire
--	décrémentation de 1
-=	soustraction unaire
*	multiplication binaire
*=	multiplication unaire
/	division binaire
/=	division unaire
%	modulo

La concaténation de champs se fait sans opérateur spécifique. Il suffit de lister les chaînes à concaténer.

Exemple 11.4 :

```
cumul=0
taux=$2
nom=$1 "du genoux" $2
nom=$1 $2
cumul=cumul + 10
ttc=$4 + $5
cumul ++                          identique à "cumul = cumul + 1"
cumul += $2                       identique à "cumul = cumul + $2"
cumul = cumul - $2
ht = ttc - tva
cumul --                          identique à "cumul = cumul - 1"
cumul -= $2                       identique à "cumul = cumul - $2"
total = cumul * 100
carre = $1 * $1
cumul *= $2                       identique à "cumul = cumul * $2"
pourcent = $2 / cumul * 100
cumul /= $2                       identique à "cumul = cumul / $2"
reste = cumul % 3
```

11.4.1 Les fonctions prédéfinies

La liste des fonctions suivantes n'est pas exhaustive. Elle donne celles qui sont usuellement utilisées. Les tableaux explicitent ces fonctions. Le tableau 11.4 donne une liste de fonctions acceptant un argument de type numérique, le tableau 11.5 donne une liste de fonctions acceptant un argument de type "*chaîne de caractères*".

Remarque 11.1 :

Pour plus de précisions sur les instructions de formatage, reportez-vous à l'annexe A.

Exemple 11.5 :

Fonction	Description
`sqrt(arg)`	renvoie la racine carré de l'argument.
`log(arg)`	renvoie le logarithme népérien de l'argument.
`exp(arg)`	renvoie l'exponentiel de l'argument.
`int(arg)`	renvoie la partie entière[a] de l'argument..

TAB. 11.4 – *Fonctions acceptant un argument de type numérique.*

[a] C'est-à-dire la fonction mathématique $E(x)$.

Fonction	Description
`length`[a]	renvoie la longueur de l'enregistrement courant.
`length(arg)`	renvoie la longueur de la chaine passée en argument.
`substr(arg,m[,n])`	renvoie la sous chaîne de la chaine "*arg*" commençant à la position "*m*" et de longueur "*n*". Le premier caractère est à la position "1"[b]. Si "*n*" n'est pas spécifié, la fonction "`substr`" renvoie la fin de l'argument à partir de la position "*m*".
`index(`str_1`,`str_2`)`	renvoie la position de "str_2" dans la chaîne "str_1". Si "str_1" ne contient pas la chaîne "str_2", "`index`" renvoie "0"[c].
`print [`arg_1`[,`arg_2`],`$\cdots$`] [> dest]`	affiche les arguments "arg_1", "arg_1", $\cdots$ sur la sortie standard sans les formater. Avec l'option "`> dest`", l'affichage est redirigé sur le fichier "*dest*" au lieu de la sortie standard.
`printf(format,`arg_1`,`arg_2`,`$\cdots$`) [> dest]`	affiche les arguments arg_1, arg_2, $\cdots$ sur la sortie standard après les avoir formatés à l'aide de la chaîne de contrôle "*format*". Avec l'option "`> dest`", l'affichage est redirigé sur le fichier "*dest*" au lieu de la sortie standard.
`sprintf(format,`arg_1`,`arg_2`,`$\cdots$`)`	renvoie une chaîne de caractères formatée intégrant arg_1, arg_2, $\cdots$ correspondant aux instruction de formatage en fonction de la chaîne de contrôle "*format*". **Attention**, cette fonction a un comportement différent de celui de la fonction "`sprintf(3)`".

TAB. 11.5 – *Fonctions acceptant des arguments de type alphanumérique.*

[a] L'appel à "`length`" ne possède aucun argument.
[b] Alors qu'en langage C, le premier caractère d'une chaine est à la position "0".
[c] 0 ne correspond à aucune position dans une chaine puisque, dans "`awk`", le premier caractère d'une chaine a la position "1", **ce qui n'est pas le cas en langage C.**

Exemple d'utilisation des fonctions prédéfinies :

```
if (length > 80 ) {
    print "la ligne no: ", NR, " du fichier ",FILENAME, \
        "est trop longue"
}
lentexte += length ($2)
val=sqrt (cumul)
val = log(sqrt(cumul))
val = exp(log(cumul))
modulo = int(cumul/100) * 100
codepostal = substr("75006 Paris",1,5)
pos = index("75006 Paris","Paris")      renvoie 7
print $1,$2,cumul
print "resultats", cumul > /tmp/result.tmp
print "cumuls", $1+$2 > $3
nom = sprintf ("%10.10s %10.10s .", nom, prenom)
```

11.4.2 Les fonctions utilisateur

En plus des fonctions prédéfinies, l'utilisateur peut définir ses propres fonctions. Ces fonctions peuvent se trouver n'importe où dans le corps du programme "awk". La déclaration d'une fonction se fait de la façon suivante :

```
function nom_fonction (arguments)
{
    instructions
}
```

La fonction peut être appelée dans n'importe quel bloc action d'une requête "awk". Il suffit de la référencer par son nom. Les fonctions utilisateurs peuvent être récursives.

Exemple 11.6 :

```
function factoriel (num)
{
    if (num == 0) return 1
    return (num * factoriel(num - 1))
}

$1 ~ /^Factoriel$/ { print factoriel($2) }
$1 ~ /^Minimum$/        { print minimum ($2, $3) }

function minimum (n,m) {
    return (m < n ? m : n)
}
```

11.4.3 Les structures de contrôle

L'ensemble des structures de contrôle de "`awk`" fonctionnent comme celles du langage C. Nous allons donc ne faire qu'un bref rappel sur la syntaxe des différentes structures de contrôles disponibles.

Dans toute la suite de ce paragraphe, le terme instruction désigne un ensemble d'instructions "`awk`" séparées par le caractère "`;`" ou "RETURN "et encadrées par des "{","}".

Structure de contrôle "`if`, `else`":

```
if (condition)
    instruction
else
    instruction
```

Structure de contrôle "`while`":

```
while (condition)
    instruction
```

Structure de contrôle "`for`":

```
for (init;condition;it{\'e}ration)
    instruction
```

Instruction "`break`":

L'instruction "`break`" provoque la sortie du niveau courant d'une boucle "`while`" ou "`for`".

Instruction "`continue`":

L'instruction "`continue`" provoque l'itération suivante au niveau courant d'une boucle "`while`" ou "`for`".

Instruction "`next`": L'instruction "`next`" force "`awk`" à passer à la ligne suivante du fichier en entrée.

Instruction "`exit`":

L'instruction "`exit`" force "`awk`" **à interrompre** la lecture du fichier d'entrée comme si la fin avait été atteinte.

Exemple 11.7 :

```
if ($3 == foo*5) {
    a = $6 % 2;
    print $5, $6, "total", a;
    b = 0;
}
else {
    next
}
while ( i <= NF) {
```

```
        print $i;
        i ++;
    }
    for (i=1; ( i<= NF) && ( i <= 10); i++) {
        if ( i < 0 ) break;
        if ( i == 5) continue;
        print $i;
    }
```

Remarque 11.2 :

Ces structures de contrôles et les instructions qui y sont liées , bien que leur nom soit identique à celui utilisé pour les structures du shell sont différentes en de multiples points :

- *tout d'abord, leur syntaxe est différente,*
- *les structures de contrôles pour "awk" s'exécutent dans le processus lié à cet utilitaire,*
- *les structures de contrôles du shell sont exécutées dans le processus lié au script ou bien à l'interpréteur de commandes actif.*

Par conséquent, l'utilisation d'un "if" peut être lié au shell ou bien à un utilitaire comme "awk".

11.5 Trucs et astuces

11.5.1 Commentaires

Le caractère permettant d'introduire des commentaires dans un programme "awk" est le caractère "#". Tout ce qui suit ce caractère est ignoré. La notion de commentaires n'a d'intérêt que lorsque le programme "awk" se trouve dans un fichier séparé.

11.5.2 Passage de paramètres du shell vers le programme "awk"

La notion de passage d'arguments n'a d'intérêt que lorsque le programme "awk" se trouve dans un fichier séparé. Elle n'est aussi utilisable que si "awk" reçoit les données sur son entrée standard. Dans ce cas, "awk" maintient deux variables :

- "ARGC" contenant le nombre d'arguments,
- "ARGV" tableau contenant la liste des arguments passés au programme "awk".

Exemple 11.8 :

```
cat mon.fichier | awk -f program.awk a v=4 b "autre arg"
```

Dans ce cas, on aura :

Variable	Valeur
ARGC	5
ARGV[0]	awk
ARGV[1]	a
ARGV[2]	v=4
ARGV[3]	b
ARGV[4]	autre␣arg

11.5.3 Utilisation de variables du Shell dans le corps du programme "awk"

Cette notion n'est applicable que si "awk" est appelé à l'intérieur d'un Shell script et que le corps du programme "awk" ne se trouve pas dans un fichier séparé. Dans ce cas, on aura :

```
awk '
    corps du programme awk
' [fichier]
```

Pour pouvoir utiliser des variables Shell ou bien le résultat de commandes, il suffit de les insérer entre deux simples quotes (caractère "'") à condition que le bloc programme de "awk" soit compris entre deux simples quotes, ce qui est préférable par rapport aux doubles quotes (caractère """).

Exemple 11.9 :

```
#!/bin/sh
echo "Entrez une chaine :\c"
read answer
echo $answer | awk '
    BEGIN {
        status=0;
        today='‘date +%H%M‘'
    }
    NR > 1 { status=1}
    $1 ~ /[0-9].*/ {
        printf ("La chaine %s comprend une partie num{\'e}rique: %d\n",
            '$answer', $1);
    }
    END {
        printf ("Execut{\'e}e le %s\n", today)
    }
'
```

ATTENTION

11.6 Exemple

Soit le fichier "donjon" :

```
LANCELOT:DULAC:chevalier:200
GODEFROY:DE BOUILLON:chevalier:300
ARTHUR::roi:150
MERLIN:L'ENCHANTEUR:magicien:0
GODEFROY:DE BOUILLON:chevalier:400
LANCELOT:DULAC:chevalier:300
```

Soit le fichier programme "cumul.awk" :

```
BEGIN {
    FS=":"
    print "Statistiques sur le nombre de sarrasins occis"
    nb=0
}

$1 ~ /[a-zA-Z]/ {
    cumul[$1] += $4
    for (i=1; (i <= nb) && (list[i] != $1; i++)
    if ( i > nb) {
        nb ++
        list[nb]=$1 $2
    }
}
END {
    printf ("%.20s\t%.20s \n","Zigoto", "Cumul")
    for (i=1; i <= nb; i++)
        printf ("%.20s\t%.20s \n",
             list[i], cumul[list[i]])
}
```

La commande

```
awk -f cumul.awk donjon
```

produit le résultat suivant :

```
Statistiques sur le nombre de sarrasins occis
Zigoto                  Cumul
LANCELOT DULAC          500
GODEFROY DE BOUILLON    700
ARTHUR                  150
MERLIN L'ENCHANTEUR     0
```

11.7 Remarque sur l'utilisation de l'option "-f"

À la règle 10 explicitée à la section 1.3.1, nous avions préciser que la première ligne d'un script devait spécifier le nom de l'exécutable associé au shell utiliser.

Nous avons vu de même, que la commande "awk" dispose de l'option "-f", permettant de préciser un fichier de requêtes "awk".

Si l'analogie est faite avec le shell, le processus chargé d'évaluer chaque ligne du fichier est l'exécutable spécifié au niveau de la première ligne du fichier contenant les instructions. Par conséquent si le fichier contenant les requêtes "awk" :

- a comme première ligne "#!/usr/bin/awk"[3],
- est accessible en exécution,

il peut être considéré comme un filtre exécutant les requêtes "awk" qu'il contient.

Par exemple, si cette notion est appliquée au fichier décrit dans l'exemple de la section 11.6, le contenu du fichier "cumul.awk" :

```
#!/usr/bin/awk
BEGIN {
    FS=":"
    print "Statistiques sur le nombre de sarrasins occis"
    nb=0
}

$1 ~ /[a-zA-Z]/ {
    cumul[$1] += $4
    for (i=1; (i <= nb) && (list[i] != $1; i++)
    if ( i > nb) {
        nb ++
        list[nb]=$1 $2
    }
}
END {
    printf ("%.20s\t%.20s \n","Zigoto", "Cumul")
    for (i=1; i <= nb; i++)
        printf ("%.20s\t%.20s \n",
             list[i], cumul[list[i]])
}
```

La commande qui sera saisie deviendra :

```
cumul.awk donjon
```

Si nous voulons utiliser le programme "cumul.awk" comme un filtre, il sera alors possible de saisir la commande suivante :

```
cat donjon | cumul.awk
```

3. Nous considérons ici que l'exécutable de "awk" se trouve dans le répertoire "/usr/bin". Pour en connaître la localisation exacte sur votre système, vous pouvez utiliser la commande "which(1)" ou "whereis(1)".

Dans tous les cas, le résultat obtenu sur la sortie standard sera identique à celui montré à la section 11.6.

Chapitre 12

Programmation avancée de Shell Scripts

12.1 Introduction

Ce chapitre a pour vocation de présenter un certain nombre de commandes en utilisant des exemples complexes.

Les exemples que nous aborderons, auront les fonctionnalités suivantes :

Exemple 1 :
Cet exemple se chargera de trier le fichier "`/etc/hosts`" par adresse IP.

Exemple 2 :
Cet exemple devra localiser le dernier "UID" disponible dans une plage de valeurs prédéfinies.

Exemple 3 :
Cet exemple se chargera de créer toutes les entrées nécessaires pour créer des comptes "utilisateur" à partir de fichiers de données. Cet exemple est utilisé dans un cas réel : création des comptes utilisateur UNIX à partir des définitions des comptes OpenVMS.

12.2 Tri par adresse IP du fichier "`/etc/hosts`"

12.2.1 Étude des fonctionnalités

Le fichier "`hosts(5)`" permet au système UNIX d'associé une adresse réseau IP à un ou plusieurs noms de machine.En se reportant au manuel du fichier "`hosts(5)`", le format utilisé est le suivante :

- Le fichier peut contenir des commentaires, tout caractère saisi après le caractère "`#`" est ignoré,
- Chaque ligne décrivant l'association "*adresse IP,noms de machine*" est

composée d'au moins deux champs séparés par un ou plusieurs espaces ou tabulations. Le premier champ est l'adresse IP de la machine sous sa forme usuelle (format décimal), le second est le nom officiel de la machine, c'est-à-dire celui qui lui sera vraiment affecté. Les autres champs, séparés de la même façon sont les noms non-officiels, c'est-à-dire une série de noms qu'il sera possible d'utiliser pour la joindre.

Le problème est donc de faire un tri **numérique** sur les quatre champs de l'adresse IP (chacun étant séparé par le caractère "." sachant que le séparateur de champ du fichier "hosts" est différent et que les champs le constituant sont différents. De plus, le caractère "." peut être utilisé dans les noms de machines, en particulier lorsque les domaines Bind/DNS sont précisés (cf. [8]).

12.2.2 Méthode utilisée

Comme il l'a été explicité précédemment, le problème réside essentiellement dans les points suivants :

- le séparateur de champ pour le fichier "hosts(5)" est différent de celui qui pourrait nous servir pour faire le tri sur les champs de l'adresse IP,
- le nom d'une machine, si son domaine est spécifié, peut comporter un certain nombre de ".", lui-même séparateur de champs pour l'adresse IP (cf [8]),
- le tri doit être effectué en numérique uniquement sur les champs de l'adresse IP.

Il est clair, que la commande à utiliser pour faire ce tri, est la commande "sort(1)". Par contre, nous devrons faire en sorte que l'entrée du tri comporte cinq champs :

- le premier champ numérique de l'adresse IP,
- le second champ numérique de l'adresse IP,
- le troisième champ numérique de l'adresse IP,
- le quatrième champ numérique de l'adresse IP,
- le ou les noms de la machine.

Il suffira alors de faire un tri numérique sur les quatre premiers champs. Le résultat à afficher devra reformater de telle sorte qu'il obéisse au format du fichier "hosts(5)".

Nous aurons donc les étapes suivantes :

1. convertir le fichier "hosts(5)" de telle sorte que l'on dispose des cinq champs explicités précédemment,
2. effectuer le tri,
3. reformater le résultat du tri pour obéir au format usuel et l'afficher sur la sortie standard.

Or, un nom de machine peut utiliser le caractère "." afin d'identifier son domaine DNS. Par conséquent, il faudra utiliser un autre caractère pour séparer

12.2.3 Développement

Comme tout script UNIX, le but est de n'utiliser, dans la mesure du possible, aucun fichier temporaire. Comme il l'a été développé au paragraphe précédent, il est clair que les résultats s'enchainent les uns aux autres. Nous utiliserons donc les mécanismes de redirection d'entrée/sortie et des filtres.

Après les commentaires d'entête d'usage, la première chose à faire est de reformater le fichier "hosts(5)" pour l'*envoyer* à la commande de tri "sort(1)". Pour cela, nous utiliserons la commande "sed(1)". Celle-ci nous servira, dans un premier temps, à supprimer les éventuels commentaires du fichier (pour plus de détails, cf. 10.4.3). Nous aurons donc :

```
sed -e '/^#.*/d' -e 's/#.*$//' /etc/hosts
```

Dans un deuxième temps, la commande "sed(1)" va nous servir à reformater l'adresse IP, afin de substituer le caractère "." par le caractère ":". Sachant que le format d'une telle adresse est composé de quatre nombres séparés par ".", l'expression régulière suivante la décrit parfaitement :

```
[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*
```

Par conséquent, toute ligne du fichier "host(5)" peut être décrite par l'expression suivante :

```
\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\(.*$\)
```

Ce qui nous permet d'avoir les références suivantes pour l'expression régulière :

Référence	Description
\1	Premier champ numérique de l'adresse IP.
\2	Second champ numérique de l'adresse IP.
\3	Troisième champ numérique de l'adresse IP.
\4	Quatrième champ numérique de l'adresse IP.
\5	Noms de la machine.

Nous pourrons donc écrire la requête de substitution suivante :

```
s/\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\(.*$\)/\1:\2:\3:\4:\5/
```

Ce qui nous donne :

```
sed -e '/^#.*/d' -e 's/#.*$//' /etc/hosts     |\
  sed -e \
    's/\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\(.*$\)/\1:\2:\3:\4:\5/'
```

Il ne reste plus qu'à effectuer le tri sachant que :

- le tri doit s'effectuer par ordre numérique croissant sur les quatre premiers champs,
- chaque champ est séparé par le caractère ":".

Nous obtenons :

```
sed -e '/^#.*/d' -e 's/#.*$//' /etc/hosts     |\
```

```
sed -e \
  's/\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\(.*$\)/\1:\2:\3:\4:\5/' |\
sort -t: +n
```

Il faut maintenant reformater le résultat pour obtenir celui qui est conforme au fichier "`hosts(5)`". Nous allons ici utiliser lLa commande "`awk(1)`". Celle-ci recevra donc des lignes (ou enregistrements) séparés par le caractère "`:`" et devra effectuer le traitement suivant pour chacunes :

- afficher le premier champ,
- afficher le caractère ".",
- afficher le second champ,
- afficher le caractère ".",
- afficher le troisième champ,
- afficher le caractère ".",
- afficher le quatrième champ,
- afficher le cinquième ,
- insérer un retour à la ligne pour marquer la fin de l'enregistrement obtenu en résultat.

Nous avons donc :

```
awk '
    BEGIN { FS=":" }
    {
        printf ("%d.%d.%d.%d%s\n", ^$1, $2, $3, $4, $5)
    }
  '
```

12.2.4 Programme obtenu

```
#!/bin/sh
#
# Description : Tri du fichier /etc/hosts par adresse IP
#
# Appel du programme : sorthost
#
# Fichiers externes : Non
#
sed -e '/^#.*/d' -e 's/#.*$//' /etc/hosts    |\
  sed -e 's/\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\.\([0-9]*\)\(.*$\)/\1:\2:\3:\4:\5/' |\
  sort -t: +n |\
  awk '
    BEGIN { FS=":" }
    {
        printf ("%d.%d.%d.%d%s\n", ^$1, $2, $3, $4, $5)
    }
  '
```

12.3 Recherche des UID et GID disponibles

12.3.1 Étude des fonctionnalités

La procédure que nous voulons mettre en place doit permettre de créer de nouvelles entrées dans le fichier "`passwd(5)`". Pour rappel, ce fichier permet définir la liste des utilisateurs pouvant se connecter au système. Comme il l'a été précisé à la section 2.3.1, chaque utilisateur sous UNIX doit posséder un identifiant numérique unique et un numéro de groupe. Ce numéro doit être unique **par groupe et non pas par utilisateur**.

Par conséquent, en se fixant une valeur de départ, il faudra chercher dans le fichier "`passwd`" le dernier identifiant non attribué à partir de la valeur de départ. Le but de ce programme n'est donc pas de prendre le dernier numéro attribué et de l'incrémenter de 1 mais bien de déterminer le premier numéro disponible supérieur à la valeur de départ. Il faudra procéder de même pour pouvoir insérer une nouvelle entrée dans le fichier "`group(5)`". Il en résulte que nous allons introduire une option dans le lancement de notre procédure qui, en fonction de celle-ci, déterminera le dernier UID ou GID disponible. Nous aurons donc les options suivantes :

Option	Action
`-u`	Détermination du dernier UID disponible.
`-g`	Détermination du dernier GID disponible.

Pour pouvoir utiliser le résultat de cette procédure, nous afficherons la valeur adéquate sur la sortie standard. Ainsi, si notre procédure s'appelle "`searchid`", nous pourrons récuperer le résultat de la façon suivante :

```
variable=‘searchid -u‘
```
ou bien
```
variable=‘searchid -g‘
```

12.3.2 Méthode utilisée

La méthode utilisée est, de partir de cette valeur de base, stockée dans la variable d'environnement "`START_UID`" pour les identifiants d'utilisateur ou "`START_GID`" pour les identifiants de groupe, de parcourir le fichier adéquat et ainsi de localiser la valeur.

Si nous nous référons au manuel des fichiers "`passwd(5)`" et "`group(5)`", la valeur de l'UID ou du GID est le troisième champ, sachant que chaque champ est séparé par le caractère "`:`".

Nous allons donc :

1. extraire ce champ de tous les enregistrements,
2. trier cette série de nombre par ordre croissant,

3. n'en extraire que les valeurs supérieures ou égales à la borne inférieure (contenue dans la variable d'environnement "START_UID" ou "START_GID" en fonction de l'option spécifiée),
4. déterminer, dans cette série, le premier numéro disponible supérieur à la borne inférieure.

Par conséquent, nous utiliserons les commandes suivantes :

- "cut" pour l'extraction des champs,
- "sort" pour l'opération de tri,
- "awk" pour faire une sélection sur les valeurs,
- "awk" pour calculer le premier numéro disponible.

Notre procédure disposera des codes de retour suivants :

Retour	Description
0	Retour sans erreur.
valeur non nulle	Erreur d'exécution.

12.3.3 Développement

Après les commentaires d'entête d'usage, la première chose à faire est d'initialiser les variables d'environnement nécessaires au programme. Toutefois, sachant qu'elles peuvent être définies au niveau du shell, les valeurs spécifiées ici seront les valeurs par défaut. Pour cela, nous utiliserons la syntaxe

```
"VARIABLE=${VARIABLE:=valeur}"
```

(cf. section 3.2). Nous aurons donc :

```
TMP_DIR=${TMP_DIR:=/home/adm/tmp}
START_UID=${START_UID:=1000}
START_GID=${START_GID:=1000}
PASSWD=${PASSWD:=/etc/passwd}
GROUP=${GROUP:=/etc/group}
AWK=${AWK:=/usr/ucb/gawk}

export TMP_DIR START_UID START_GID PASSWD GROUP AWK
```

Remarque 12.1 :

Nous utilisons ici une extension de la commande "awk" : "gawk". Les fonctionnalités et les syntaxes vues à la section 11 sont identiques. Par contre, ses possibilités sont étendues et les options disponibles sur la ligne de commande sont plus importantes. Pour assurer l'indépendance du script, nous utiliserons une variable d'environnement "AWK" référençant l'exécutable "awk" à prendre en compte.

En cas d'erreur de syntaxe, il est de règle de préciser à l'utilisateur le format de la commande, ainsi que les options. Pour cela, nous allons créer une fonction qui affichera la syntaxe à utiliser sur la sortie d'erreur standard, canal à utiliser

pour tout affichage de messages d'erreur. Nous aurons donc :

```
_usage()
{
    echo "Usage: `basename $0` -u|-g" >&2
}
```

Maintenant, nous allons voir si le nombre d'arguments spécifiés sur la ligne de commandes est conforme. D'après le format explicité au paragraphe 12.3.1, ce script doit avoir un et un seul argument sur la ligne de commande qui est l'option spécifiant si la recherche doit s'effectuer pour les UID ou les GID. Par conséquent, la valeur de la variable "#" doit être égal à 1. Dans tous les autres cas, l'appel de cette procédure est incorrect. Nous aurons donc :

```
if [ $# -lt 1 ]; then
    _usage
    exit 1
fi
```

Nous devons maintenant :

- analyser l'argument passé sur la ligne de commande,
- mémoriser le fichier sur lequel les informations sont à prendre (stocker dans la variable locale "map"),
- mémoriser quelle sera la valeur de la borne inférieure (stockée dans la variable "start_id" à partir de la variable d'environnement adéquate,
- afficher un message d'erreur si l'option spécifiée n'est pas connue.

Nous utiliserons donc l'instruction "case" permettant de faire des branchements conditionnels à partir de la valeur contenue dans le seul et unique argument, présent dans la variable "1" (dont le contenu est obtenu par la séquence "$1"). Nous avons donc :

```
case $1 in
    -u)
        map=$PASSWD
        start_id=$START_STUDENT_UID
        ;;

    -g)
        map=$GROUP
        start_id=$START_PROJECT_GID
        ;;
    *)
        _usage
        exit 1
        ;;
esac
```

Sachant que les informations seront extraites à partir du fichier dont le nom est contenu dans la variable locale "map", avant toute opération de traitement de fichier, il est préférable de vérifier que celui-ci existe. Si c'est le cas, aucun

problème. Par contre, s'il n'existe pas, on peut considérer que nous sommes en train d'en créer un nouveau de toute pièce. Par conséquent, la première valeur à affecter est la borne inférieure. Nous avons donc :

```
if [ ! -f $map ]; then
    echo $start_id
    exit 0
fi
```

À ce point du programme,

- nous sommes sûr de la présence des informations dans les fichiers adéquats,
- nous connaissons la valeur de la borne inférieure,
- nous connaissons le type d'information à communiquer (UID ou GID).

Comme nous l'avons vu au paragraphe 12.3.2, la méthode utilisée pour calculer cette donnée repose sur :

1. l'extraction de tous les valeurs déjà affectées,
2. le tri de ces valeurs,
3. la sélection des valeurs déjà allouées supérieures ou égales à la borne inférieure,
4. le calcul de la première valeur disponible.

Pour extraire les données, il suffit d'utiliser la commande "`cut`" à partir des informations contenues dans le fichier dont le nom est stocké dans la valriable locale "`map`". Sachant que le champ UID du fichier "`passwd(5)`" et le champ GID du fichier "`group(5)`" est, dans les deux cas, le troisième champ et que le séparateur de champ est le caractère "`:`", il nous suffit d'écrire :

```
cut -f3 -d: $map
```

Le résultat est donc envoyé sur la sortie standard, résultat à traiter pour l'étape suivante.

Le tri de ces valeurs doit se faire dans l'ordre numérique. La commande "`sort`" est don,c utilisée avec l'option "`-n`". Ici, nous n'avons pas à spécifier de séparateur de champs, ni même de champs sur lesquels le tri est effectué. En effet, chaque enregistrement (c'est-à-dire chaque ligne arrivant sur l'entrée standard), n'est composée que d'un seul champ : le résultat de la commande "`cut`". Nous avons donc :

```
cut -f3 -d: $map |\
    sort -n -u
```

L'option "`-u`" de "`sort`" permet d'éliminer les éventuels doublons[1] Le résultat est donc envoyé sur la sortie standard, résultat à traiter pour l'étape suivante.

L'étape suivante consiste à ne sélectionner les informations dont la valeur est supérieure ou égale à la borne inférieure contenue dans la variable locale

1. Il existe des techniques d'administrations sous Unix affectant le même UID à deux utilisateurs différents dans un soucis d'exploitation. Nous ne rentrerons pas dans ces détails. Sachez uniquement que cette possibilité existe.

"start_id". Nous allons, pour cela, utiliser "awk". À partir de la ligne de commande, on initialise une variable interne à "awk", la variable "min_id", à partir de la valeur contenue dans la variable "start_id" du shell. Si le contenu du premier champ (voire même de la totalité de l'enregistrement) est supérieur ou égal à la variable "min_id", alors l'enregistrement est affiché sur la sortie standard. Nous obtenons donc :

```
cut -f3 -d: $map                                    |\
    sort -n -u                                      |\
    $AWK -v min_id=$start_id '$1 >= min_id { print $0}'
```

Comme pour les étapes précédentes, le résultat obtenu est dirigé sur la sortie standard, canal par défaut pour le filtre "awk".

Remarque 12.2 :

Comme il l'a été précisé précédemment, la variable d'environnement "AWK" permet de référencer l'exécutable "awk" souhaité.

La dernière étape consiste à prendre les données précédentes et de déterminer quelle est la première valeur disponible. Pour cela, nous allons utiliser à nouveau la commande "awk" (ou plus précisemment celle contenue dans la variable d'environnement "AWK"). Comme précédemment, les données arrivant sur l'entrée standard sont une suite de nombres, chacun sur une ligne distincte. Nous allons procéder de la façon suivante :

- On initialise une variable interne à "awk", la variable "min_id", à partir de la variable du shell "start_id".
- Avant tout traitement des enregistrements, on enregistre cette valeur dans une variable "awk", la variable "affected_id", qui nous permettra par la suite de comparer la valeur de l'enregistrement courrant à celle contenue dans cette variable.
- Chaque enregistrement, dont les valeurs sont ordonnées, est comparé à la valeur courrante de la variable "affected_id".
  - S'il y a égalité, la valeur contenue dans "affected_id" est donc déjà enregistrée dans le fichier de données. Il faut donc prendre une valeur supérieure, pour cela elle est incrémentée de "1".
  - Dans le cas, contraire, cette valeur est disponible. Par conséquent, elle reste inchangée. Tous les enregistrements suivants auront donc une valeur différente de celle contenue dans la variable "affected_id".
- En fonction de l'algorithme décrit précédemment, lorsque le dernier enregistrement est traité, nous avons deux cas de figure possibles :
  1. Une valeur intermédiaire a été localisée. Dans ce cas, la variable "awk", "affected_id", contient cette valeur.
  2. Aucune valeur intermédiaire n'a été localisée. Dans ce cas, la variable "affected_id" a été systématiquement incrémentée de "1". Elle contient donc la plus grande valeur enregistrée dans le fichier de données incrémentée de "1".

  Par conséquent, lorsque le dernier enregistrement est traité, la variable "affected_id" contient la valeur adéquate. Il suffit donc d'en afficher le contenu sur la sortie standard.

Nous obtenons donc :

```
cut -f3 -d: $map                                |\
    sort -n -u                                  |\
    $AWK -v min_id=$start_id '$1 >= min_id { print $0}' |\
    $AWK -v min_id=$start_id '
        BEGIN {
            affected_id=min_id
        }
        {
            if ( $1 == affected_id )
                affected_id ++
        }
        END {
            print affected_id
        }'
```

Sachant que nous voulons récupérer ce résultat dans une variable, il suffit de faire en sorte que l'évaluation de l'expression précédente serve à initialiser une variable. Or nous avons vu que le résultat de l'évaluation était dirigé sur la sortie standard. Grâce à la syntaxe examinée à la section 2.2, le résultat d'une commande affichée sur la sortie standard peut être mémorisée dans une variable de la façon suivante :

```
variable=‘expression‘
```

Par conséquent, si le résultat de l'expression précédente doit être mémorisée dans la variable *shell* "no_id", il suffit d'écrire :

```
no_id=‘ cut -f3 -d: $map                        |\
    sort -n -u                                  |\
    $AWK -v min_id=$start_id '$1 >= min_id { print $0}' |\
    $AWK -v min_id=$start_id '
        BEGIN {
            affected_id=min_id
        }
        {
            if ( $1 == affected_id )
                affected_id ++
        }
        END {
            print affected_id
        }’‘
```

Il ne reste plus qu'à afficher la valeur de la variable sur la sortie standard et terminer avec un code de sortie équivalent à un bon déroulement des opérations. Nous avons donc :

```
echo $no_id
exit 0
```

12.3.4 Programme obtenu

```
#!/bin/sh
#
#
#   SERVICE DES ADMINISTRATEURS:
#       Administration des utilisateurs
#       Recherche pour l'affectation d'un UID ou d'un GID
#
#   Programme: $DIR_USERS_BIN/searchid
#
#   Codes de retour:
#       OK          0
#       ERREUR      1
#       Autre       2,...
#
#   Exemples: searchid -u     Retourne le prochaine uid libre
#             searchid -g     Retourne le prochaine gid libre
#
#   Creation: S. Baudry
#
#   Modifications:
#

#-------------------------------------------------------------
# Initialisation
#
TMP_DIR=${TMP_DIR:=/home/adm/tmp}
START_UID=${START_UID:=1000}
START_GID=${START_GID:=1000}
PASSWD=${PASSWD:=/etc/passwd}
GROUP=${GROUP:=/etc/group}
AWK=${AWK:=/usr/ucb/gawk}

export TMP_DIR START_UID START_GID PASSWD GROUP AWK

#-------------------------------------------------------------
# Fonctions locales
#
_usage()
{
    echo "Usage: `basename $0` -u|-g" >&2
    exit 1
}

#-------------------------------------------------------------
# Analyse de la ligne de commande
#
#
```

```
# Teste le passage de parametre
#
if [ $# -lt 1 ]; then
    _usage
    exit 1
fi

case $1 in
    -u)
        map=$PASSWD
        start_id=$START_STUDENT_UID
        ;;

    -g)
        map=$GROUP
        start_id=$START_PROJECT_GID
        ;;
    *)
        _usage
        exit 1
        ;;
esac

#-------------------------------------------------------------
# Corps du programme
#

if [ ! -f $map ]; then
echo $start_id
    exit 0
fi

#
# Calcul du numero a creer
#

no_id=‘ cut -f3 -d: $map                    |\
    sort -n -u                              |\
    $AWK -v min_id=$start_id '$1 >= min_id { print $0}' |\
    $AWK -v min_id=$start_id '
        BEGIN {
            affected_id=min_id
        }
        {
            if ( $1 == affected_id )
                affected_id ++
        }
        END {
            print affected_id
        }'‘
```

```
#------------------------------------------------------------
# Fin du programme
#

echo $no_id
exit 0
```

12.4 Traduction de fichiers d'informations

12.4.1 Étude des fonctionnalités

Le but de cet exemple met en évidence comment, avec une *simple* procédure, il est possible de mettre en place des outils puissants de conversion.

Le but de cet exemple est de transformer la base complète des comptes utilisateurs et projets sur un système OpenVMS vers un système UNIX. Nous considérerons qu'éventuellement, une partie des ces comptes sont déjà existants sur la machine UNIX et, par conséquent, l'entrée déjà existante doit être prise en compte. En effet, si nous considérons que :

- tout élève de l'année "n" doit posséder un compte sur la machine UNIX,
- seulement une partie des élèves de l'année "$n+1$" doit posséder un compte sur la machine UNIX,
- tous les élèves de toutes les années possèdent un compte sur OpenVMS,
- le nom du compte OpenVMS fait au plus douze caractères,
- le nom du compte UNIX fait au plus huit caractères,
- le nom du compte OpenVMS est de la forme "*nom_initiale_prénom*" en majuscules,
- le nom du compte UNIX est de la forme "*nom*" en minuscules,
- le nom des groupes projets sur OpenVMS est de la forme "*projet*",
- le nom des groupes projets sur UNIX est de la forme "p*projet*".

Nous supposons que nous disposons d'un ensemble de fichiers donnant les caractéristiques des utilisateurs à créer. Dans le cas présent nous disposons des informations décrites au tableau 12.1.

Remarque 12.3 :

Le fichier "`list.txt`*" est facilement déduisible du fichier "*`lcluaf.txt`*". Afin de ne pas surchager cet exemple, nous considèrerons qu'un utilitaire en amont a généré le premier à partir du second.*

Remarque 12.4 :

Le fichier "`who.txt`*" est le résultat de la commande Open*VMS *"*`WHO`*" pour l'ensemble des comptes dont il faut une entrée sur* UNIX*.*

La création des bases utilisateur sur le système UNIX nécessite la création

Fichier	Description
`lcluaf.txt`	Contient la liste des noms des groupes de projets sur OpenVMS. Chaque ligne est composée de deux champs séparés par le caractère `":"` avec : – le nom du groupe projet sur OpenVMS, – la liste des noms de compte OpenVMS appartenant à ce projet, chaque nom étant séparé par le caractère `","`.
`list.txt`	La liste de tous les noms de compte OpenVMS devant posséder un accès sur la machine UNIX.
`who.txt`	Toutes les informations sur les utilisateurs OpenVMS. Nous aurons le prénom, le nom et la classe en toute lettre. Le format de ce fichier est le suivant : – le nom du compte OpenVMS, – un nombre quelconque d'espaces, – le caractère `"!"`, – un espace, – le prénom et le nom – un espace, – le caractère `"-"`, – un espace, – la classe
`passwd`	La base des utilisateurs UNIX actuels.

TAB. 12.1 – *Description des fichiers en entrée ou en sortie pour la converions des utilisateurs Open*VMS *vers* UNIX.

de plusieurs fichiers :

- un fichier obéissant au format du fichier "`passwd(5)`" donnant la liste des nouveaux utilisateurs du système,
- un fichier obéissant au format du fichier "`group(5)`" contenant la liste des nouveaux groupes associés chacun à un projet, sur chaque ligne, on trouvera aussi les membres de ce projets (les noms seront les noms des comptes utilisateur UNIX),
- un fichier obéissant au format du fichier "`shadow(5)`" associé à la liste des nouveaux utilisateurs afin d'assurer la gestion du vieillissement des mots de passe,
- un fichier permettant de faire la liaison entre le nom du répertoire pour le compte personnel et le serveur de disque adéquat (ce fichier sera utilisé par le processus "`autofs(8)`" UNIX),
- un fichier permettant de faire la liaison entre le nom du répertoire associé au projet et le serveur de disque adéquat (ce fichier sera utilisé par le processus "`autofs(8)`" UNIX),
- un fichier de compte-rendu sur les opérations effectuées avec :
  - le nom du compte UNIX,
  - le nom du compte OpenVMS,
  - le numéro d'utilisateur UNIX (UID),
  - le projet auquel appartient cet utilisateur.

Nous considèrerons ici que l'ensemble des fichiers en entrée se trouvent dans un répertoire donné et que les fichiers résultats se trouveront dans un autre répertoire.

Remarque 12.5 :

L'ancien fichier des mots de passe contient la classe de l'année passée. Par conséquent, lors de la reprise des anciennes entrées, il faudra éliminer cette information pour la remplacer par celle correspondant à l'année en cours.

12.4.2 Méthode utilisée

Comme dans tout script de cette importance, deux autres fichiers y sont systématiquement attachés :

- un fichier initialisant l'ensemble des variables d'environnement, dont le nom, par convention, se termine par "`.define`",
- un fichier contenant l'ensemble des fonctions shell qui seront utilisées, dont le nom, par convention, se termine par "`.functions`".

À ce programme est attaché un autre script vérifiant la présence de tous les fichiers nécessaires, c'est-à-dire :

- l'ensemble des fichiers en entrée,
- les fichiers en sortie,
- l'ensemble des scripts ou programmes externes nécessaires pour générer les informations.

De même, la convention utilisée dans l'écriture des scripts veut que **seulement certaines** variables d'environnements permettant de générer l'ensemble de l'environnement, donc, dans le cas présent, de charger le fichier des définitions. De même, il convient, en général, de définir une variable d'environnement donnant le répertoire de base, et, ensuite, de s'en servir pour initialiser les valeurs de toutes les autres.

Si, par hasard, le script nécessite des fichiers exécutables binaires, c'est-à-dire de programmes générés à partir d'un programme source, il convient de pouvoir le regénérer à partir d'un fichier "`Makefile`[2]".

La première étape de développement consiste à générer un fichier temporaire à partir des anciennes entrées du fichier des mots de passe, en supprimant l'information de la classe. Celle-ci occupe les cinq derniers caractères du champ "*GCOS*[3]" du fichier "`passwd(5)`". C'est ce fichier temporaire qui sera utilisé dans toute la suite. Ce fichier sera référencé grâce à la variable d'environnement "`PASSWD_REF`".

La seconde étape consiste à générer l'équivalent à partir du fichier "`who.txt`" (référencé grâce à la variable "`WHO`"). Le résultat sera stocké dans un fichier temporaire dont le nom sera contenu dans la variable d'environnement "`WHO_REF`". C'est celui-ci qui sera utilisé par la suite.

Pour chaque nouvel utilisateur à créer, connu grâce à la liste contenue dans le fichier "`list.txt`", il suffit exécuter les étapes suivantes :

1. déterminer les nouvelles entrées à créer en fonction des anciens utilisateurs déjà enregistrés et la liste complète des comptes à créer,
2. pour chaque utilisateur à créer (nouveaux comme anciens), générer toutes les informations nécessaires et éviter les éventuels doublons (en effet, la définition des comptes utilisateurs sous OpenVms se fait sur 12 caractères alors que sous Unix seulement 8 sont significatifs),
3. supprimer les anciens répertoires, associés aux comptes à supprimer (on supposera qu'une sauvegarde aura été faite au préalable),
4. créer les entrées nécessaires pour la gestion des comptes "*projet*" grâce au fichier "`group(5)`",
5. créer l'ensemble des répertoires nécessaires (utilisateurs et projets) en y affectant les droits d'accès adéquat.

12.4.3 Développement

Nous ne détaillerons ici que le corps du script principal. En effet, les fichiers associés à la définition des variables d'environnement et à la vérification de la présence des données, ne présentent aucune difficulté majeure. Le fichier contenant les différentes fonctions reprennent les exemples vus précédemment.

Pour chaque fichier de données en entrée (décrits au tableau 12.1) et en sortie, pour les fichiers temporaires et les exécutables ou scripts externes, nous définirons les variables d'environnements suivantes :

2. cf. commande "`make(1)`".
3. Cinquième champ du fichier.

Fichier	Variable	Description
Fichiers en entrée		
`lcluaf.txt`	`LCLUAF`	Contient la liste des noms des groupes de projets sur OpenVMS.
`list.txt`	`LIST`	La liste de tous les noms de compte OpenVMS devant posséder un accès sur la machine UNIX.
`who.txt`	`WHO`	Toutes les informations sur les utilisateurs OpenVMS.
`passwd`	`PASSWD`	La base des utilisateurs UNIX actuels.
Fichiers temporaires		
`passwd.`*num*	`PASSWD_REF`	Référence utilisée pour la définition des utilisateurs actuels.
`who.`*num*	`WHO_REF`	Référence utilisée pour la correspondance entre les "*username*s" OpenVMS et le prénom/nom de l'utilisateur.
Fichiers en sortie		
`passwd.new`	`PASSWD_NEW`	Résultat obtenu pour les nouvelles entrées utilisateur. Ce fichier viendra complèter ou remplacer les définitions des utilisateurs sur le système. Il obéira à la syntaxe définie dans "`passwd(5)`".
`group.new`	`GROUP_NEW`	Résultat obtenu pour les nouvelles entrées utilisateur. Ce fichier viendra complèter ou remplacer les définitions des utilisateurs sur le système. Il obéira à la syntaxe définie dans "`group(5)`".
`shadow.new`	`SHADOW_NEW`	Résultat obtenu pour les nouvelles entrées utilisateur. Ce fichier viendra complèter ou remplacer les définitions des utilisateurs sur le système. Il obéira à la syntaxe définie dans "`shadow(5)`".

Suite page suivante ···

Suite de la page précédente ···

Fichier	Variable	Description
`users.infos`	`USERS_INFO_FILE`	Résultat obtenu pour les nouvelles entrées utilisateur. Ce fichier viendra complèter ou remplacer les définitions des utilisateurs sur le système. Il obéira au format suivant : – Chaque ligne représentera un enregistrement mémorisant les caractéristiques des utilisateurs créés. – Chaque champ sera délimité par le caractère `":"`. – Le premier champ correspond au "*logname*" UNIX. – Le second champ est le "*username*" OpenVMS. – Le troisième champ contient le mot de passe non crypté par défaut. – Le quatrième champ est l'"UID" UNIX. – Le cinquième champ est le nom du groupe associé au projet.
`auto.projects`	`AUTO_PROJECTS`	Résultat obtenu pour les nouvelles entrées utilisateur. Ce fichier viendra complèter ou remplacer les définitions des utilisateurs sur le système. Il obéira à la syntaxe définie dans "`autofs(8)`" ou "`automount(8)`".
`auto.students`	`AUTO_USERS`	Résultat obtenu pour les nouvelles entrées utilisateur. Ce fichier viendra complèter ou remplacer les définitions des utilisateurs sur le système. Il obéira à la syntaxe définie dans "`autofs(8)`" ou "`automount(8)`".
Scripts ou exécutables externes		
`searchid`	`SEARCHID`	Script permettant de localiser le dernier "UID" disponible (cf. section 12.3).
`buildpasswd`	`BUILDPASSWD`	Exécutable permettant de générer un mot de passe crypté à partir d'une chaine de caractères passée en argument.
`/bin/echo`	`ECHO`	Exécutable "`echo`" utilisé.
`/usr/ucb/gawk`	`AWK`	Exécutable "`awk`" utilisé.

La localisation de ces fichiers sera référencée par une variable d'environnement :

– "`IN_DIR`" pour les fichiers en entrée,
– "`OUT_DIR`" pour les fichiers en sortie,
– "`TMP_DIR`" pour les fichiers temporaires,
– "`BIN_DIR`" pour certains scripts ou exécutables externes.

Nous allons donc examiner les méthodes de développement pour les différentes parties :

- Détermination des entrées de référence par rapport aux informations déjà existante.
- Extraction des anciennes entrées et mise-à-jour afin de correspondre au profil courrant.
- Création des nouvelles entrées "*utilisateur*", c'est-à-dire celles dont le compte n'existait pas auparavant (contrairement aux anciennes qu'il a fallu modifier).
- Suppression de anciens répertoires qui n'ont plus lieu d'être. Par exemple, si une entrée utilisateur est supprimée, les répertoires associés doivent être effacés. Nous supposerons toutefois qu'une sauvegarde des données aura été faite au préalable.
- Création des entrées nécessaires pour la gestion des comptes "*projet*" grâce au fichier "`group(5)`".
- Création l'ensemble des répertoires nécessaires. Ces répertoires correspondront aux répertoires personnels et aux répertoires "*projet*". Il est clair que les droits d'accès devront être, par la même occasion, positionnés de telle sorte que les utilisateurs puissent travailler correctement.

12.4.3.1 Détermination des entrées de référence

Les entrées de références correspondent aux anciennes informations, modifiées de telle sorte qu'elles puissent être exploitable directement. En effet, la convention adoptée par exemple, pour le champ "*informations*" ou "*GCOS*" du fichier "`passwd(5)`", est la suivante :

- le prénom
- le nom en majuscules,
- le caractère "`-`" délimité par un espace de chaque côté,
- la classe.

Par conséquent, d'une année sur l'autre, la classe peut changer. L'ancienne information n'est donc plus valable dans sa globalité, mais par contre la totalité de ce champ sans l'information "*classe*" est valide et permet, en plus, de localiser un enregistrement de façon unique.

Il nous faut donc prendre l'ancien fichier "`passwd(5)`" et enlever l'information "*classe*". Ceci est effectué en supprimant les cinq derniers caractères du cinquième champ (le champ "*GCOS*" du fichier "`passwd(5)`". Le résultat nous servira donc de référence pour tout traitement exploitant les anciennes entrées du fichier de définition des utilisateurs, car nous serons sûr que toutes les renseignements présents seront valides. Nous obtenons donc la requête suivante :

```
$AWK '
    BEGIN { FS=":"}
    {
        field = substr ($5, 1, length($5) - 5)
        printf ("%s:%s:%s:%s:%s:%s:%s\n",
```

```
                $1, $2, $3, $4, field, $6, $7)
    }
' $PASSWD > $PASSWD_REF
```

Il en va de même pour le fichier associant "*username* OpenVMS" avec le prénom, le nom et la classe de la personne. Le format de ce fichier est :

- le *username* OpenVMS,
- le caractère "-" délimité par un espace de chaque côté,
- la classe avec, éventuellement, l'option.

Par conséquent, nous ne connaissons par la longueur de la chaine représentant la classe. Par contre, l'information "*classe*", y compris le délimiteur, peut être représenté par l'expression régulière suivante :

```
␣-␣[0-9][ABC].*$
```

L'opération à effectuer est donc de supprimer tout ce qui correspond à cette expression, ou bien remplacer tout le texte associé à cette expression par "*rien*".La commande la plus appropriée est "`sed(1)`". En effet, la notion de champ n'intervient pas, il suffit d'effectuer une substitution sur chaque ligne. La commande associée est donc :

```
sed -e 's/ - [0-9][ABC].*$//' $WHO > $WHO_REF
```

12.4.3.2 Extraction des anciennes entrées et mise-à-jour

Les anciennes entrées sont localisables grâce à l'intersection de l'ensemble des anciens utilisateurs et celui des utilisateurs à créer. L'opération se schématise comme suit.

Soit $\mathcal{U}_{\text{UNIX}}$ = utilisateur UNIX.

$\mathcal{U}_{\text{OpenVMS}}$ = utilisateur OpenVMS.

$\mathcal{E}_{\text{OpenVMS}}$ = liste des utilisateurs OpenVMS devant posséder un compte sous UNIX.

$$\begin{aligned}
\mathcal{E}_{ancien} &= \{\mathcal{U}_{\text{UNIX}} \text{ déjà existants}\} \\
\mathcal{E}_{nouveau} &= \left\{\mathcal{U}_{\text{UNIX}} \; tq. \; \exists\, \mathcal{U}_{\text{OpenVMS}} \in \mathcal{E}_{\text{OpenVMS}}\right\} \\
\mathcal{E}_1 &= \mathcal{E}_{ancien} \cap \mathcal{E}_{nouveau}
\end{aligned}$$

Or cette dernière liste ne contient que les "*username*s" OpenVMS. Par conséquent, à partir du "*username*", nous allons extraire son prénom et son nom. Grâce à cette information, nous pourrons rechercher dans "`PASSWD_REF`", en fonction du contenu du cinquième champ, la présence éventuelle de l'ancienne définition.

Nous allons donc faire une boucle de lecture sur chaque enregistrement du fichier "`LIST`", la valeur sera stockée dans la variable locale "`username`". Nous

obtenons :

```
cat $LIST |\
while
    read username
do
...
done
```

L'extraction du prénom et du nom se fait grâce :

- une recherche en fonction du "*username*" dans le fichier "`WHO_REF`" avec la commande "`grep`",
- un extraction du second champ, avec la commande "`cut`" utilisant comme délimiteur le caractère "`!`",
- la suppression des deux premiers espaces, donc des deux premiers caractères, avec la commande "`cut`".

Pour obtenir le "*login*" UNIX, il suffira d'utiliser "`awk`". La condition du sélecteur du programme "`awk`" sera l'identité entre le cinquième champ du fichier "`PASSWD_REF`" et le nom obtenu grâce à la précédente requête. Nous aurons donc :

```
name=‘grep $username $WHO_REF  | cut -d! -f2 | cut -c2-‘
is_user=‘$AWK -v name="$name" ’
    BEGIN { FS=":" }
    $5 == name { print $0 }
’ $PASSWD_REF‘
```

Par conséquent, si la variable "`is_user`" est vide, le bloc action associé au sélecteur "`$5 == name`" n'a jamais été exécuté. Il n'y a donc aucun utilisateur actuel qui satisfait ce critère : **cet utilisateur n'est pas déjà défini** . Par contre, si cette variable est non vide, **l'utilisateur courrant est déjà enregistré**. Dans ce cas, il ne restera plus qu'à générer les informations adéquates en fonction des deux informations clef : le "*username*" OpenVMS et le "*logname*" UNIX de l'utilisateur en train d'être traité.

Pour plus d'information sur les fichiers concernés et leurs formats, reportez-vous aux pages de manuels adéquates. Pour le détail concernant la génération des informations, reportez-vous à la section 12.4.4.4.

Remarque 12.6 :

La variable "`is_user`" contiendra alors l'ancienne définition associée au compte UNIX *(extraite de "`PASSWD_REF`") pour cet utilisateur.*

12.4.3.3 Création des nouvelles entrées utilisateur

Tout comme les anciennes entrées, les nouvelles sont détectées en excluant l'ensemble des utilisateurs existants à celui contenant l'ensemble des utilisateurs

à créer. L'opération se schématise comme suit.

Soit $\mathcal{U}_{\text{UNIX}}$ = utilisateur UNIX.

$\mathcal{U}_{\text{OpenVMS}}$ = utilisateur OpenVMS.

$\mathcal{E}_{\text{OpenVMS}}$ = liste des utilisateurs OpenVMS devant posséder un compte sous UNIX.

$$\mathcal{E}_{ancien} = \{\mathcal{U}_{\text{UNIX}} \text{ déjà existants}\}$$
$$\mathcal{E}_{nouveau} = \left\{\mathcal{U}_{\text{UNIX}} \; tq. \; \exists\, \mathcal{U}_{\text{OpenVMS}} \in \mathcal{E}_{\text{OpenVMS}}\right\}$$
$$\mathcal{E}_2 = \mathcal{E}_{nouveau} - (\mathcal{E}_{ancien} \cap \mathcal{E}_{nouveau})$$

Le principe adopté repose sur le même principe que celui décrit à la section adv-programming-ex3-devlext. À partir de la liste des nouveaux utilisateurs, seuls ceux qui disposent d'une entrée dans "`USERS_INFO_FILE`" sont traités, c'est-à-dire ceux dont le second champ de ce fichier est égal à l'entrée traitée du fichier "`LIST`". Nous utiliserons "`awk(1)`". La condition du sélecteur du programme associé sera l'identité entre le second champ de "`USERS_INFO_FILE`" et le contenu de la variable "`username`". Le résultat de "`awk(1)`" sera stocké dans la variable "`is_defined`". Si cette variable est non vide, il faudra passé à l'enregistrement suivant. Sinon, le traitement devra se poursuivre. Nous aurons donc :

```
cat $LIST |\
while
    read username
do
    is_defined=‘$AWK -v username="$username" -F: ’
        $2 == username { print $0 }
        ’ $USERS_INFO_FILE ‘
    [ "$is_defined" != "" ] && continue
...
done
```

Le "*logname*" UNIX obéira au format suivant :

- le "*logname*" correspond au nom de l'utilisateur,
- s'il dépasse huits caractères, il sera tronqué à huit,
- si ce nom est déjà attribué, il sera demandé manuellement à l'administrateur d'en attribuer un.

Il ne restera plus qu'à générer les informations complémentaires. Nous obtenons :

```
login=‘echo $username | tr ’[A-Z]’ ’[a-z]’  | cut -d_ -f1‘
new_who=‘grep $username $WHO | cut -d! -f2 | cut -c2-‘

login=‘echo $login | cut -c-8‘
```

```
status=0
while
    is_allocated=`grep "^${login}:" $PASSWD_NEW`
    [ "$is_allocated" != "" ]
do
    echo "" >&2
    echo "`basename $0`: login $login deja alloue pour $username" >&2
    $ECHO "`basename $0`: Quel login : \c" >&2
    read login < /dev/tty
    status=1
done
[ $status -eq 1 ] && $ECHO "Creation des nouvelles entrees dans passwd : \c"

project="p`grep $username $LCLUAF | cut -d: -f1 | tr '[A-Z]' '[a-z]'`"

new_passwd=`echo $login | cut -c-4``date +%M%S`
crypt_passwd="`$BUILDPASSWD $new_passwd`"
uid=`$SEARCHID -u`
gid=$STUDENT_GID
student_home=${HOME_STUDENT}/$login

output="${login}:${crypt_passwd}:${uid}:${gid}"
echo "${output}:${new_who}:${student_home}:${LOGIN_SHELL}" >> $PASSWD_NEW
echo "${login}  ${FS_SERVER}:${FS_STUDENTS}/${login}" >> $AUTO_USERS
output="${login}:${crypt_passwd}:0:${PASSWD_WILL_CHANGE}"
echo "${output}:${PASSWD_EXPIRATION_TIME}:${PASSWD_WILL_EXPIRE}::" \
    >> $SHADOW_NEW
echo "${login}:${username}:${new_passwd}:${uid}:${project}" >> $USERS_INFO_FILE
```

12.4.3.4 Suppression des répertoires inutiles

La détection des anciens répertoires se fait à partir de "`PASSWD_REF`". Il suffira de vérifier que l'entrée courrante n'a pas été conservée grâce à "`USERS_INFO_FILE`". En effet, s'il existe un enregistrement dans ce fichier avec le même "*logname*", le répertoire doit être conservé. Dans le cas contraire, il doit être supprimé.

Par conséquent, nous allons faire une boucle de lecture sur chaque enregistrement du fichier "`PASSWD_REF`", la valeur sera stockée dans la variable locale "`line`". Pour chacun d'entre eux,

- nous extrairons le "*logname*" (premier champ) grâce à la commande "`cut(1)`", le séparateur étant le caractère "`:`",
- nous vérifierons la présence dans "`USERS_INFO_FILE`" avec la commande "`awk(1)`" en cherchant une équivalence entre le premier champ et la valeur du "*logname*",
- si une occurence est trouvée, il suffira d'effacer le répertoire avec tout son contenu grâce à la commande "`rm(1)`".

Nous obtenons :

```
cat $PASSWD_REF |\
while
    read line
do
    old_login=‘echo $line | cut -d: -f1‘
    is_present=‘$AWK -F: -v old_login="$old_login" ’
        $1 == old_login { print $1 }
        ’ $USERS_INFO_FILE‘
    if [ "$is_present" = "" ]; then
        [ -d $FS_STUDENTS/$old_login ] && \
            rm -rf $FS_STUDENTS/$old_login 2>&/dev/null
    fi
done
```

12.4.3.5 Création des nouvelles entrées "*projet*"

La création des nouvelles entrées "*projet*" se fait grâce aux informations contenues dans "LCLUAF". Comme pour les étapes précédentes, nous allons effectuer une boucle de lecture sur ce fichier. Ce qui donne :

```
cat $LCLUAF |\
while
    read line
do
...
done
```

Pour rappel (cf. tableau 12.1), le format utilisé ici est :

- chaque champ est séparé par le caractère ":",
- le premier champ contient le nom du projet sur OpenVms (donc en majuscules),
- le second champ contient la liste des "*username*s" OpenVms membres de ce projet séparés par ",".

La transformation du nom de projet OpenVms vers le nom Unix sera effectué grâce à la combinaison des commandes "echo(1)", "cut(1)" et "tr(1)". L'affectation du "GID" se fera avec "SEARCHID" (cf. exemple 12.3). Nous avons :

```
vms_project="‘echo $line | cut -d: -f1 | tr ’[A-Z]’ ’[a-z]’‘"
unix_project="p${vms_project}"
project_gid=‘$SEARCHID -g‘
$ECHO "${unix_project}:*:${project_gid}:\c" >> $GROUP_NEW
```

Remarque 12.7 :

La terminaison par "\c" de la commande "ECHO" nous permettra de compléter la ligne qui, pour l'instant, n'est pas encore complète. En effet, il manque encore les membres de ce groupe, c'est-à-dire la liste des "logname" Unix adéquats.

Une liste, sous UNIX est composée chaines de caractères séparées par un ou plusieurs espaces, il faudra donc substituer "," par un espace. Cette opération sera réalisée par la commande "sed(1)". Il faudra toutefois, extraire au préalable l'information de l'enregistrement courrant avec la commande "cut(1)". La liste des membres pour le projet courrant sera stocké dans la variable locale "members". Elle contient les "*username*s" OpenVMS associés. Il ne reste plus alors qu'à rechercher le "*logname*" UNIX correspondant. Pour cela, il suffit de faire une boucle sur chaque membre de la liste contenue dans "members", d'extraire l'enregistrement correspondant dans "USERS_INFO_FILE" et d'en afficher que le premier champ. Ceci sera effectué grâce à la commande "awk". Le résultat sera mémorisé dans la variable locale "group_members".

Le format de "group(5)" demande une liste de "*logname*s" séparés par ",". Par commodité, le programme "awk(1)" affichera donc systématiquement le "*logname*" suivi de ",". Il ne restera donc qu'à supprimer la dernière virgule. Ce critère correspond à l'expression régulière ",$". Il suffira d'appliquer une substition avec la commande "sed(1)" sur le contenu de la variable "group_members" et lui réaffecter le résultat. Il ne restera plus qu'à compléter l'entrée dans "GROUP" et générer l'entrée associée à ce projet dans "AUTO_PROJECTS".

Nous obtenons donc :

```
members="`echo $line | cut -d: -f2 | sed -e 's/,/ /g'`"
group_members=`for this_member in $members
    do
        awk -F: -v  this_member=$this_member '
            $2 == this_member { printf ("%s,",$1) }
            ' $USERS_INFO_FILE
    done`
echo "$group_members" | sed -e 's/,$//' >> $GROUP_NEW
echo "${unix_project}   ${FS_SERVER}:${FS_PROJECTS}/${unix_project}" \
    >> $AUTO_PROJECTS
```

12.4.3.6 Création des répertoires utilisateurs et "*projet*"

Que ce soit pour les répertoires utilisateurs ou "*projet*", la méthode reste la même. Nous partons du fichier contenant les descriptions des répertoires sur le serveur (fichiers "AUTO_USERS" et "AUTO_PROJECTS"), nous en extrayons les informations, c'est-à-dire le répertoire à créer éventuellement, et nous y appliquons les opérations suivantes :

- détermination du propriétaire et du groupe,
- création du répertoire s'il n'existe pas,
- copie de certains fichiers nécessaires à l'environnement utilisateur,
- affectation des droits d'accès.

Par convention, le nom d'un répertoire utilisateur ou "*projet*" est justement celui affecté à cet utilisateur ou ce projet. Par conséquent, il suffit d'utiliser la commande "basename[4]" avec comme argument, le nom du répertoire. L'"UID"

4. La commande "basename(1)" permet d'extraire le nom d'un fichier en fonction du chemin absolu ou relatif passé en argument.

correspond au troisième champ du fichier "PASSWD_REF", sachant que le premier champ doit ête identique à l'utilisateur traité (le nom est connu grâce à la commande "basename". Cette opération est donc effectuée de la façon suivante :

```
owner=‘basename $directory‘
uid=‘awk -F: -v owner=$owner ’
    $1 == owner { print $3 }
    ’ $PASSWD_NEW‘
```

où "directory" correspond au répertoire en cours de traitement. Il ne restera plus qu'à copier les fichiers de profil nécessaires et mettre à jour les droits d'accès.

La technique est rigoureusement identique pour les répertoires associés aux projets. Le fichier permettant de connaître les "GIDs" est "GROUP_REF".

Nous obtenons donc, pour cette partie, les instructions suivantes :

```
cat $AUTO_USERS | cut -d: -f2 |\
while
    read directory
do
    owner=‘basename $directory‘
    uid=‘awk -F: -v owner=$owner ’
        $1 == owner { print $3 }
        ’ $PASSWD_NEW‘
    if [ ! -d $directory ]; then
        mkdir -p $directory
        cp $HOME_PROTOTYPE/.[a-z]* $HOME_PROTOTYPE/.[A-Z]*      \
            $HOME_PROTOTYPE/* $FS_STUDENTS/$newname 2>/dev/null
        chown -R ${uid}.${STUDENT_GID} $directory
    fi
done

cat $AUTO_PROJECTS | cut -d: -f2 |\
while
    read directory
do
    group=‘basename $directory‘
    gid=‘awk -F: -v group=$group ’
        $1 == group { print $3 }
        ’ $GROUP_NEW‘
    if [ ! -d $directory ]; then
        mkdir -p $directory
        chown -R root.${gid} $directory
    fi
done
```

12.4.4 Programmes obtenus

Comme il l'a été spécifié précédemment, nous disposons ici de quatre fichiers pour remplir les fonctionnalités :

Nom du fichier		Description
`mkpasswd.define`	section 12.4.4.1, p. 173	Définition de l'ensemble des variables d'environnement nécessaire au programme. Les valeurs définies dans le script *principal* seront reprises ou bien seront initialisées si aucune affectation n'a été faite au préalable.
`mkpasswd.functions`	section 12.4.4.2, p. 176	Définition de l'ensemble des fonctions nécessaires au programme. Comme il l'a été précisé à la section 5.2, les fonctions sont l'équivalent de *macros* d'un langage de programmation. Nous définirons ici trois fonctions : – `"_waiting_chars"` permet de faire une *animation* pendant l'exécution de certaines étapes, – `"_ask"` gère la réponse des questions de type "*Oui*/*Non*", – les intructions à exécuter lors d'une interruption.
`mkpasswd.check`	section 12.4.4.3, p. 177	Vérification de l'ensemble des fichiers en entrée et de la présence des scripts externes. De plus nous faisons appel à un exécutable `"buildpasswd"` qui sera éventuellement reconstruit à partir de ses sources.
`mkpasswd`	section 12.4.4.4, p. 179	Le script à proprement parlé.

12.4.4.1 Fichier `"mkpasswd.define"`

```
#!/bin/sh
#
#
#   SERVICE DES ADMINISTRATEURS:
#       Traduction des bases OpenVMS vers Unix.
#       Definition des parametres
#
#   Fichier: $BIN_DIR/mkpasswd.define
#
#   Creation: S. Baudry
#
#   Modifications:
#
#-------------------------------------------------------------
```

```
#
# Repertoires
#
#-------------------------------------------------------------

MKPASSD_DIR=${MKPASSD_DIR:=/home/adm/users/convert}
export MKPASSD_DIR

IN_DIR=${IN_DIR:=$MKPASSD_DIR/in}
OUT_DIR=${OUT_DIR:=$MKPASSD_DIR/out}
TMP_DIR=${TMP_DIR:=$MKPASSD_DIR/tmp}
BIN_DIR=${BIN_DIR:=$MKPASSD_DIR/bin}
export IN_DIR OUT_DIR TMP_DIR BIN_DIR

#-------------------------------------------------------------
#
# Scripts/executables externes par defaut
#
#-------------------------------------------------------------

SEARCHID=$BIN_DIR/searchid
BUILDPASSWD=$BIN_DIR/buildpasswd

export SEARCHID BUILDPASSWD

ECHO=/bin/echo
AWK=/usr/ucb/gawk

export ECHO AWK


#-------------------------------------------------------------
#
# Fichiers en entree
#
#-------------------------------------------------------------

LCLUAF=$IN_DIR/lcluaf.txt
LIST=$IN_DIR/list.txt
WHO=$IN_DIR/who.txt
PASSWD=$IN_DIR/passwd

export LCLUAF LIST WHO PASSWD

#-------------------------------------------------------------
#
# Fichiers temporaires
#
#-------------------------------------------------------------
```

```
PASSWD_REF=$TMP_DIR/passwd.$$
WHO_REF=$TMP_DIR/who.$$

export PASSWD_REF WHO_REF

#-------------------------------------------------------------
#
# Fichiers en sortie
#
#-------------------------------------------------------------

PASSWD_NEW=$OUT_DIR/passwd.new
GROUP_NEW=$OUT_DIR/group.new
SHADOW_NEW=$OUT_DIR/shadow.new
USERS_INFO_FILE=$OUT_DIR/users.infos
AUTO_PROJECTS=$OUT_DIR/auto.projects
AUTO_USERS=$OUT_DIR/auto.students

export PASSWD_NEW GROUP_NEW SHADOW_NEW USERS_INFO_FILE
export AUTO_PROJECTS AUTO_USERS

#-------------------------------------------------------------
#
# Parametres
#
#-------------------------------------------------------------

PASSWD_EXPIRATION_TIME=60
PASSWD_WILL_EXPIRE=7
PASSWD_WILL_CHANGE=‘expr $PASSWD_EXPIRATION_TIME - $PASSWD_WILL_EXPIRE‘
export PASSWD_EXPIRATION_TIME PASSWD_WILL_EXPIRE PASSWD_WILL_CHANGE

START_PROJECT_GID=2000
STUDENT_GID=1001
START_STUDENT_UID=2000
export START_PROJECT_GID STUDENT_GID START_STUDENT_UID

HOME_STUDENT=/home/students
HOME_PROJECT=/home/projects
HOME_PROTOTYPE=/home/adm/users/convert/etc/prototypes
FS_SERVER=ampere.esme.fr
FS_STUDENTS=/export/home/disk3/students
FS_PROJECTS=/export/home/disk3/projects
export HOME_STUDENT HOME_PROTOTYPE FS_SERVER FS_STUDENTS FS_PROJECTS

PROTO_STUDENT_QUOTA="-p nobody"
PROTO_PROJECT_QUOTA="-p nogroup"
export PROTO_STUDENT_QUOTA PROTO_PROJECT_QUOTA

LOGIN_SHELL=/bin/csh
```

```
export LOGIN_SHELL
```

12.4.4.2 Fichier "mkpasswd.functions"

```
#!/bin/sh
#
#
#   SERVICE DES ADMINISTRATEURS:
#       Traduction des bases OpenVMS vers Unix.
#       Fonctions internes
#
#   Fichier: $BIN_DIR/mkpasswd.functions
#
#   Creation: S. Baudry
#
#   Modifications:
#
#-------------------------------------------------------------

_waiting_chars()
{
    tmp_index=`expr $1 % 8`
    case $tmp_index in
        0) $ECHO "-\b\c";;
        1) $ECHO "\\ \b\b\c";;
        2) $ECHO "|\b\c";;
        3) $ECHO "/\b\c";;
        4) $ECHO "-\b\c";;
        5) $ECHO "\\ \b\b\c";;
        6) $ECHO "|\b\c";;
        7) $ECHO "/\b\c";;
    esac
}

#-------------------------------------------------------------

_ask ()
{
    while
        case "$2" in
            y|Y) $ECHO "$1 ([y]/n) : \c" >&2 ;;
            n|N) $ECHO "$1 (y/[n]) : \c" >&2 ;;
        esac
        read answer
        [ "$answer" = "" ] && answer=$2
        answer=`echo $answer | tr '[A-Z]' '[a-z]'`
        [ "$answer" != "y" -a "$answer" != "n" ]
    do
        echo "Invalid answer, check validity." >&2
```

```
    done
    echo $answer
}

#-------------------------------------------------------------

_stop_exec ()
{
    echo "Arret en cours ..."
    [ -f $PASSWD_REF ] && rm $PASSWD_REF
    [ -f $WHO_REF ] && rm $WHO_REF
    exit 1
}
```

12.4.4.3 Fichier "mkpasswd.check"

```
#!/bin/sh
#
#
#   SERVICE DES ADMINISTRATEURS:
#       Traduction des bases OpenVMS vers Unix.
#       Verification des donnees
#
#   Fichier: $BIN_DIR/mkpasswd.check
#
#   Creation: S. Baudry
#
#   Modifications:
#
#-------------------------------------------------------------
#-------------------------------------------------------------
#
# Verifie la presence des scripts ou executables externes
#
#-------------------------------------------------------------

if [ ! -x $SEARCHID ]; then
    echo "`basename $1`: impossible de trouver  $SEARCHID" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi

if [ ! -x $BUILDPASSWD ]; then
    if [ ! -f ${BUILDPASSWD}.c ]; then
        echo "`basename $1`: impossible de construire  $BUILDPASSWD" >&2
        echo "`basename $1`: execution avortee" >&2
        exit 1
    fi
    ( cd `dirname $BUILDPASSWD`; make `basename $BUILDPASSWD` ) \
```

```
        >/dev/null 2>&1
    if [ ! -x $BUILDPASSWD ]; then
        echo "`basename $1`: impossible de trouver  $BUILDPASSWD" >&2
        echo "`basename $1`: execution avortee" >&2
        exit 1
    fi
fi

if [ ! -x $ECHO ]; then
    echo "`basename $1`: impossible de trouver  $ECHO" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi

if [ ! -x $AWK ]; then
    echo "`basename $1`: impossible de trouver  $AWK" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi

#------------------------------------------------------------
#
# Verifie la presence des fichiers en entree
#
#------------------------------------------------------------

if [ ! -f $LCLUAF ]; then
    echo "`basename $1`: fichier en entree $LCLUAF manquant" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi

if [ ! -f $LIST ]; then
    echo "`basename $1`: fichier en entree $LIST manquant" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi

if [ ! -f $WHO ]; then
    echo "`basename $1`: fichier en entree $WHO manquant" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi

if [ ! -f $PASSWD ]; then
    echo "`basename $1`: fichier en entree $PASSWD manquant" >&2
    echo "`basename $1`: execution avortee" >&2
    exit 1
fi
```

```
#-----------------------------------------------------------
#
# Verifie la presence des fichiers en sortie
#
#-----------------------------------------------------------

if [ -f $PASSWD_NEW ]; then
    echo "`basename $1`: fichier en sortie $PASSWD_NEW deja existant." >&2
    echo "`basename $1`: $PASSWD_NEW supprime." >&2
    rm -f $PASSWD_NEW
fi

if [ -f $GROUP_NEW ]; then
    echo "`basename $1`: fichier en sortie $GROUP_NEW deja existant." >&2
    echo "`basename $1`: $GROUP_NEW supprime." >&2
    rm -f $GROUP_NEW
fi

if [ -f $SHADOW_NEW ]; then
    echo "`basename $1`: fichier en sortie $SHADOW_NEW deja existant." >&2
    echo "`basename $1`: $SHADOW_NEW supprime." >&2
    rm -f $SHADOW_NEW
fi

if [ -f $USERS_INFO_FILE ]; then
    echo "`basename $1`: fichier en sortie $USERS_INFO_FILE deja existant." >&2
    echo "`basename $1`: $USERS_INFO_FILE supprime." >&2
    rm -f $USERS_INFO_FILE
fi

if [ -f $AUTO_PROJECTS ]; then
    echo "`basename $1`: fichier en sortie $AUTO_PROJECTS deja existant." >&2
    echo "`basename $1`: $AUTO_PROJECTS supprime." >&2
    rm -f $AUTO_PROJECTS
fi

if [ -f $AUTO_USERS ]; then
    echo "`basename $1`: fichier en sortie $AUTO_USERS deja existant." >&2
    echo "`basename $1`: $AUTO_USERS supprime." >&2
    rm -f $AUTO_USERS
fi

exit 0
```

12.4.4.4 Fichier "mkpasswd"

```
#!/bin/sh
#
#
```

```
#   SERVICE DES ADMINISTRATEURS:
#       Traduction des bases OpenVMS vers Unix.
#
#   Fichier: $BIN_DIR/mkpasswd
#
#   Creation: S. Baudry
#
#   Modifications:
#
#-------------------------------------------------------------

$ECHO "`basename $0`: chargement en cours ...\c"

#-------------------------------------------------------------
#
# Repertoires par defaut
#
#-------------------------------------------------------------
MKPASSD_DIR=${MKPASSD_DIR:=/home/adm/users/convert}
export MKPASSD_DIR

BIN_DIR=${BIN_DIR:=$MKPASSD_DIR/bin}
export BIN_DIR

#-------------------------------------------------------------
#
# Chargement des definitions
#
#-------------------------------------------------------------

if [ ! -f $BIN_DIR/`basename $0`.define ]; then
message="impossible de trouver le fichier de definition"
    echo "`basename $0`: $message $BIN_DIR/`basename $0`.define" >&2
    echo "`basename $0`: execution avortee" >&2
    exit
fi

. $BIN_DIR/`basename $0`.define

#-------------------------------------------------------------
#
# Chargement des fonctions
#
#-------------------------------------------------------------

if [ ! -f $BIN_DIR/`basename $0`.functions ]; then
message="impossible de trouver le fichier de definition"
    echo "`basename $0`: $message $BIN_DIR/`basename $0`.functions" >&2
    echo "`basename $0`: execution avortee" >&2
    exit
```

```
fi

. $BIN_DIR/‘basename $0‘.functions

$ECHO " Ok."

#-------------------------------------------------------------
#
# Verification du contexte
#
#-------------------------------------------------------------
$ECHO "‘basename $0‘: Verifications ..."

if [ ! -x $BIN_DIR/‘basename $0‘.check ]; then
message="impossible d'executer le fichier de controle"
    echo "‘basename $0‘: $message $BIN_DIR/‘basename $0‘.check" >&2
    echo "‘basename $0‘: execution avortee" >&2
    exit
fi

$BIN_DIR/‘basename $0‘.check ‘basename $0‘
[ $? -ne 0 ] && exit

$ECHO "‘basename $0‘: verifications Ok."

trap "_stop_exec; exit" 1 9 15

#-------------------------------------------------------------
#
# Creation du fichier temporaire PASSWD_REF
#
#-------------------------------------------------------------
$ECHO "‘basename $0‘: creation de l'espace de travail ...\c"

$AWK '
    BEGIN { FS=":"}
    {
        field = substr ($5, 1, length($5) - 5)
        printf ("%s:%s:%s:%s:%s:%s:%s\n",
            $1, $2, $3, $4, field, $6, $7)
    }
' $PASSWD > $PASSWD_REF

#-------------------------------------------------------------
#
# Creation du fichier temporaire WHO_REF
#
#-------------------------------------------------------------

sed -e 's/ - [0-9][ABC].*$//' $WHO > $WHO_REF
```

```
$ECHO " Ok."

#------------------------------------------------------------
#
# Extraction des anciennes entrees
#
#------------------------------------------------------------

$ECHO "Extraction des anciennes entrees dans passwd : \c"
index=1

cat $LIST |\
while
    read username
do
    _waiting_chars $index
    name=‘grep $username $WHO_REF  | cut -d! -f2 | cut -c2-‘
    is_user=‘$AWK -v name="$name" ’
        BEGIN { FS=":" }
        $5 == name { print $0 }
    ’ $PASSWD_REF‘
    if [ "$is_user" != "" ]; then
        new_who=‘grep $username $WHO‘
        login=‘echo $is_user | cut -d: -f1‘
        new_who=‘echo $new_who | cut -d! -f2 | cut -c2-‘
        project="p‘grep $username $LCLUAF | cut -d: -f1 | tr ’[A-Z]’ ’[a-z]’‘"
        uid=‘echo $is_user | cut -d: -f3‘
        gid=‘echo $is_user | cut -d: -f4‘
        part2=‘echo $is_user | cut -d: -f6-‘
        new_passwd=‘echo $login | cut -c-4‘‘date +%M%S‘
        crypt_passwd="‘$BUILDPASSWD $new_passwd‘"
        echo "${login}:${crypt_passwd}:${uid}:${gid}:${new_who}:$part2" \
            >> $PASSWD_NEW
        echo "${login}  ${FS_SERVER}:${FS_STUDENTS}/${login}" >> $AUTO_USERS
        output="${login}:${crypt_passwd}:0:${PASSWD_WILL_CHANGE}"
        echo "${output}:${PASSWD_EXPIRATION_TIME}:${PASSWD_WILL_EXPIRE}::" \
            >> $SHADOW_NEW
        echo "${login}:${username}:${new_passwd}:${uid}:${project}" \
            >> $USERS_INFO_FILE
    fi
    index=‘expr $index + 1‘
done
echo " Ok."

#------------------------------------------------------------
#
# Creation des nouvelles entrees dans passwd
#
#------------------------------------------------------------
```

```
$ECHO "Creation des nouvelles entrees dans passwd : \c"
index=1

cat $LIST |\
while
    read username
do
    _waiting_chars $index

    is_defined=`$AWK -v username="$username" -F: '
        $2 == username { print $0 }
        ' $USERS_INFO_FILE `
    [ "$is_defined" != "" ] && continue

    login=`echo $username | tr '[A-Z]' '[a-z]'  | cut -d_ -f1`
    new_who=`grep $username $WHO | cut -d! -f2 | cut -c2-`

    login=`echo $login | cut -c-8`

    status=0
    while
        is_allocated=`grep "^${login}:" $PASSWD_NEW`
        [ "$is_allocated" != "" ]
    do
        echo "" >&2
        echo "`basename $0`: login $login deja alloue pour $username" >&2
        $ECHO "`basename $0`: Quel login : \c" >&2
        read login < /dev/tty
        status=1
    done
    [ $status -eq 1 ] && $ECHO "Creation des nouvelles entrees dans passwd : \c"

    project="p`grep $username $LCLUAF | cut -d: -f1 | tr '[A-Z]' '[a-z]'`"

    new_passwd=`echo $login | cut -c-4``date +%M%S`
    crypt_passwd="`$BUILDPASSWD $new_passwd`"
    uid=`$SEARCHID -u`
    gid=$STUDENT_GID
    student_home=${HOME_STUDENT}/$login

    output="${login}:${crypt_passwd}:${uid}:${gid}"
    echo "${output}:${new_who}:${student_home}:${LOGIN_SHELL}" >> $PASSWD_NEW
    echo "${login}  ${FS_SERVER}:${FS_STUDENTS}/${login}" >> $AUTO_USERS
    output="${login}:${crypt_passwd}:0:${PASSWD_WILL_CHANGE}"
    echo "${output}:${PASSWD_EXPIRATION_TIME}:${PASSWD_WILL_EXPIRE}::" \
        >> $SHADOW_NEW
    echo "${login}:${username}:${new_passwd}:${uid}:${project}" >> $USERS_INFO_FILE

    index=`expr $index + 1`
```

```
done
echo " Ok."


#------------------------------------------------------------
#
# Suppression des anciens repertoires
#
#------------------------------------------------------------
$ECHO "Suppression des anciens repertoires: \c"
index=1

cat $PASSWD_REF |\
while
    read line
do
    old_login=`echo $line | cut -d: -f1`
    is_present=`$AWK -F: -v old_login="$old_login" '
        $1 == old_login { print $1 }
        ' $USERS_INFO_FILE`
    if [ "$is_present" = "" ]; then
        [ -d $FS_STUDENTS/$old_login ] && \
            rm -rf $FS_STUDENTS/$old_login 2>&/dev/null
    fi
    index=`expr $index + 1`
done

echo " Ok."

#------------------------------------------------------------
#
# Creation des nouvelles entrees dans group
#
#------------------------------------------------------------

$ECHO "Creation des nouvelles entrees dans group : \c"
index=1

cat $LCLUAF |\
while
    read line
do
    _waiting_chars $index

    vms_project="`echo $line | cut -d: -f1 | tr '[A-Z]' '[a-z]'`"
    unix_project="p${vms_project}"
    project_gid=`$SEARCHID -g`
    $ECHO "${unix_project}:*:${project_gid}:\c" >> $GROUP_NEW
    members="`echo $line | cut -d: -f2 | sed -e 's/,/ /g'`"
    group_members=`for this_member in $members
```

```
    do
        awk -F: -v  this_member=$this_member '
            $2 == this_member { printf ("%s,",$1) }
            ' $USERS_INFO_FILE
    done`
    echo "$group_members" | sed -e 's/,$//' >> $GROUP_NEW
    echo "${unix_project}   ${FS_SERVER}:${FS_PROJECTS}/${unix_project}" \
        >> $AUTO_PROJECTS

    index=`expr $index + 1`

done

echo " Ok."

#------------------------------------------------------------
#
# Creation des repertoires
#
#------------------------------------------------------------

$ECHO "Creation des repertoires : \c"
index=1

cat $AUTO_USERS | cut -d: -f2 |\
while
    read directory
do
    _waiting_chars $index

    owner=`basename $directory`
    uid=`awk -F: -v owner=$owner '
        $1 == owner { print $3 }
        ' $PASSWD_NEW`
    if [ ! -d $directory ]; then
        mkdir -p $directory
        cp $HOME_PROTOTYPE/.[a-z]* $HOME_PROTOTYPE/.[A-Z]*      \
            $HOME_PROTOTYPE/* $FS_STUDENTS/$newname 2>/dev/null
        chown -R ${uid}.${STUDENT_GID} $directory
    fi
    index=`expr $index + 1`
done

cat $AUTO_PROJECTS | cut -d: -f2 |\
while
    read directory
do
    _waiting_chars $index

    group=`basename $directory`
```

```
    gid=‘awk -F: -v group=$group ’
        $1 == group { print $3 }
        ’ $GROUP_NEW‘
    if [ ! -d $directory ]; then
        mkdir -p $directory
        chown -R root.${gid} $directory
    fi
    index=‘expr $index + 1‘
done

echo " Ok."

#------------------------------------------------------------
#
# Fin du programme
#
#------------------------------------------------------------

trap ’’ 1 9 15
[ -f $PASSWD_REF ] && rm $PASSWD_REF
[ -f $WHO_REF ] && rm $WHO_REF
```

Quatrième partie

Annexes

Annexe A

Instructions de formatage

La chaîne de contrôle dans diverses fonctions standards des bibliothèques de programmation d'UNIX est une suite de symboles définissant le formatage de la sortie des arguments. Quelques unes de ces fonctions sont :

- `printf(3)`, `sprintf(3)`, `fprintf(3)`, ···
- `scanf(3)`, `sscanf(3)`, `fscanf(3)`, ···
- `printf(1)`,
- etc.

Ces codes de contrôle sont aussi utilisés dans "`awk`" et "`perl`"[1]

Une chaîne de contrôle contient deux types d'objets :

- les caractères ordinaires (ils sont recopiés tels quels),
- les symboles de formatage. Ils correspondent aux positions relatives des arguments de la fonction.

Les symboles de formatage sont de la forme "`%[-][m][.n]a`" avec :

Symbole	Description
`%`	introduit un symbole de formatage.
`-`	oblige le cadrage à gauche (par défaut à droite) du champ affiché.
`m`	spécifie la largeur minimum du champ.
`n`	spécifie le nombre maximum de caractères à afficher dans la chaîne correspondante, ou bien le nombre de décimales à afficher pour la valeur numérique correspondante.
`a`	désigne le type d'argument correspondant.

Les différents codes possibles pour désigner le type d'arguments sont :

1. "`perl`" est un langage de programmation largement utilisé dans les serveurs Web et disponible sur la quasi-totalité des systèmes d'exploitation : UNIX, OpenVMS, Windows, MacOS. Il utilise la notion d'expressions régulières, d'objets, etc.

Symbole	Description
`s`	désigne une chaîne de caractères.
`c`	désigne un caractère.
`f`	désigne une valeur réelle (virgule flottante).
`d`	désigne une valeur décimale.

Exemple A.1 :

Exemple d'utilisation des codes de format :
Si la variable "`cumul`" est égale à "31,12345" alors :

Chaîne de contrôle	Affichage
`%f`	`31.12345`
`%10.2f`	`31.12`
`%-10.3f`	`31.123`

Si la variable "`nom`" contient la chaîne "`schmoll`", alors :

Chaîne de contrôle	Affichage
`%s`	`schmoll`
`%10s`	`␣␣␣schmoll`
`%10.3s`	`␣␣␣␣␣␣␣sch`
`%-10.3s`	`sch␣␣␣␣␣␣␣`
`%.3s`	`sch`

A.1 Exercices d'utilisation des expressions régulières

Soit le tableau de chaînes suivant :

Numéro	Chaîne
a	`abc`
b	`zzzz␣xx`
c	`abcdef`
d	`123456␣7890␣abcaziuz`
e	`yyyy`
f	`xyz␣stuv␣abc`
g	`xx␣abcxxxxxxxxx`
h	`xAb*␣␣12345`
i	`xAB*␣␣45678`
j	`98745␣xaB*␣23654`
k	`abc$!k;`
l	`567`
m	`5666777`
n	`57`
o	`Suite...␣du␣paragraphe`
p	`Suite...␣de␣l'histoire`
q	`la␣suite...`
r	`Suite..␣au␣prochain␣numero.`

Pour chaque expression régulières ci dessous, trouver la ou les chaînes du tableau ci-dessus satisfaisant à l'expression régulière :

	Expression Régulière	Chaînes correspondantes
1	`c$`	
2	`c\$`	
3	`^abc`	
4	`abc$`	
5	`^abc$`	
6	`^abc.`	
7	`45`	
8	`^56[67]`	
9	`.56[67]`	
10	`x[Aa][Bb]`	
11	`x[^Aa]`	
12	`[Aa][^b]`	
13	`abcd*`	
14	`566?7`	
15	`[r-v]`	
16	`56*7*`	
17	`56+7+`	
18	`56?7?`	

Suite page suivante ···

Suite de la page précédente ···

	Expression Régulière	Chaînes correspondantes
19	`566?7`	
20	`987|789`	
21	`abc|[def]`	
22	`.*[Aa][Bb].*12.*`	
23	`.*12.*[Aa][Bb]`	
24	`.*[Aa]b.*12.*|.*12.*[Aa]b.*`	
25	`.*([Aa]b.*12.*|.*12.*[Aa]b).*`	
26	`abc[def][m-x]*`	
27	`^[Ss]uite\.\.\..*`	

Annexe B

Correspondances entre le Bourne Shell et le C Shell

B.1 Manipulation des variables

C Shell	Bourne Shell
Syntaxe	
`set␣`*variable*`=`*valeur* `unset␣`*variable*	*variable*`=`*valeur* `unset␣`*variable*
Remarques	
– Les variables peuvent être de type caractère ou numérique entier. – Un nombre quelconque d'espaces entre les mots et les séparateurs est autorisé. – On peut utiliser des tableaux dont le dimensionnement est défini à l'initialisation. La numérotation commence à 1.	– Les variables peuvent être de type caractère ou numérique entier. – Pas d'espace de part et d'autre du signe "=". – Adoptez les traitements orientés liste (non bornée) en alternative aux tableaux.
Exemples	
`set␣a="toto"` `set␣b␣=␣titi`	`a="toto"` `b=titi`
Suite page suivante ···	

Suite de la page précédente ···

C Shell	Bourne Shell
`set␣liste=nom1␣nom2␣nom3`	`liste="nom1␣nom2␣nom3"`
`set␣x="5"`	`x="5"`
`set␣i=1`	`i=1`
`set␣tab=(1␣2␣3␣4␣5␣6␣7␣8␣9)`	pas d'équivalent en local, mais on peut allouer une liste de valeurs dans l'environnement par `"set␣1␣2␣3␣4␣5␣6␣7␣8␣9"`, ces valeurs étant récupérables via les paramètres positionnels `$1` à `$9`.
`"$tab[6]"` contient le caractère `"6"`	`"$6"` contient le caractère `"6"`, au delà de 9 paramètres, il est nécessaire d'utiliser la commande `"shift"`.

B.2 Évaluation de variables

C Shell	Bourne Shell
Syntaxe	
$*variable*	$*variable*
${*variable*}	${*variable*}
${?*variable*} vaut `"1"` si "*variable*" est initialisée, `"0"` sinon.	Pas d'équivalent.
$*variable*[*indice*]	Pas d'équivalent.
${*variable*[*indice*]}	Pas d'équivalent
Remarques	
Préférez la notation avec les accolades «`{}`" afin d'éviter les problèmes lors des concaténations. En effet, `"${var}_extension"` est différent de `"$var_extension"`.	Préférez la notation avec les accolades «`{}`" afin d'éviter les problèmes lors des concaténations. En effet, `"${var}_extension"` est différent de `"$var_extension"`.
Exemples	
`set␣rep1=~/essai`	`rep1=$HOME/essai`
`cat␣${rep1}/fichier`	`cat␣${rep1}/fichier`

B.3 Expression arithmétiques

C Shell	Bourne Shell
Syntaxe	
`set␣`*variable*`=`*expression*	*variable*`=`*expression*
`@`*variable*`=`*expression*	*variable*`=‘expr␣`*expression*`‘`
`@`*variable*`[`*indice*`]=`*expression*	Pas d'équivalent.
Opérations arithmétiques et assignations	
`+` addition	`+` addition
`-` soustraction	`-` soustraction
`*` multiplication	`*` multiplication
`/` division	`/` division
`%` reste	`%` reste
`()` pour forcer l'ordre d'évaluation	`\(\)` pour forcer l'ordre d'évaluation
Remarques	
– Le symbole "`@`" est synonyme de "`set`". – Ne pas oublier un espace autour de chaque opération arithmétique. – L'indice est lui-même une variable entière.	– **Attention**, c'est la commande "`expr`" qui assure l'évaluation de l'expression arithmétique. Aussi pour lever l'ambiguïté d'interprétation de certains caractères spéciaux, l'usage du "`\`" sert à les protéger.
Exemples	
`@i␣+=␣1` ou `@i␣++`	`i=‘expr␣$i␣+␣1‘`
`@a␣=␣$b␣+␣$c`	`$a=‘expr␣$b␣+␣$c‘`
`@x␣=␣5␣*␣4`	`x=‘expr␣5␣\*␣4‘`
`set␣c␣=␣(300␣4)`	`set␣300␣4`
`set␣i␣=␣$#c`	`i=$#`
`@j␣=␣$i␣-␣1`	`rel=‘echo␣\$$i‘`
`@f␣=␣($c[$i]␣*␣$c[$j])␣+␣$x`	`c2=‘eval␣$rel‘`
	`j=‘expr␣$i␣-␣1‘`
	`rel=‘echo␣\$$j‘`
	`c1=‘eval␣$rel‘`
	`f=‘expr␣\($c1␣\*␣$c2\)␣+␣$x‘`

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell
`"f"` vaut 1220	`"f"` vaut 1220

B.4 Variables formelles

C Shell		Bourne Shell	
Syntaxe			
`$$`	PID du processus courant	`$$`	PID du processus courant
`$0` à `$n`	Arguments (peut être supérieur à 9)	`$0` à `$n`	Arguments (avec $n \leq 9$)
`$`n	n^e argument	`$`n	n^e argument
`$argv[`n`]`	n^e argument		
`$#argv[*]`	nombre d'arguments	`$#`	nombre d'arguments
`$*` ou `$argv[*]`	liste des arguments	`$*`	liste des arguments
`$<`	ligne courante en entrée	`read`	ligne courante en entrée
`$status`	état de la dernière commande	`$?`	état de la dernière commande
Remarques			
Attention, l'usage de `"set noglob"` inhibe la substitution des noms de fichiers et donc la prise en compte des métacaractères associés.		Attention, l'usage de `"-f"` à l'invocation du Bourne Shell (`"#!/bin/sh -f"` ou `"set -f"`) inhibe la substitution des noms de fichiers et donc la prise en compte des métacaractères associés.	

C Shell	Bourne Shell
Exemples	
`set␣reponse=$<`	`read␣reponse`
`ls␣*.data`	`ls␣*.data`
`echo␣$status`	`echo␣$?`
`set␣nb_params␣=␣$#argv`	`nb_params=$#`
`set␣premier␣=␣$argv[1]`	`premier=$1`
`set␣deuxieme=$argv[2]`	`deuxieme=$2`

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell
`set␣dernier=$argv[$nb_params]`	`shift␣$#` `dernier=$1`

B.5 Environnement

B.5.1 Empilement de variables

C Shell	Bourne Shell
Syntaxe	
`setenv␣`*variable*`␣`*valeur*	*variable*=*valeur* `export␣`*variable*
Exemple	
`setenv␣REPERTOIRE␣../source`	`REPERTOIRE="../source"` `export␣REPERTOIRE`

B.5.2 Variables d'environnement

C Shell		Bourne Shell	
`PATH`	chemin d'accès aux commandes	`PATH`	chemin d'accès aux commandes
`path`	chemin d'accès aux commandes		
`TERM`	type de terminal	`TERM`	type de terminal
`term`	type de terminal		
`MAIL`	répertoire pour la boîte aux lettres	`MAIL`	répertoire pour la boîte aux lettres
`USER`	nom de l'utilisateur	`USER`	nom de l'utilisateur
`HOME`	répertoire de connexion	`HOME`	répertoire de connexion
`home`	répertoire de connexion		
`DISPLAY`	écran pour X-Window	`DISPLAY`	écran pour X-Window
`TZ`	fuseau horaire	`TZ`	fuseau horaire
`LANG`	langue nationale	`LANG`	langue nationale

Suite page suivante ···

Suite de la page précédente ···

C Shell		Bourne Shell	
`status`	état de la dernière commande	`?`	état de la dernière commande
`prompt`	caractère d'invite ("*prompt*")	`PS1`	caractère d'invite ("*prompt*")
`EXINIT`	nom du fichier d'initialisation de `"vi"`	`EXINIT`	nom du fichier d'initialisation de `"vi"`
`cwd`	nom du répertoire courant	pas d'équivalent en Bourne Shell. Cependant il existe la variable `"PWD"` en Korn Shell donnant la même information.	
etc.		etc.	

Remarques	
Ces variables sont initialisées au démarrage de la session. Chaque script peut y accéder y faisant référence en la faisant précéder du caractère `"$"`. Elles sont modifiables et connues de tous les scripts de la session. La variable `"path"` et `"PATH"` ont la même fonctionnalité **mais obéissent à des syntaxes de spécification différentes**. Les variables `"path"` et `"term"` s'initialisent comme des variables **locales** et non pas comme des variables d'environnement.	Elles sont initialisées au démarrage de la session. Elles sont par convention en majuscules. Chaque script peut y accéder y faisant référence en la faisant précéder du caractère `"$"`. Elles sont modifiables et connues de tous les scripts de la session.

|Exemples|	
`echo $HOME`	`echo $HOME`
`set prompt="<>"`	`PS1="<>"`
`setenv DISPLAY myhost:0.0`	`DISPLAY=myhost:0.0`
	`export DISPLAY`
`setenv TERM vt100`	`TERM=vt100; export TERM`

B.6 Entrées/Sorties et redirections

B.6.1 Entrée en ligne

C Shell	Bourne Shell
Syntaxe	
commande␣<<␣*borne*	*commande*␣<<␣*borne*

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell
Exemple	

`cat << EOF`
`  1- choix1`
`  2- choix2`
`EOF`
	`cat << EOF`
	`  1- choix1`
	`  2- choix2`
	`EOF`

B.6.2 Séparation, regroupement des sorties (standard et d'erreurs)

C Shell	Bourne Shell
Syntaxe	
(*commande*␣>␣*fichier.sortie*␣>&␣*fichier.err* Le seul moyen de séparer les sorties nécessite un sous shell (utilisation de "(" et de ")").	*commande*␣2>␣*fichier.err* `commande␣>␣fichier.log␣2>fichier.err`
Regroupement	
`commande␣>&␣fichier.trace`	`commande␣1>fichier.trace␣2>&1`
Exemple	

`(cc␣-c␣source.c)␣>&␣source.err`
`(f77␣-v␣-u␣source.f␣>␣source.lst␣)␣\`
`>&␣warnings`
`./mon.script␣>&␣result`
	`cc␣-c␣source.c␣2>source.err`
	`f77␣-v␣-u␣source.f␣>␣source.lst␣\`
	`2>␣warnings`
	`./mon.script␣1>result␣2>&1`

B.7 Les fonctions

B.7.1 Les fonctions internes (*built in*)

Nous n'expliciterons dans ce paragraphe que les principales commandes internes du C Shell ayant un équivalent en Bourne Shell.

C Shell	Bourne Shell
Syntaxe	
`cd␣`*répertoire*	`cd␣`*répertoire*

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell
`echo␣`*texte*	`echo␣`*texte*
`eval␣`*argument*	`eval␣`*argument*
`exec␣`*commande*	`exec␣`*commande*
`shift`	`shift`
`source␣\textsl{script}`	`.␣`*script*
`time`	`times`
`wait`	`wait`

B.7.2 Les fonctions externes

Le C Shell **ne supporte pas** la notion de fonctions, par contre on peut simuler leur avec la commande "`goto`". Le Bourne Shell, par contre, supporte les fonctions. Pour plus de renseignements, reportez vous à la section 5.2.

Exemple B.2 :

```
#!/bin/sh
#appel de fonctions externes
_f1()
{
    echo "execution fonction f1"
    return
}

_f2 ()
{
    echo "execution fonction f2"
    return
}

_main ()
{
    # appel fonctions
    f1
    echo "retour en s{\'e}quence"
    f2
    echo "retour en s{\'e}quence"
    exit 0
}
```

```
_main
```

B.8 Les structures de contrôle

B.8.1 Les tests ("if")

C Shell	Bourne Shell
Syntaxe	

`if` ␣`(`␣*expression*␣`)`␣`then`
`...`
`else` ␣`if` ␣`(`␣*expression*␣`)`
`...`
`else`
`...`
`endif`
	`if` ␣`[`␣*expression*␣`]`
	`then`
	`...`
	`elif` ␣`[`␣*expression*␣`]`
	`then`
	`...`
	`else`
	`...`
	`fi`

Remarque	
Les instructions et les évaluations de tests sont internes au C Shell.	Un espace est **indispensable** entre les crochets (`"["`, `"]"`) et l'expression. L'évaluation des tests est externe au Bourne Shell.

C Shell		Bourne Shell	
Opérateurs			
`!`	négation	`!`	négation
`==`	égal	`=`	égal (chaînes)
		`-eq`	égal (nombres)
`!=`	différent	`!=`	différent (chaînes)
		`-ne`	différent (nombres)
`<`	inférieur	`-lt`	inférieur
`<=`	inférieur ou égal	`-le`	inférieur ou égal
`>`	supérieur	`-gt`	supérieur

Suite page suivante ···

Suite de la page précédente ···

C Shell		Bourne Shell	
`>=`	supérieur ou égal	`-ge`	supérieur ou égal
`&&`	et (logique)	`-a`	et (logique)
`||`	ou (logique)	`-o`	ou (logique)
Remarque			
cf. "`csh(1)`" pour les opérateurs sur les fichiers.		cf. "`test(1)`" pour les opérateurs sur les fichiers.	

C Shell	Bourne Shell
Exemple	

`if␣(-x␣$dir/$file)␣then`
`␣␣␣source␣$dir/$file`
`endif`
 	
`set␣system=‘uname␣-a␣|␣\`
`␣␣␣cut␣-d’␣’␣-f1‘`
`if␣(␣"$system"␣==␣"Irix")␣then`
`␣␣␣setenv␣MACHINE␣SGI`
`endif`
	`if␣[␣-x␣$dir/$file␣]`
	`then`
	`␣␣␣.␣$dir/$file`
	`fi`
	`system=‘uname␣-a␣|␣\`
	`␣␣␣cut␣-d’␣’␣-f1‘`
	`if␣[␣"$system"␣==␣"Irix"];␣then`
	`␣␣␣MACHINE=SGI`
	`␣␣␣export␣MACHINE`
	`fi`

B.8.2 Choix multiples (`case`, `switch`)

C Shell	Bourne Shell
Syntaxe	
`switch␣(`*nom*`)`	`case␣`*nom*`␣in`
`␣␣case␣`*label1*`:`	*label1*`)`
commandes	*commandes*
...	...
`breaksw`	`;;`
`␣␣case␣`*label2*`:`	*label2*`)`
commandes	*commandes*
...	...

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell

`breaksw`
`default:`
`# tous les autres cas`
commandes
...
`breaksw`
`endsw`
	`;;`
	`*)`
	`# tous les autres cas`
	commandes
	...
	`;;`
	`esac`

Remarques	
Les labels suivant le "`case`" peuvent exprimer une alternative de type "*ou*" avec la notation "`[]`", ainsi: "`case␣[oO][uU][iI]:`" accepte "*oui*" et "*OUI*" ainsi que toutes les combinaisons possibles avec les minuscules et les majuscules.	Les labels de chaque cas peuvent exprimer un alternative de type "*ou*" en étant séparés par le caractère "`|`", ainsi "`oui|OUI)`" accepte seulement les deux combinaisons suivantes: "*oui*" et "*OUI*".
Le branchement s'effectue sur une clause "`case`" en fonction de la valeur spécifiée entre parenthèses à l'instruction "`switch`". Les instructions sont exécutées jusqu'à ce que "`breaksw`" (ou "`endsw`") soit localisé.	Le branchement s'effectue sur une clause "`label)`" en fonction de la valeur spécifiée entre les mots clef "`case`" et "`in`". Les instructions sont exécutées jusqu'à ce que "`;;`" (ou "`esac`") soit localisé.

Exemple

```
#!/bin/csh
switch ($MACHINE)
   case "SUN":
      echo "Je suis sur Sun"
      breaksw
   case "SGI":
      echo "Je suis sur Silicon"
      breaksw
   case "DEC":
   case "HP":
   case "IBM":
      echo "Je suis sur DEC"
```

```
#!/bin/sh
case $MACHINE in
   "SUN")
      echo "Je suis sur Sun."
      ;;
   "SGI")
      echo "Je suis sur Silicon."
      ;;
   "DEC"|"HP"|"IBM")
      echo "Je suis sur DEC"
      echo "Peut-etre ou HP"
      echo "Ou bien encore IBM."
```

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell

`    echo "Peut-etre ou HP"`
`    echo "Ou bien encore IBM."`
`default:`
`    echo "Je suis en CSH"`
`endsw`
	`    ;;`
	`*)`
	`    echo "Je suis en SH"`
	`esac`

B.8.3 Les boucles

B.8.3.1 Boucle du type "`for`"

C Shell	Bourne Shell
Syntaxe	

`foreach` *variable* `(`*liste*`)`
commandes
`end`
	`for` *variable* `in` *liste*
	`do`
	commandes
	`done`

Exemple	

`set nom="part"`
`foreach fichier (‘ls $nom.?a‘)`
`pr $fichier | lp -`
`end`
	`nom="part"`
	`for fichier in ‘ls $nom.?a‘`
	`do`
	`pr $fichier | lp -`
	`done`

B.8.3.2 Boucle du type "`while`"

C Shell	Bourne Shell
Syntaxe	

`while (`*expression*`)`
commandes
`end`
	`while [` *expression* `]`
	`do`
	commandes

Suite page suivante ···

Suite de la page précédente ···

C Shell	Bourne Shell
	`done`
	Remarque **Ne pas oublier les espaces** pour séparer "*expression*" des crochets ("[]").
Exemple	

`while␣($#argv)`
`␣␣echo␣$argv[1]`
`␣␣shift`
`end`
	`while␣[␣$#␣-gt␣0␣]`
	`do`
	`␣␣echo␣$1`
	`␣␣shift`
	`done`

B.8.4 Tableaux et listes

Le Bourne Shell ne connaît pas les tableaux. Par contre, il traite les listes aisément. On peut donc simuler les tableaux grâce à elles.

Exemple B.3 :

```
#!/bin/sh
for script in ‘ls ex?.sh
do
    . ./$script
done
exit 0
```

Les tableaux existent en C Shell. Ils doivent être réservés aux manipulation de variables numériques. On peut toutefois y mettre d'autres objets comme des noms de fichiers comme le montre l'exemple ci-dessous.

Exemple B.4 :

```
#!/bin/csh
set liste_fic=‘ls ex?.csh‘
set tableau=($liste_fic)
set num=$#liste_fic
@i=1
while ($i <= $num)
    source $tableau[$i]
    @i++
end
exit 0
```

La solution ci-dessous est préférable (car plus lisible).

```
#!/bin/csh
```

```
foreach script (`ls ex?.csh`)
    source $script
end
exit 0
```

B.8.5 Interruptions, déroutements sur signaux

C Shell	Bourne Shell
Syntaxe	
`onintr␣[-l␣`*label*`]`	`trap␣[`*liste commandes*`]␣[`*liste signaux*`]`
Remarque Seul le signal "*9*" ne peut être masqué.	
Exemple	
`onintr␣-` → Ignorer toute interruption.	`trap␣''␣0␣1␣2␣3␣15` → Ignorer toute interruption.
`onintr␣sortie` → Aller au label "`sortie`" du script courant.	`trap␣commande_on_exit␣0␣1␣3` → Exécute la commande "`commande_on_exit`" lorsque le processus reçoit les signaux 0, 1, 3 et 15.
`onintr` → Restaure le comportement par défaut pour tous les signaux.	`trap␣0␣1␣2␣3␣15` → Restaure le comportement par défaut pour tous les signaux.

B.8.6 Les retours

C Shell	Bourne Shell
Syntaxe	
`exit␣(`*expression*`)`	`exit␣`*valeur* `return␣`*valeur*
`exit`	`exit`
Remarques	
Suite page suivante ···	

Suite de la page précédente ···

C Shell	Bourne Shell
La valeur par défaut, envoyée par la commande "exit" est celle de la variable "status". Sa valeur est nulle si la dernière commande s'est exécutée sans erreurs, elle est non-nulle dans le cas contraire.	La valeur par défaut, envoyée par la commande "exit" est celle de la variable "status". Sa valeur est nulle si la dernière commande s'est exécutée sans erreurs, elle est non-nulle dans le cas contraire. "return" a la même fonctionnalité que "exit" **mais seulement pour des fonctions**. En conséquence, "return" permet de sortir d'une fonction sans quitter le script. "exit" termine le processus courrant (donc le script courrant).
Exemple	

`set␣machine=$1`
`if␣($machine␣=="")␣then`
`␣␣␣echo␣"Oubli␣de␣param{\‘e}tre"`
`␣␣␣exit␣(1)`
`endif`
	`machine=$1`
	`if␣[␣-z␣"$machine"␣]`
	`then`
	`␣␣␣echo␣"Oubli␣de␣param{\‘e}tre"`
	`␣␣␣exit␣1`
	`fi`

Annexe C

Utilisation des éditeurs de texte sous Unix

C.1 Introduction

Dans ce chapitre de l'annexe, nous allons décrire les deux éditeurs de texte les plus communément utilisés sous UNIX :

- "vi",
- "emacs".

"vi" est l'éditeur de texte de base sur tout système UNIX. Il est livré systématiquement et utilise toutes les notions de syntaxes vues avec les utilitaires comme "sed" et "awk" (cf. sections 10 et 11). Il fonctionnera sur tout type de terminal, même sur un terminal télétype[1]. "vi" n'est pas réputé pour sa convivialité. Il demande un certain temps d'adaptation. Une fois que l'on a réussi à s'y habituer, son utilisation devient aisée et toutes ses fonctionnalités sont très rapides d'accès.

"emacs" est un éditeur de texte du domaine public, livré maintenant en standard sur certains UNIX (comme "Digital UNIX"[2], "LINUX"[3] "Irix"[4]). Dans le cas contraire, vous devrez aller chercher les sources et les recompiler sur votre machine[5]. Cet éditeur fonctionne aussi sur d'autres systèmes d'exploitation comme MacOS, Windows et OpenVMS. "emacs" est beaucoup plus facile d'approche que "vi". Toutefois, une utilisation poussée de cet éditeur montre que, lui aussi, nécessite un apprentissage d'un nombre impressionnant de séquence de touches[6]

"emacs" est entièrement reprogrammable. En effet, il s'appuie sur un moteur Lisp[7]. On peut donc développer toutes les extensions que l'on désire grâce à ce langage. "emacs" se rapprocherait donc de l'éditeur "LSE"[8] et du langage associé "TPU"[9]. De nombreuses extensions ont été réalisées pour "emacs", à un point tel que certaines personnes s'en servent comme un environnement de travail complet (gestionnaire de fichiers, logiciel de messagerie, navigateur Web, environnement de développement, etc.). Il existe une version allégée d'"emacs" sans le moteur Lisp: "micro-emacs".

Remarque C.1 :

Il est possible de se programmer un environnement d'édition avec "vi", mais il n'est pas programmable au sens où on l'entend pour l'éditeur "emacs".

Remarque C.2 :

1. même chose qu'un terminal ASCII classique mais l'écran est remplacé par une imprimante
2. UNIX livré sur les machines Digital, comme les *AlphaStations*, *AlphaServers*, etc.
3. UNIX du domaine public pouvant tourner sur les architectures PC-Intel, PowerMacintosh ou compatibles, machines à base de processeurs MIPS et Alpha
4. UNIX de Silicon Graphics
5. ftp://ftp.ibp.fr/pub/gnu par exemple.
6. les mauvaises langues diront qu'"emacs" représente les initialises de ESC, META, ALT, CTRL et SHIFT.
7. Le langage Lisp était très utilisé dans les développements en intelligence artificiel dans la fin des années 80.
8. DEC Language Sensitive Editor, disponible sous OpenVMS.
9. DEC Text Processing Utility, disponible sous OpenVMS.

Il existe d'autres éditeurs de texte sous UNIX, *mais moins répandus ou bien spécifique à un constructeur comme :*

- `textedit` *éditeur de texte livré en standard sur les machines livrées avec SunSolaris et offrant les fonctions de base d'un éditeur de texte.*
- `jot` *éditeur de texte livré en standard sur les machines livrées avec Irix de Silicon Graphics et offrant les fonctions de base d'un éditeur de texte.*
- `dtpad`, *éditeur de texte offrant les fonctions de base, livré avec l'environnement CDE*[10],
- *etc.*

10. Common Desktop Environment, environnement utilisateur normalisé par l'OSF, afin d'avoir un environnement identique entre constructeurs de stations de travail. Actuellement, cet environnement est disponible sur tous les UNIXs du marché et même sur des systèmes propriétaires comme OpenVMS de Digital Corp.

C.2 Commandes de base de l'éditeur `vi`

C.2.1 Introduction et conventions

Cette section donne un aperçu du fonctionnement de l'éditeur de texte "`vi`". Cet éditeur est en standart sur l'ensemble des systèmes sous UNIX, voire même sur d'autres systèmes d'exploitation comme OpenVMS, MS-DOS, MacOS, Windows et Windows–NT. Par conséquent, à partir d'une installation de base de quelqu'UNIX que ce soit, vous aurez toujours "`vi`" à votre disposition. De même, c'est cet éditeur qui est utilisé par défaut pour de nombreuses commandes de configuration du système, car, peut importe le type de terminal utilisé pour la console d'administration, "`vi`" vous permettra d'éditer les différents fichiers de configuration.

"`vi`" souffre d'une très mauvaise image de par son peu de convivialité. Cependant, il peut être utilisé sur n'importe quel type de terminal `ASCII`, contrairement à `LSEDIT` sur OpenVMS et dispose de nombreuses fonctionnalités facilitant l'écriture de programmes complexes.

Par contre, contrairement à «`emacs`", "`vi`" n'est pas un éditeur programmable, c'est-à-dire extensible grâce à un langage de programmation.

Dans toute la suite, les conventions suivantes seront utilisées :

- Toute commande précédée par le caractère "`:`" correspond à une commande interne à "`vi`". Par conséquent, elle doit être validée par RETURN .
- *fichier* désigne le nom d'un fichier.
- *cmd_curseur* désigne une commande liée au déplacement du curseur dans "`vi`".
- *caract* désigne un et un seul caractère.
- *str* désigne une chaîne de caractères ou bien une expression régulière (cf. chapitre 8).
- *n,m* représentent :
  - ⋆ soit deux numéros de lignes, par exemple "`4,50`",
  - ⋆ soit deux marqueurs de lignes, par exemple "`.,$`",
  - ⋆ soit deux expressions de recherche utilisant les expressions régulières (cf. chapitre 8), par exemple "`/recherche␣1/,/recherche␣2/`",
  - ⋆ soit toute combinaison possible des trois possibilités précédentes, par exemple "`1,.`", "`/recherche␣1/,$`".
- *(a-z)* désigne n'importe quel caractère de "`a`" à "`z`"

Nous avons introduit la notion de marqueur de ligne. Un marqueur est une convention pour désigner certaines lignes particulières dont il n'est pas possible de déterminer le numéro à l'avance. Les deux marqueurs prédéfinis dans "`vi`" sont :

Marqueur	Description

Suite page suivante ···

`.`	ligne courante, c'est-à-dire la ligne où se trouve actuellement le curseur,
`$`	fin de fichier

C.2.2 Modes de fonctionnement de "`vi`"

"`vi`" distingue deux modes de fonctionnements :

le mode "*commande*"
: Ce mode correspond lorsque vous n'êtes pas en train de saisir du texte. Pour cela, chaque touche du clavier alphanumérique correspond à une fonctionnalité de l'éditeur de texte. Vous pouvez déplacer le curseur, rechercher du texte, sauvegarder le *buffer* courrant, appeler le *prompt* de "`vi`" afin de saisir des commandes internes de l'éditeur. Comme tout utilitaire UNIX, "`vi`" fait la différence entre les majuscules et les minuscules. Par conséquent, **vous devez prendre garde que la touche** CAPS **ou "*Verrouillage en majuscule*" ne soit pas en fonction**. Dans ce cas, le comportement que pourrait avoir "`vi`" risque de vous surprendre et **surtout** ne pas faire ce que vous souhaité (mais par contre ce qui a été demandé !).

le mode "*insertion*"
: Ce mode correspond lorsque vous êtes en train de saisir ou de modifier du texte. Vous pouvez insérer autant de lignes que vous voulez, par contre, vous ne pouvez modifier le texte que de la ligne courrant.
Pour quitter le mode "*insertion*", il vous suffit d'appuyer sur la touche ESC .

À la section C.2.1, nous avons parlé de commandes "`vi`" commençant par le caractère "`:`" et validée par RETURN . Ce mode de fonctionnement correspond au "*prompt*" de "`vi`", invite à partir de laquelle il est possible de saisir des commandes internes à "`vi`". Ce mode est accessible à partir du mode "*commande*". Les différences entre ces deux comportements sont les suivantes :

- En mode *commande*, toutes les touches du clavier alphanumérique ont une fonction particulière, par exemple pour déplacer le curseur, effacer des caractères ou des lignes, sélectionner du texte, etc. Ce mode serait l'équivalent, sous les éditeurs de texte de OpenVMS, d'accéder à toutes les fonctionnalités des touches du pavé numérique sur des claviers LK200, LK400. En effet, pour ces éditeurs, chaque touche correspond à une fonctionnalité donnée. Sous "`vi`", le clavier numérique est censé permettre d'afficher des nombres ou d'utiliser les flèches (sur des claviers de type PC). Il a donc fallu trouver une méthode pour associer des raccourcis clavier à un certain nombre de fonctionnalité. L'une d'entre elle est d'accéder au "*prompt*" de "`vi`" afin de saisir des commandes internes à l'éditeur.

– À l'invite de "`vi`", il est possible de saisir un certain nombre de commandes explicitées dans toute la suite. Ces commandes servent à définir l'environnement de travail, exécuter des actions, etc. Pour y accéder, il suffit d'appuyer sur la touche [:] à partir du mode commande. Ce mode de fonctionnement est équivalent à celui des éditeurs de texte sous OpenVMS une fois que la touche [DO] est pressée.

Dans tout ce qui suit, les commandes explicitées sont valables en mode "*commande*" à moins que celui-ci ne soit précisé. Toutes les commandes précédées par le caractère «`:`" sont accessibles à partir du "*prompt*" de "`vi`". Pour appeler le prompt, il suffit de:

1. être en mode "*commande*",
2. appuyer sur la touche [:].

C.2.3 Démarrage d'une session "`vi`"

`vi` *fichier*	Édite *fichier*.
`vi -r` *fichier*	Édite la dernière version sauvegardée de *fichier* après une sortie anormale de l'éditeur de texte. Cette option est l'équivalent de la commande "`RECOVER BUFFER`" sous l'éditeur "`LSEDIT`" sous OpenVMS.
`vi +`*n fichier*	Édite *fichier* et place le curseur directement à la ligne *n*.
`vi +` *fichier*	Édite *fichier* et place le curseur directement sur la dernière ligne.
`vi` *fichier1 fichier2* $\cdots$ *fichiern*	Édite successivement les fichiers *fichier1 fichier2* $\cdots$ *fichiern*. Le passage ne peut se faire que du précédent au suivant (donc pas de remontée possible). La commande pour passer au fichier suivant est "`:n`". Si le fichier n'a pas été sauvegardé, un message d'alarme est affiché et le passage au fichier suivant n'est pas effectué. Si, toute fois vous désirez forcer ce passage, tapez la commande "`:n!`".
`vi +/`*str fichier*	Édite *fichier* et place le curseur directement sur la première ligne contenant la chaîne "*str*".

C.2.4 Sauvegarder et quitter "`vi`"

`ZZ` ou `:wq` ou `:x`	Sortie avec sauvegarde.
`:w␣`*fichier*	Sauvegarde le contenu de l'éditeur dans le fichier nommé "*fichier*". Si aucun nom n'est donné, celui pris en compte sera celui du fichier courrant.
`:w!` *fichier*	Force la sauvegarde du fichier. Si aucun nom n'est donné, celui pris en compte sera celui du fichier courrant.
`:`*n*`,`*m*`w␣`*fichier*	Sauvegarde de la ligne "*n*" à la ligne "*m*" dans le fichier nommé "*fichier*". Si aucun nom n'est donné, celui pris en compte sera celui du fichier courrant.
`:`*n*`,`*m*`w>>`*fichier*	Sauvegarde de la ligne "*n*" à la ligne "*m*"à la fin du fichier nommé "*fichier*". Si aucun nom n'est donné, celui pris en compte sera celui du fichier courrant.
`:q`	Quitte "`vi`" sans sauvegarde. Si toutefois des modifications ont été effectuées, "`vi`" le signalera et annulera l'opération de sortie.
`:q!`	Force la sortie de "`vi`" sans sauvegarde. Si des modifications ont été effectuées, aucun message ne sera affiché et les modifications seront perdues.
`:e!`	Permet d'annuler toutes les modifications effectuées depuis la dernière sauvegarde.

ATTENTION :

Aucune sauvegarde de la version précédente n'est effectuée. Si vous sortez de "`vi`" avec sauvegarde ou bien si vous sauvegardez les modifications faites, aucune trace de votre précédent travail ne sera conservée.

C.2.5 Commandes d'état

`:.=`	Affiche le numéro de la ligne courrante.

Suite page suivante ···

`:=`	Affiche le nombre de lignes du fichier chargé dans l'éditeur.
CTRL-g	Affiche le nom du fichier traité, le numéro de la ligne courrante,le nombre total de lignes et le pourcentage relatif de la position courrante.

C.2.6 Insertion de texte

`a`	Ajoute du texte **après** la position courrante du curseur.
`A`	Ajoute du texte après la fin de la ligne courrante.
`i`	Insère du texte **avant** la position courrante du curseur.
`I`	Insère du texte au début de la ligne courrante.
`o`	Ouvre une nouvelle ligne **en dessous** de la ligne courrante.
`O`	Ouvre une nouvelle ligne **au dessus** de la ligne courrante.
CTRL-v *caract*	En mode insertion, cette commande permet d'insérer n'importe quel caractère de controle, comme ESC , un saut de page, etc. Par exemple, pour insérer le caractère ESC dans le fichier, il suffit de taper la séquence suivante **en mode insertion** : CTRL-v ESC
`:r␣`*fichier*	Lit le fichier passé en argument et l'insère après la ligne courrante.
`:`*n*`r␣`*fichier*	Lit le fichier passé en argument et l'insère après la ligne "*n*".

La figure C.1 résume les commandes précédentes.

Remarque C.3 :

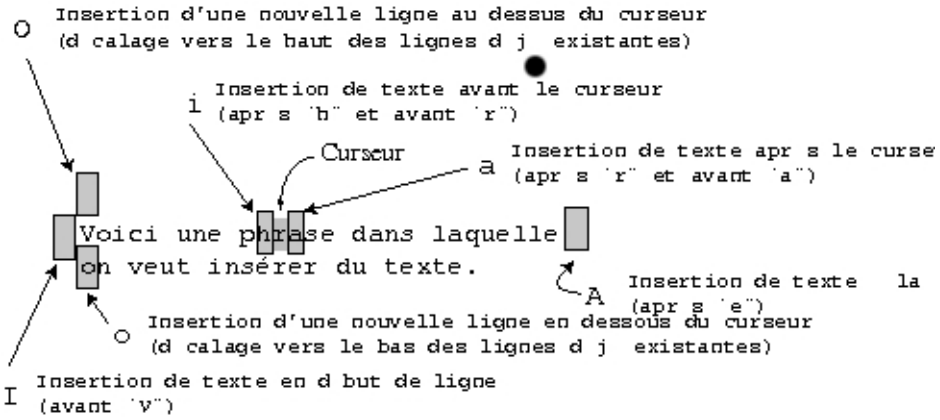


FIG. C.1 – *Résumé des commandes d'insertion de texte de "*`vi`*".*

Toutes les commandes d'insertion présentées ici, permettent d'insérer plusieurs lignes. En effet, il suffit de taper son texte au kilomètre et presser sur la touche RETURN *pour créer une autre ligne, comme tout autre éditeur de texte pleine page. Par contre, vous ne pouvez revenir en arrière que sur la même ligne que le curseur. Si vous désirez revenir sur une ligne précédente, que vous avez saisi sans sortir du mode insertion,* **il vous faudra revenir en mode commande pour vous y placer**.

C.2.7 Annuler ou répéter des commandes

`u`	Annule la dernière commande.
`U`	Restaure la ligne courrante à son état d'origine.
`n`	Répète la dernière opération de recherche "`/`" ou "`?`". (cf. section C.2.10). Cette opération s'effectue de la position courrante vers la fin du fichier.
`N`	Répète la dernière opération de recherche "`/`" ou "`?`". (cf. section C.2.10). Cette opération s'effectue **en sens inverse**, c'est-à-dire de la position courrante vers **le début du fichier**.
`;`	Répète la dernière commande de recherche "`f`", "`F`", "`t`" ou "`T`" (cf. section C.2.10.2).

Suite page suivante ···

,	Répète la dernière commande de recherche "f", "F", "t" ou "T" **en sens inverse**(cf. section C.2.10.2).
.	Répète la dernière opération de insertion/suppression ou modification de texte dans le fichier.

C.2.8 Déplacement du curseur

k ou CTRL-p ou ↑	Déplacement vers le haut.
j ou CTRL-j ou CTRL-n ou ↓	Déplacement vers le bas.
h ou CTRL-h ou BACK SPACE ou ←	Déplacement vers la gauche
l ou SPACE ou →	Déplacement vers la droite.
w ou W	Déplace le curseur au début du mot suivant. "W" permet d'ignorer la ponctuation.
b ou B	Déplace le curseur au début du mot précédent. "B" permet d'ignorer la ponctuation.
e ou E	Déplace le curseur à la fin du mot courrant. "E" permet d'ignorer la ponctuation.
0 (zéro) ou | (pipe)	Déplace le curseur à la première colonne de la ligne courrante.
n| (pipe)	Déplace le curseur à la colonne "*n*" de la ligne courrante.
^	Déplace le curseur sur le premier caractère différent de l'espace ou de la tabulation sur la ligne courrante.
$	Déplace le curseur à la fin de la ligne courrante.

Suite page suivante ···

+ ou RETURN	Deplace le curseur au début de la ligne suivante.
-	Déplace le curseur sur le premier caractère différent de l'espace ou de la tabulation de la ligne précédente.
1G	Ramène le cuseur à la première ligne.
G	Déplace le curseur à la fin du fichier.
G$	Déplace le curseur à la fin de la dernière ligne du fichier.
nG	Déplace le curseur à la n^e ligne du fichier.
(	Ramène le curseur au début de la phrase courrante. Une phrase, pour "vi" est une suite de mots séparés par des espaces ou des caractères de ponctuations et terminés par le caractère ".".
)	Déplace le curseur au début de la phrase suivante.
{	Ramène le curseur au début du paragraphe courrant. Un paragraphe, pour "vi", est une suite de phrases séparés par une ligne blanche. Cette terminologie obéit à la syntaxe de TEX et de LATEX.
}	Déplace le curseur au début du paragraphe précédent.

La figure C.2 décrit les différentes actions de ces commandes en fonction de la position du curseur dans le fichier.

Sachant que "vi" a été dédié aux développeurs UNIX, il est clair que "vi" reprend des notions du langage C. Lorsque la première colonne d'une ligne du fichier contient le caractère "{", "vi" considère tout le texte compris entre ces deux marques (texte compris entre deux "{" situés en première colonne) comme une section. Par exemple :

```
{
    Le caractere sur la ligne precedente permet de marquer une section.
    La prochaine section commencera des que le caractere { apparaitra
    a nouveau en premiere colonne dans le fichier.
{
    Ici nous sommes dans une nouvelle section. On pourrait faire l'analogie
    avec le langage C lorsque l'on declare le corps d'une
```

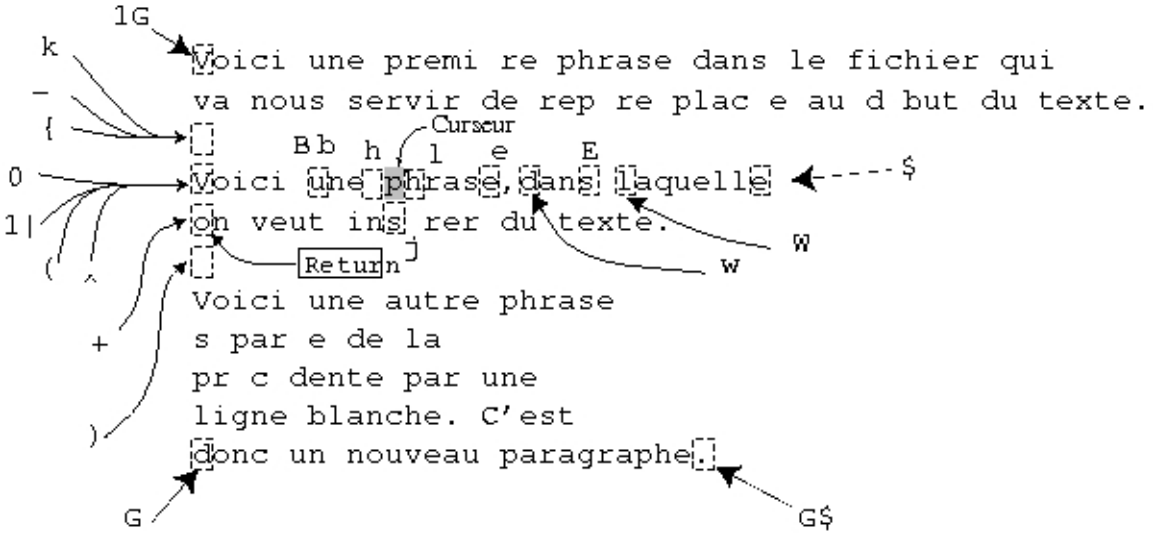


FIG. C.2 – *Commandes de déplacement du curseur avec "vi".*

```
    fonction. En effet on aura :
main()
{
    section associee au corps de la fonction main.
    Donc nous sommes ici dans la troisieme section de ce fichier texte.
}

ma_fonction()
{
    section associee au corps de la fonction ma_fonction.
    Donc nous sommes ici dans la quatrieme section de ce fichier texte.
}
```

Pour se déplacer d'une section à l'autre, "vi" dispose des commandes suivantes :

`[[`	Ramène le curseur au début de la section courrante.
`]]`	Déplace le curseur au début de la section suivante.

C.2.9 Effacement de texte

Dans toute la suite de cette section, le caractère courrant désignera celui sur lequel est positionné le curseur. De même, la ligne courrant sera celle ou se trouve le curseur.

Pour "vi", un mot est une séquence de caractères alphanumériques séparés par un ou plusieurs espaces ou tabulation ou bien encore un caractère de ponctuation. Le mot courrant désigne la chaine de caractère **commençant à la position du curseur** jusqu'à un caractère valide de délimitatio pour un mot.

CTRL – h BACK SPACE	ou	**En mode insertion**, efface le caractère précédent.
CTRL – w		**En mode insertion**, efface le mot précédent.

n`X`	Efface les "n" caractères précédents y compris le caractère courrant. Si "n" n'est pas spécifié, "`vi`" n'efface que le caractère précédent.
`xp`	Intervertit le caractère courrant avec le caractère suivant.
n`dw`	Efface les "n" mots suivants. Si "n" n'est pas spécifié, "`vi` ï¿½> détruit le mot courrant. Pour rappel, un mot est une suite de caractères séparés par un ou plusieurs espaces ou tabulation ou bien par un caractère de ponctuation.
n`db`	Efface les "n" mots précédents. Si "n" n'est pas spécifié, "`vi` ï¿½> détruit le mot courrant.
n`dd`	Efface les "n" lignes suivantes en commençant par la ligne courrante. Si «n n" n'est pas spécifié, "`vi`" ne détruit que la ligne courrante.
`:`*n*`,`*m*`d`	Détruit les lignes de "n" à "m". Si "$n \in \mathcal{N}$" ou "$m \in \mathcal{N}$", alors l'opération de destruction commence ou se termine à la ligne dont le numéro est spécifié. Si "n" ou "m" est un marqueur, l'opération de destruction commence ou se termine à la position indiquée par le marqueur (cf. section C.2.1). Si "n" ou "m" est une expression régulière délimitée par le caractère "`/`", l'opération commence à la première ligne trouvée à partir de la position courrant satisfaisant l'expression régulière ou bien se termine à la **première** ligne satisfaisant l'expression régulière se trouvant à la suite de la première ligne à effacer.
`D` ou `d$`	Détruit tout ce qui suit à partir de la position courrante du curseur jusqu'à la fin de la ligne.

Suite page suivante ···

`d`$cmd_{curseur}$	Détruit le texte à partir de la position courrante du curseur jusqu'au point indiqué par la commande de déplacement du cuseur "$cmd_{curseur}$". Par exemple, "`dG`" efface toutes les lignes à partir de la position courrante jusqu'à la fin de fichier. Cet exemple est équivalent à "`:.,$d`". Reportez-vous à la section C.2.8 pour les commandes de déplacement du curseur.

La figure C.3 décrit les différentes actions de ces commandes en fonction de la position du curseur dans le fichier.

FIG. C.3 – *Commandes de suppression de texte avec "`vi`".*

C.2.10 Recherche

C.2.10.1 Expression de recherches

`:set␣magic`	Autorise les expressions de recherche en utilisant les expressions régulières (cf. chapitre 8). Cette option est positionnée par défaut (cf. section C.2.17).
`:set␣nomagic`	N'autorise que les symboles "`^`" et "`$`" pour les expressions de recherche. Pour plus de précisions sur les options de "`vi`", reportez-vous à la section C.2.17.
`^`	Correspond au début de ligne (idem que les expressions régulières, cf. 8).

Suite page suivante ···

`$`	Correspond à la fin de ligne (idem que les expressions régulières, cf. 8).
`.`	Correspond à n'importe quel caractère (idem que les expressions régulières, cf. 8).
`\<`	Correspond à un début de mot.
`\>`	Correspond à une fin de mot.
[*str*]	Correspond à un et un seul caractère parmi ceux composant "*str*" (idem que les expressions régulières, cf. 8).
[^*str*]	Correspond à un et un seul caractère différent de ceux composant "*str*" (idem que les expressions régulières, cf. 8).
[*a*-*w*]	Correspond à un et un seul caractère entre les caractères "*a*" et "*w*" (idem que les expressions régulières, cf. 8).
*	Spécifie le nombre d'occurence qu'un caractère peut apparaître. Dans ce cas, le caractère spécifié peut apparaître un nombre quelconque de fois ($n \in \{0, \cdots, +\infty\}$).
\	Annule l'interprétation du caractère suivant.
\\	Permet de spécifier le caractère "\". En effet, "\\" implique que : – le premier "\" annule l'évaluation du caractère suivant. – le second caractère "\" doit être pris tel quel.

Remarque C.4 :

Ce tableau montre que les règles de syntaxes des expressions régulières sont largement utilisées dans "`vi`". Tout comme les commandes "`sed`" et "`awk`" (cf. chapitres 10 et 11), "`vi`" fait partie des nombreuses commandes Unix *utilisant la notion des expressions régulières.*

C.2.10.2 Opérations de recherche

`%`	Permet de localiser la parenthèse fermante ou ouvrante correspondant à celle sur laquelle est positionné le curseur. Cette fonctionnalité est opérationnelle pour "`(,)`", "`[,]`" et "`{,}`".
`f`*caract*	Recherche, dans la ligne courrante, le premier caractère "*caract*", **suivant** la position courrante du curseur.
`F`*caract*	Recherche, dans la ligne courrante, le premier caractère "*caract*", **précédant** la position courrante du curseur.
`t`*caract*	Recherche, dans la ligne courrante, le premier caractère immédiatement précédent à "*caract*", **suivant** la position courrante du curseur.
`T`*caract*	Recherche, dans la ligne courrante, le premier caractère immédiatement suivant à "*caract*", **précédant** la position courrante du curseur.
/*str* RETURN	Recherche la chaine "*str*" de la position courrante du curseur vers la **fin** du fichier.
?*str* RETURN	Recherche la chaine "*str*" de la position courrante du curseur vers le **début** du fichier.
`:set␣ic`	Ignore la différence entre les majuscules et les minuscules. Pour plus d'informations sur les options de "`vi`", reportez-vous à la section C.2.17.
`:set␣noic`	Fait la différence entre les majuscules et les minuscules. C'est le comportement par défaut. Pour plus d'informations sur les options de "`vi`", reportez-vous à la section C.2.17.

C.2.10.3 Recherche globale et substitution

`:`*n*`,`*m*`s/`*str*$_1$`/`*str*$_2$`/`*opt*	Substitue l'expression "*str*$_1$" par "*str*$_2$" dans l'espace de travail délimité par "*n*" et "*m*", c'est-à-dire sur toutes les lignes du fichier caractérisées par "*n*" et "*m*". Les options disponibles sont : "`g`" : La substitution est globale, c'est-à-dire qu'elle se répète autant de fois que nécessaire sur chaque ligne sélectionnée. Par défaut, la substitution ne s'effectue qu'**une seule fois** par ligne. "`c`" : Une confirmation est demandée avant d'effectuer toute opération de substitution. Il suffit d'appuyer sur [y] pour confirmer et sur [RETURN] ou [n] pour infirmer. "`p`" : Les lignes modifiées sont affichées. Cette commande est identique à la commande "`s`" de "`sed`" (cf. 10.3.3).
`&`	Répète l'appel à la dernière commande de substitution "`:s`".
`:g/`*str*`/`*cmd*	Exécute la commande "*cmd*" sur toutes les lignes contenant l'expression "`str`".
`:g/`*str*$_1$`/s/`*str*$_2$`/`*str*$_3$`/`	Localise la ligne contenant l'expression "*str*$_1$" et y substitue "*str*$_2$" par "*str*$_3$/".
`:v/`*str*`/`*cmd*	Exécute la commande "*cmd*" sur toutes les lignes **ne contenant pas** l'expression "*str*".

C.2.11 Indentation de texte

[CTRL]–[i] ou [TAB]	En mode "*insertion*", insère un caractère de tabulation servant à l'indentation du programme.
`:set␣ai`	Active ou déasctive l'indentation automatique (cf. section C.2.17). Si l'indentation automatique est activée, lorsque vous tapez un retour chariot, "`vi`" insère automatiquement le nombre nécessaire de tabulations afin que la nouvelle ligne commence au même niveau que la ligne précédente.

Suite page suivante ···

`:set␣sw=`*n*	Fixe, **pour l'affichage**, la taille d'une tabulation. **Attention**, cette commande ne modifie pas le contenu du fichier en remplaçant les tabulations par un certain nombre d'espaces, les caractères de tabulation (code `ASCII` 9) sont bien présents dans le fichier. "`vi`" interprètera ces caractères pour les faire correspondre à une certaine taille, c'est-à-dire à un certain nombre de colonnes. Par conséquent, si vous modifiez la taille des tabulations (par défaut toutes les 8 colonnes), d'autres éditeurs pourront donner un présentation différent de votre code source, à moins biensûr d'y modifier aussi la taille des taille des tabulations.
n`<<` ou *n*`>>`	Décale vers la gauche ("`<<`") ou vers la droite ("`>>`") les «*n*" lignes. Si "*n*" n'est pas précisé, "`vi`" décale la ligne courrante.
`<`$cmd_{curseur}$ ou `>`$cmd_{curseur}$	Permet de décaler plusieurs lignes vers la gauche ("`<`") ou vers la droite ("`>`") en fonction de la commande de déplacement du curseur "$cmd_{curseur}$" (cf. section C.2.8).

C.2.12 Copier/Coller

"`vi`" dispose d'un certain nombre de *buffer* permettant de copier ou coller un certain nombre de lignes, de mots ou de caractères qu'il sera possible de replacer n'importe où dans le fichier.Ces *buffers* sont nommés par une lettre allant de "`a`" à "`z`".

Dans toute la suite, nous désignerons l'opération "*couper*" par le fait de copier du texte dans un *buffer* et de les effacer du fichier. De même, nous désignerons l'opération "*coller*" par le fait de placer à la position courrante du curseur, le contenu d'un *buffer*.

Dans le tableau suivant, nous désignerons l'un de ces *buffers* par "*(a-z)*".

n`yy` ou *n*`Y`	Copie les "*n*" lignes à partir de la position courrante dans le *buffer* par défaut. Si "*n*" n'est pas précisé, alors seule la ligne courrante est mémorisée dans le *buffer*.

Suite page suivante ···

`y`*cmd*$_{curseur}$	Copie la partie de texte spécifiée par la commande de déplacement de curseur "*cmd*$_{curseur}$" dans le *buffer* par défaut, à partir de la position du curseur. Par exemple, "`yG`", copie le texte à partir de la position courrante du curseur jusqu'à la fin de la ligne dans le *buffer*.
"*(a-z)n*`yy`	Copie les "*n*" lignes à partir de la position courrante dans le *buffer* spécifié. Si "*n*" n'est pas précisé, alors seule la ligne courrante est mémorisée dans le *buffer*. Par exemple, "`""a10yy`" copie les dix lignes suivantes (ligne courrante comprise) dans le *buffer* "`a`".
"*(a-z)n*`dd`	*Coupe* les "*n*" lignes à partir de la position courrante dans le *buffer* spécifié. Si "*n*" n'est pas précisé, alors seule la ligne courrante est mémorisée dans le *buffer*. Par exemple, "`""a10dd`" *coupe* les dix lignes suivantes (ligne courrante comprise) dans le *buffer* "`a`".
`p` (minuscule)	*Colle* le contenu du *buffer* par défaut **après**ï¿½a position courrante du curseur. Après cette opération, le *buffer* est vidé.
`P` (majuscule)	*Colle* le contenu du *buffer* par défaut **avant**ï¿½a position courrante du curseur. Après cette opération, le *buffer* est vidé.
"*(a-z)n*`p`ï¿½	*Colle* le contenu du *buffer* spécifié **après**ï¿½a position courrante du curseur. Après cette opération, le *buffer* est vidé.
"*(a-z)n*`P`	*Colle* le contenu du *buffer* spécifié **avant**ï¿½a position courrante du curseur. Après cette opération, le *buffer* est vidé.

C.2.13 Modifier du texte

`r`*caract*	Substitue le caractère courrant par le caractère "*caract*".

Suite page suivante ···

R*texte* ESC		Réécrit le texte saisi "*texte*" par dessus ce qui est déjà présent. Ceci équivaut au mode "*surimpression*" des éditeurs pleine-page classiques comme "`EVE`" ou "`LSEDIT`" sous OpenVMS. Le remplacement de texte se termine par la touche ESC . En mode "*insertion*", le texte saisi est inséré à la position courrante du curseur. En mode "*surimpression*", tout ce qui est saisi vient s'écrire par dessus le texte déjà existant.
s*texte* ESC		Substitue le caractère courrant avec le texte saisi "*texte*". L'insertion du nouveau texte se termine par ESC . Par conséquent, après la saisie de la commande "`s`", "`vi`" passe en mode "*insertion*".
S*texte* ESC cc*texte* ESC	ou	Efface la ligne courrante et la substitue par le texte saisi "*texte*". L'insertion du nouveau texte se termine par ESC . Par conséquent, après la saisie de la commande "`s`" ou "`cc`", "`vi`" passe en mode "*insertion*".
cw*texte* ESC		Change le mot courrant par le texte saisi "*texte*". L'insertion du nouveau texte se termine par ESC . Par conséquent, après la saisie de la commande "`cw`", "`vi`" passe en mode "*insertion*".
C*texte* ESC		Change le reste de la ligne à partir de la position courrante par le texte saisi "*texte*". L'insertion du nouveau texte se termine par ESC . Par conséquent, après la saisie de la commande "`C`", "`vi`" passe en mode "*insertion*".
c$curs_{cmd}$*texte* ESC		De façon plus générale, la commande "`c`" suivie d'une commande "`vi`" de déplacement de curseur (cf. section C.2.8), change le texte correspondant avec ce qui a été saisi, jusqu'à ce que la touche ESC soit pressée. Par conséquent, après la saisie de la commande "c$curs_{cmd}$", "`vi`" passe en mode "*insertion*".
J		Rassemble la ligne courrante et la ligne suivante sur une seule et même ligne.

Suite page suivante ···

*n*J	Rassemble les "*n*" lignes, y compris la ligne courrante, sur une seule et même ligne.

C.2.14 Placement direct du curseur et ajustement du texte à l'écran

H	Ramène le curseur sur la première ligne de l'écran, ligne qui n'est pas forcément la première ligne du fichier.
*n*H	Ramène le curseur sur la n^e ligne de l'écran, ligne qui n'est pas forcément la n^e ligne du fichier.
M	Amène le curseur au milieu de l'écran.
L	Amène le curseur sur la dernière ligne de l'écran, ligne qui n'est pas forcément la dernière ligne du fichier.
*n*L	Amène le curseur sur la n^e ligne de l'écran en partant du bas de l'écran.
CTRL-e	Décale l'affichage d'une ligne vers le haut.
CTRL-y	Décale l'affichage d'une ligne vers le bas.
CTRL-u	Décale l'affichage d'une demi page vers le haut.
CTRL-d	Décale l'affichage d'une demi page vers le bas.
CTRL-b	Décale l'affichage d'une demi page vers le haut. Cette fonction est aussi disponible avec la touche "*Page Précédente*".
CTRL-f	Décale l'affichage d'une page vers le bas. Cette fonction est aussi disponible avec la touche "*Page Suivante*".
CTRL-l (lettre "l")	Rafraîchit l'écran. Cette fonction équivaut à CTRL-w avec les éditeurs «EVE" ou "LSEDIT" sous OpenVMS.

Suite page suivante ···

`z` RETURN	Décale l'affichage de telle sorte que la ligne courrant devienne la première ligne affichée en haut de l'écran.
n`z` RETURN	Décale l'affichage de telle sorte que la n^e ligne affichée en partant du haut, devienne la première ligne affichée à l'écran.
`z.`	Décale l'affiche de telle sorte que la ligne courante devienne celle du milieu de l'écran.
n`z.`	Décale l'affichage de telle sorte que la n^e ligne affichée en partant du haut, devienne celle du milieur de l'écran.
`z-`	Décale l'affichage de telle sorte que la ligne courrante devienne la dernière ligne affichée en bas de l'écran.
n`z-`	Décale l'affichage de telle sorte que la n^e ligne affichée en partant du haut, devienne la dernière ligne affichée en bas de l'écran.

Remarque C.5 :

L'ensemble de ces commandes s'adaptent en fonction de la taille de l'écran, c'est-à-dire du nombre de lignes et de colonnes.

C.2.15 Interaction avec le *shell*

`:!`*commande*	Exécute la commande dans un sous processus. Cette commande peut être n'importe quelle commande *shell*, y compris un appel à une autre session `"vi"`. Lors de la saisie de la commande au niveau du *prompt* de `"vi"`, il est possible d'utiliser les caractères spéciaux suivants : `%` correspond au nom du fichier en train d'être édité. `#` correspond au nom du dernier fichier édité.

Suite page suivante ···

	L'interpréteur de commandes utilisé par défaut est le Bourne Shell ("`/bin/sh`". Cette valeur peut être modifiée grâce à la commande "`:set␣shell=`*chemin*" (cf. C.2.17).
`:!!`	Réexécute la dernière commande *shell*, c'est-à-dire la dernière commande "`:!`*commande*".
`:r!␣`*commande*	Inclut dans le fichier à la position courrante du curseur la sortie standard de la commande "*commande*".
`:f␣`*nouv_fichier*	Renomme le fichier courrant sous le nouveau nom "*nouv_fichier*". Dans ce cas de figure, le fichier édité change de nom sur le disque mais aussi pour "`vi`". Ainsi, toute opération future de sauvegarde se fera sous le nouveau nom de fichier.
`:w␣!`*commande*	Sauvegarde le fichier courrante et envoie son contenu sur l'entrée standard de la commande "*commande*".
`:cd␣`*répertoire*	Change le répertoire par défaut de "`vi`". Si aucun répertoire n'est spécifié, le contenu de la variable d'environnement "`HOME`" est utilisé.
`:sh`	Démarre un sous processus de "`vi`" dans lequel sera exécuté un nouvel interpréteur de commandes. Pour revenir à "`vi`", il suffit de taper la commande "`exit`" ou CTRL-d, commande ou séquence de touche permettant de terminer une session *shell* (cf. 1.2 et 7.7).
`:so␣`*fichier*	Lit et exécute les commandes *shell* présentes dans le fichier "*fichier*".
`!`*cmd*$_{curseur}$`␣`*commande*	Envoie le texte sélectionné par la commande de déplacement de curseur "*cmd*$_{curseur}$" à la commande *shell* "*commande*". Le texte correspondant à la commande de déplacement du curseur est remplacé par la sortie standard de la commande. Par exemple :

Suite page suivante ···

`!w␣ls␣-l` RETURN	envoie le mot situé au niveau du curseur à la commande "`ls -l`". Le mot en question est remplacé par le résultat de la commande.
`!}sort` RETURN	Sélectionne le texte de la position courrante du curseur jusqu'à la fin du paragraphe et le trie grâce à la commande "`sort`". Le texte sélectionné est remplacé par le résultat du tri.

C.2.16 Macros et abréviations

`:map␣`*touche*`␣`*séquence*$_{cmds}$	Associe la séquence de commandes "*séquence*$_{cmds}$" à la touche "*touche*". Ainsi, dès que cette touche est appuyée, la séquence de commandes est exécutée.
`:map`	Affiche l'ensemble des définitions effectuées, c'est-à-dire l'ensemble des "*macros*" de "`vi`" créées grâce à la commande précédente.
`:unmap␣`*touche*	Détruit l'association entre une touche et la séquence de commandes précédemment faite. Cette commande détruit donc la macro associée à la touche "*touche*".
`:ab␣`*chaine*$_1$`␣`*chaine*$_2$	Permet de définir une abréviation. Lorsque "*chaine*$_1$" est saisi, "`vi`" la substitue par "*chaine*$_2$".
`:ab`	Affiche l'ensemble des abréviations définies.
`:una␣`*chaine*	Détruit l'abréviation "*chaine*".

La commande "`:map`" permer de définir des séquences de commandes ou "*macros*" "`vi`". En effet, par définition, une macro, comme en langage C ou n'importe quel logiciel, est une série de commandes ou d'actions de base regroupées sous un nom et appelable par l'utilisateur.

Si l'option "`timeout`" est positionnée (cf. section C.2.17), toute exécution de macros ne peut dépasser une seconde. Par conséquent, si vous utilisez des macros importantes, désactivez cette option.

Sachant que les commandes "`vi`" utilisent des caractères de contrôle en mode

commande, il est possible de les insérer dans la définition des macros grâce à la séquence CTRL-V (cf. section C.2.6). De même, le caractère "" est utilisé dans les commandes "vi" (cf. sections C.2.7 et C.2.12). Par conséquent, s'il doit être utilisé dans une macro dans un autre cadre que celui d'une commande le référençant, il doit être précédé du caractère "\".

Remarque C.6 :

Les touches inutilisées sous "vi" sont :

- *les touches de fonction,*
- *les touches "K", "V", "g", "q", "v", "*" et "=".*

Exemple C.1 :

:map␣v␣/Je␣CTRL-V ESC dwiTu␣CTRL-V ESC ESC

Lorsque la touche V est appuyée **en mode "***commande***"**, les actions suivantes sont exécutées :

1. Recherche de la chaine "Je␣" ("/Je␣ESC "),
2. Efface le mot courrant ("dw"),
3. Insère la chaine "Tu␣ESC " ("iTu␣CTRL-V ESC "),
4. Termine le mode "*insertion*" ("ESC ").

Dans la section C.2.17, vous trouverez un ensemble d'abréviations définies par défaut dans "vi". La commande ":ab␣*chaine*$_1$␣*chaine*$_2$" permet de définir celles qui vous seront propres.

C.2.17 Commandes de configuration de "vi"

"vi" dispose d'un certain nombre d'options permettant de paramètrer son fonctionnement et l'environnement de travail. Celles-ci sont regroupées en trois catégories :

les options de type "*bascule*" :
: Dans ce cas de figure, l'option correspond à deux état possible :
  - activé,
  - désactivé.

les options de type "*numérique*" :
: À ces options sont associées des valeurs numériques.

les options de type "*chaine*" :
: Comme pour le type précédent, l'option est associée à une valeur de type "*chaine de caractères*".

:set	Affiche toutes les options **qui ont été modifiées par rapport à la configuration par défaut**, et la valeur ou l'état associé.
:set␣all	Affiche toutes les options et la valeur ou l'état associé.

Suite page suivante ···

`:set␣`*option*	Si l'option est de type "*bascule*", elle est armée. Si elle est d'un autre type ("*chaine*" ou "*numérique*", la valeur associée est affichée.
`:set␣no`*option*	Ce cas de figure n'est valable que si l'option est de type "*bascule*". Dans ce cas, elle est désactivée.
`:set␣inv`*option*	Inverse l'état d'une option de type «n *bascule*".
`:set␣`*option*`=`*value*	Fixe la valeur associée à une option de type "*chaine*" ou "*numérique*".
`:set␣`*option*`?`	Affiche la valeur ou l'état associée à l'option spécifiée.

Option	Abbréviation	Type	Défaut	Description
`autoindent`	`ai`	bascule	non	Lorsque cette option est activée, les tabulations sont automatiquement insérées en mode "*insertion*" afin que les lignes soient alignées sur une même colonne. Pour revenir en arrière, au niveau de l'alignement des colonnes, il suffit de repasser en mode "*commande*" et positionner le curseur à la colonne souhaitée.
`directory`	`dir`	chaine	`/tmp`	Spécifie le répertoire temporaire de `"vi"` pour stocker ses informations. Ce répertoire contiendra, entre autre, un fichier que vous pourrez utiliser en cas d'interruption de la session `"vi"` et retrouver toutes les modifications que vous aurez effectuées (cf. option `"-v"` à la section C.2.3).
`errorbells`	`eb`	bascule	non	Précède tous les messages d'erreur par un "*bip*".
`ignorecase`	`ic`	bascule	non	Ignire les majuscules/minuscules lors des opérations de recherche.

Suite page suivante ···

Suite de la page précédente ···

Option	Abbréviation	Type	Défaut	Description
`insertmode`	`im`	bascule	non	Démarre la session "`vi`" en mode "*insertion*". Par défaut, "`vi`" démarre en mode "*commande*".
`lines`		nombre	25	Spécifie le nombre de lignes à afficher à l'écran. Par défaut, le nombre de lignes est "25" ou, plus précisément le nombre de lignes affichable par le terminal. Ce paramètrage est accessible par la commande "`stty(1)`" ou "`resize(1)`". La première permet de spécifier les paramètres du terminal manuellement. La seconde interroge le terminal pour obtenir ces caractéristiques.
`list`		bascule	non	Affiche tous les caractères invisibles. Lorsque cette option est activée, nous aurons, entre-autre : `^I` Tabulation `$` Fin de ligne
`magic`		bascule	oui	Autorise les expressions régulières pour les opérations de recherche.
`makeprg`	`mp`	chaine	`make`	Spécifie le nom de l'utilitaire permettant de générer les dépendances entre les différents fichiers source et les exécutables à générer. Par défaut, la commande UNIXï¿½tilisée est "`make(1)`".
`number`	`nu`	bascule	non	Affiche le numéro de chaque ligne.
`readonly`	`ro`	bascule	non	Le fichier courrant passe en lecture seule. Il est possible de l'éditer, mais la sauvegarde n'est pas accessible. Cependant, si les droits d'accès au fichier le permettent, la commande "`:w!`" force la sauvegarde (cf. section C.2.4). Ce mode est positionné par défaut si les droits d'accès au fichier n'autorisent que la lecture.

Suite page suivante ···

Suite de la page précédente ···

Option	Abbréviation	Type	Défaut	Description
`remap`		bascule	oui	Autorise l'appel de macros à l'intérieurs de macros (*macros* récursives).
`report`		nombre	2	Spécifie le nombre de lignes minimales pour que "`vi`" puisse afficher ses informations.
`revins`	`ri`	bascule	non	Au lieu d'insérer du texte de la gauche vers la droite (sens normal pour l'alphabet romain) mais de la droite vers la gauche.
`ruler`	`ru`	bascule	non	Affiche la position courrante du curseur (ligne×colonne) dans la zone où "`vi`" inscrit ses informations.
`scroll`		nombre	12	Précise le nombre de lignes à prendre en compte pour les commandes CTRL-u, CTRL-d, CTRL-b et CTRL-f (cf. section C.2.14).
`scrolljump`	`sj`	nombre	1	Précise le nombre de lignes minimal pour les passages aux écrans précédents et suivants.
`shell`	`sh`	chaine	`sh`	Précise l'interpréteur de commande à utiliser par défaut pour les inter-actions avec le "*shell*" (cf. section C.2.15).
`shiftwidth`	`sw`	nombre	8	Précise la taille des décalages pour les commandes "<" et ">" (cf. section C.2.11).
`shortname`	`sn`	bascule	non	Assure la compatibilité avec des systèmes de fichier type MS-DOS, c'est-à-dire des noms de fichiers en majuscules, ne comportant que huits caractères au maximum avec une extension d'au plus trois caractères.

Suite page suivante ···

Suite de la page précédente ···

Option	Abbréviation	Type	Défaut	Description
`showmatch`	`sm`	bascule	non	Si cette option est activée, "`vi`" indique à l'utilisateur quelle est la *parenthèse* ouvrante correspondante dès qu'il saisit l'un des caractères suivants : – «`)`" (caractère associé "`(`"), – "`}`" (caractère associé "`{`"), – "`]`" (caractère associé "`[`"),
`showmode`	`smd`	bascule	oui	Si cette option est active, "`vi`" indique le mode courrant. – si le mode courrant est le mode "*insertion*", "`vi`" inscrit "`- INSERT -`", – si le mode courrant est le mode "*commande*", **aucune information** n'est spécifiée dans la ligne d'état. **Attention**, dans certains cas, cette option n'est pas positionnée par défaut. De même, pour certaines versions de "`vi`", les informations affichées sont plus explicites : – "`vi`" inscrit "`- INSERT -`" si l'utilisateur insère du texte **avant** le curseur (commandes i, I, O, etc. – cf. section C.2.6), – "`vi`" inscrit "`- INSERT -`" si l'utilisateur insère du texte **après** le curseur (commandes a, A, o, etc. – cf. section C.2.6).
`sidescroll`	`ss`	nombre	0	Nombre de colonnes minimales pour le *scrolling* horizontal.
`tabstop`	`ts`	nombre	8	Précise la taille des tabulations.
`term`		chaine		Type de terminal sur lequel l'utilisateur travaille. "`vi`" prend, par défaut, le contenu de la variable d'environnement du *shell* : "`TERM`".

Suite page suivante ···

Suite de la page précédente ···

Option	Abbréviation	Type	Défaut	Description
`terse`		bascule	non	Utilise les messages d'erreurs abrégés.
`textauto`	`ta`	bascule	oui	Détecte les caractères utilisés pour séparer les lignes d'un fichier. "`vi`" positionne alors automatiquement l'option "`textmode`". En effet, – sous MS-DOS, chaque ligne est séparée par les deux caractères *carriage-return–line-feed* (respectivement `ASCII`(13) et `ASCII`(10)), – sous UNIX, chaque ligne n'est séparée que par le caractère *line-feed* (`ASCII`(10)), – sous MacOS, chaque ligne n'est séparée que par le caractère *carriage-return* (`ASCII`(13)).
`textmode`	`tx`	bascule	non	Utilise les caractères de fin de ligne de MS-DOS.
`textwidth`	`tw`	nombre	0	Spécifie la longueur maximale d'une ligne en mode "*insertion*".
`timeout`		bascule	oui	Prend en compte la valeur de la temporisation fixée par l'option "`timeoutlen`" (ou "`tm`" pour l'exécution des macros "`vi`". Ainsi, aucune macros ne pourra s'exécuter plus de "`timeoutlen`" millisecondes.
`timeoutlen`	`tm`	nombre	1000	Fixe la valeur de la temporisation pour l'option "`timeout`". La valeur est exprimée en **millisecondes**.
`visualbell`	`vb`	bascule	non	Au lieu d'avoir un signal sonore, "`vi`" fait flasher l'écran.

Suite page suivante ···

Suite de la page précédente ···

Option	Abbréviation	Type	Défaut	Description
`warn`		bascule	oui	Affiche un message si, lors d'une sortie de l'éditeur, le fichier n'a pas été sauvegardé. Typiquement le message sera "`No write since last change.`".
`wildchar`	`wc`	nombre	TAB	Touche ou caract'ere utilisé pour compl'eter automatiquement les nom de fichiers. Le nombre spécifié correspond au code `ASCII` du caract'ere concerné. Le comportement par défaut est similaire à celui observable avec "`tcsh`" et "`bash`".
`wrap`		bascule	oui	Lorsque cette option est active, si la saisie dépasse la largeur maximale de la fenêtre d'affichage, un retour automatique à la ligne est effectué **sans pour autant avoir le caractère RETURN inséré dans le texte**. Si cette option n'est pas active, l'affichage se décalera horizontalement.
`wrapmargin`	`wm`	nombre	0	Le retour à la ligne suivante s'effectue à partir de la colonne "*nombre de colonnes de la fenêtre*" - "*valeur associée à l'option* `wrapmargin`". Cette opération s'exécute si l'option "`wrap`" est active.

C.2.18 Fichier d'initialisation de "`vi`" : "`.exrc`"

L'ensemble des commandes de "`vi`" permettant de définir son environnement de travail, c'est-à-dire l'ensemble des commandes appelable à partir du *prompt* de l'éditeur (caractère "`:`") sont mémorisables dans un fichier de configuration : le fichier "`.exrc`" situé dans le répertoire de connexion de l'utilisateur (répertoire contenu dans la variable d'environnement "`HOME`").

Toutes ces commandes, vues dans les sections précédentes peuvent y être placée **sans les faire préfixer du caractère "`:`"**.

Par exemple :

```
set number
set autoindent
map v d$
```

permet, pour toutes les sessions "`vi`" qui seront lancées :

- d'afficher les numéros des lignes,
- de faire de l'indentation automatique,
- de définir la macro "`v`".

Le nom de ce fichier est paramètrable grâce à la variable d'environnement "`EXRC`" qui contiendra le chemin absolu et le nom du nouveau fichier de configuration.

Par conséquent, à chaque nouvelle session "`vi`", l'éditeur suivra le processus suivant :

1. "`vi`" regarde si la variable d'environnement "`EXRC`" est définie. Si oui, il mémorise le nouveau nom du fichier de configuration. Si non, il prendra la valeur par défaut ("`$HOME/.exrc`").
2. "`vi`" regarde si le fichier de configuration existe et est accessible en lecture. Si oui, il charge son contenu. Si non, aucune opération n'est effectuée.

C.3 Commandes de base de l'éditeur `emacs`

C.3.1 Introduction

"`emacs`" est un éditeur *pleine page* utilisable notamment sur UNIX, OpenVMS, MS-DOS, Windows, MacOS, etc.

Il fournit des fonctionnalités puissantes :

- par des combinaisons de touches (notamment par emploi des touches "CTRL " et "ESC "),
- des menus déroulants accessibles via la souris, sur des terminaux graphiques ou par une séquence de touches, sur les terminaux `ASCII`.

Par exemple, "`emacs`" permet la recherche ou le remplacement d'une chaîne de caractères dans un programme ou la visualisation de plusieurs fichiers en même temps par utilisation de clés de fonctions prédéfinies.

Le tableau C.21 donne une liste des commandes de base de l'éditeur.

Fonctionnalité	Commande / Touches
Appel de "`emacs`"	`emacs` *fichier*
Quitter "`emacs`"	CTRL-X CTRL-C
Sauvegarder le fichier courant sous le même nom	CTRL-X CTRL-S [a] ou CTRL-X CTRL-W puis RETURN
Sauvegarder le fichier courant sous un nom différent	CTRL-X CTRL-W *nouveau_nom* puis RETURN
Documentation en ligne	CTRL-h

TAB. C.21 – *Commandes de base de "`emacs`"*

[a] cf. remarque C.3.3

Remarque C.7 :

Si un fichier sauvegardé écrase un fichier préexistant, ce dernier sera automatiquement sauvegardé sous un fichier de même nom suivi du caractère "~".

Remarque C.8 :

Quand un enregistrement du fichier en entrée est plus long que la ligne d'écran, la ligne courante se termine par le caractère "\" et l'enregistrement continue sur la ligne suivante.

C.3.2 Organisation de l'écran

L'organisation de l'écran d'"`emacs`" se décompose de la façon suivante :

Position du curseur
: C'est à partir de ce point que seront executées les commandes "`emacs`".

Echo area
: Zone en bas de l'écran où sont affichés les messages relatifs à des commandes (sauvegarde, messages d'erreur, intervention utilisateur). Cette zone correspond au *buffer* "`LSE$MESSAGE`" de l'éditeur "`LSEDIT`" d'OpenVMS.

Affichage du *buffer*
: Cette ou ces zones permettent de visualiser le contenu du ou des fichiers à éditer. Elles correspondent aux *buffers* associés à des fichiers de l'éditeur "`LSEDIT`" d'OpenVMS.

La barre associée à un *buffer* contient les informations suivantes :

- Elle indique si le *buffer* courrant a été modifié ou non. Si le buffer n'a pas été modifié, elle commencera par :

  `---Emacs:` *fichier*

  Dans le cas contraire, elle commencera par :

  `-**-Emacs:` *fichier*
- Chaque *buffer* peut être associé à un langage de programmation. Cette opération est effectuée automatiquement lors de l'ouverture du fichier après son analyse par "`emacs`". Si tel est le cas, le langage associé sera affiché sous la forme suivante :

  `(language)`
- Enfin, "`emacs`" indiquera la position courrante du curseur dans le fichier (numéro de ligne et de colonne) et sa position relative (en pourcentage).

Pour plus de précisions sur la notion de "*buffers*", reportez-vous à la section C.3.4.

C.3.3 Clés de Fonction

Une clé de fonction peut être :

- une combinaison simple de la forme CTRL–X ; par exemple CTRL–r pour une recherche arrière.
- une combinaison multiple de la forme CTRL–X CTRL–*y* ; par exemple, pour quitter "`emacs`", il faudra faire la séquence de touches suivante "CTRL–X" puis "CTRL–c". Par convention, les clés de fonction multiples commencent par l'une des séquences suivantes :
  - CTRL–X

- CTRL-c
- CTRL-h

Tout début de commande multiple (ou passage d'argument) peut être annulé en cours de saisie par la commande CTRL-g. Ceci est aussi valable lors du passage d'arguments par l'utilisateur.

Remarque C.9 :

Il existe un certain nombre de clés de fonctions prédéfinies, mais l'utilisateur peut définir ses propres clés de fonctions, soit parce que la fonctionnalité n'existe pas sous "`emacs`", soit parce que la séquence de touche à laquelle elle est normalement attribuée est inaccessible sur certaines types de terminaux passifs, par exemple CTRL-s pour une recherche, est inaccessible sur les consoles VT).

"`emacs`" est doté d'un certain nombre de fonctions prédéfinies, qui sont en fait des associations entre des touches de fonctions et des fonctions standards reconnus par l'éditeur de texte, par exemple "`scroll-down`", "`kill-word`", "`search-forward`", etc. Il existe un fichier d'initialisation, où "`emacs`" va lire les définitions utilisateurs avant toute entrée dans l'éditeur. Ce fichier s'appelle "`.emacs`" et est situé dans votre répertoire de connexion. Il est basé sur le langage "*lisp*", langage de programmation utilisé pour l'ensemble des fichiers de configuration d'"`emacs`". Ce langage était, à l'origine, associé à l'intelligence artificielle et a été le précurseur de la programmation objet.

Exemple C.2 :

Exemple de fichier "`.emacs`"

```
(define-key global-map "\C-xf" 'isearch-forward)
(define-key global-map "\C-xl" 'goto-line)
```

La première ligne associe la séquence CTRL-x f à "*recherche avant*". La seconde ligne associe la séquence CTRL-x l à "*positionnement à une ligne*".

Attention :

Si la clé de fonction est déjà affecté par le système à une autre fonction, l'affectation précédente est écrasée par l'affectation utilisateur.

C.3.4 Notion de buffers

C.3.4.1 Introduction

Chaque fichier édité est stocké dans un buffer, et au cours d'une session peut être rappelé à tout moment, par menu déroulant ou par la commande CTRL-x b *nom_de_buffer*. Un buffer a en général le même nom que le fichier qu'il représente.

La commande permettant de connaître la liste des buffers est CTRL-x CTRL-b. Une commande peut être passée en tête de chaque ligne pour sélectionner un buffer (mode pleine-page, mode fenêtre, etc.). Une astérisque en

colonne 1 indique que le buffer correspondant a été modifié, un "`%`" indique un accès en mode lecture uniquement. Pour avoir la liste des fonctionnalités disponibles, taper [?] en tête de ligne.

La commande permettant de supprimer un buffer est [CTRL]–[x] [k]. Pour supprimer plusieurs buffers, activer la liste des buffers (commande [CTRL]–[X] [CTRL]–[B]), taper [d] au début des lignes indiquant les buffers à supprimer. La commande sera effective lorsque vous taperez [x].

Si vous créez un nouveau buffer ([CTRL]–[x] [b]), il n'y aura aucun lien entre le buffer ouvert et un éventuel fichier disque portant le même nom. Encore une fois, un buffer ne représente qu'un espace de travail.

Remarque C.10 :

Supprimer un buffer n'entraîne pas suppression du fichier disque. Seule la copie du fichier dans l'espace de travail "`emacs`" est supprimée.

C.3.4.2 Les "*mini-buffers*"

Le "*mini-buffer*" est une facilité "`emacs`" pour lire les arguments d'une commande (nom de fichier, commande interactive nécessitant une réponse de l'utilisateur, etc.). Il est par exemple utilisé lorsque l'utilisateur désire ouvrir un nouveau "*buffer*" associé à un fichier déjà existant (commande [CTRL]–[x] [f]). Un nom de fichier devrait suivre cette commande. Or, si l'utilisateur saisit simplement [CTRL]–[x] [f], "`emacs`" réclame le nom de fichier dans la ligne d'état situé en bas de l'écran.

En cas d'hésitation sur l'argument à entrer dans le "*mini-buffer*", une assistance permet d'aiguiller l'utilisateur :

- "`?`" affiche l'utilisation des noms possibles compte tenu de ce qui a déjà été saisi. La fenêtre affichée est accessible et les commandes "`emacs`" sont actives (*scrolling* par exemple) ainsi que les commandes de sélection de "*buffer*" vues précédemment.
- [TAB] complète le texte dans la mesure du possible.
- [SPACE] complète le texte dans la mesure du possible mais limité à un seul mot.

C.3.5 Utilisation de l'aide

L'aide est activé par [CTRL]–[H] suivi d'une lettre ou d'un signe déterminant la nature de l'aide.

Le tableau C.22 liste les principales commandes accédant aux rubriques de l'aide.

Séquence de touche	Rubrique / Fonction
CTRL h ?	Liste des rubriques d'aide.
CTRL H *chaîne* A	Liste des commandes contenant le mot "*chaîne*".
CTRL h b	Liste des raccourcis clavier disponibles.
CTRL h f *fonction*	Description d'une fonction "`emacs`".
CTRL h n	Nouveautés de la version actuelle d'"`emacs`".
CTRL h t	Tutoriel

TAB. C.22 – *Principales commandes accédant aux rubriques d'aide d'"`emacs`".*

C.3.6 Utilisation de quelques commandes

C.3.6.1 Déplacement dans le "buffer"

Séquence de touche	Description
CTRL a	Début de ligne
CTRL e	Fin de ligne
CTRL f	Déplacement un caractère à droite
CTRL b	Déplacement un caractère à gauche
ESC f	Déplacement un mot à droite
ESC b	Déplacement un mot à gauche
CTRL n	Déplacement une ligne en dessous
CTRL p	Déplacement une ligne au dessus
ESC <	Déplacement début de fichier
ESC >	Déplacement fin de fichier
CTRL v	Déplacement d'une page vers le bas
ESC v	Déplacement d'une page vers le haut
ESC a	Déplacement début phrase courante
ESC e	Déplacement fin phrase courante
ESC k	Destruction jusqu'à la fin de la phrase courante.

C.3.6.2 Recherche / Remplacement de caractères

Séquence de touche	Description
CTRL s	Recherche vers le bas
CTRL r	Recherche vers le haut
ESC %	Remplacement par une chaîne (vers le bas).

C.3.6.3 Réponses possibles pour la recherche et le remplacement de caractères

Séquence de touche	Description

Suite de la page précédente ···

Séquence de touche	Description
DEL	Positionnement à l'occurence suivante, sans remplacement.
.	Remplacement occurrence courante et arrêt du processus.
!	Remplacement occurences restant jusqu'à la fin du texte.
?	Affichage des réponses possibles.

C.3.6.4 Couper / Copier / Coller

Marquage de la région

Quelques commandes de `"emacs"` gèrent sur une portion (à définir) du buffer. Par exemple, la copie ou suppression d'un bloc de texte nécessite préalablement de définir le texte à couper. La région sera l'espace entre le point du début de marquage et la position actuelle du curseur.

Définition du bloc de texte sur lequel va s'opérer la commande

Séquence de touche	Description
CTRL–SPACE ou CTRL–0	Marquage de début de bloc
CTRL–x CTRL–x	Passage position curseur / Début de marquage
CTRL–x h	Activer le buffer entier comme région.
ESC–h	Activer le paragraphe courant comme région.

Couper / Coller sur une région

Séquence de touche	Description
CTRL–w	Destruction de la région.
CTRL–y	Insertion du texte coupé. Le curseur est positionné après le texte inséré.
CTRL–u CTRL–y	Insertion texte coupé. Le curseur est positionné; avant le texte inséré.

Couper / Coller sur d'autres éléments

Séquence de touche	Description
CTRL-d	Suppression du caractère sous le curseur.
DEL	Suppression caractère avant le curseur.
CTRL-k	Suppression jusqu'à la fin de la ligne.
CTRL-u *n* CTRL-k	Suppression de "*n*" lignes .
ESC-d	Suppression jusqu'à la fin du mot.
ESC-k	Suppression jusqu'à la fin de la phrase.

Autres opérations sur une région

Séquence de touche	Description
CTRL-x CTRL-u	Conversion de la région en majuscules.
CTRL-x CTRL-l	Conversion de la région en minuscules.
CTRL-x TAB	Indentation d'une région.

ATTENTION :

- Il existe un seul espace de stockage des textes coupés pour l'ensemble des buffers. C'est ce qui permet, entre autres, de copier un bloc de texte d'un buffer et le dupliquer dans un autre buffer.

- Si plusieurs commandes de coupure de texte s'enchaînent sans qu'une autre commande ne vienne interférer, les textes coupés seront stockés ensemble, et une commande CTRL-Y insèrera l'ensemble des textes coupés successifs. Par contre, si une commande a été executée entre deux coupures, y compris un déplacement, la prochaine insertion de texte concernera le dernier texte coupé.

- Un texte coupé précédemment peut être inséré, même si ce n'est pas le dernier texte coupé. La procédure à suivre est la suivante :
  1. CTRL-Y
  2. ESC-y jusqu'à ce que le texte désiré soit inséré à l'écran.
     ou
     CTRL-u *n* CTRL-Y (Insertion du texte coupé "*n*" fois précédemment).

C.3.6.5 Opérations sur les fenêtres

"`emacs`" peut partager l'écran en deux ou plusieurs fenêres qui affichent plusieurs parties d'un même buffer ou de plusieurs buffers. Il n'y a pas identification

entre fenêtre et buffer. Une fenêtre peut être détruite, et le buffer correspondant être encore actif.

Si un même buffer apparait dans plusieurs fenêtres, les modifications effectuées dans une fenêtre sont répercutées dans les autres. Les commandes les plus fréquentes sont explicitées dans le tableau suivant.

Séquence de touche	Description
CTRL-x 2	Partage la fenêtre courante en 2 fenêtres horizontales.
CTRL-x o	Sélection d'une autre fenêtre.
ESC CTRL-v	Sélection de la fenêtre suivante.
CTRL-x 4 b *buffer*	Sélection d'un buffer dans une autre fenêtre.
CTRL-x 4 f *buffer*	Sélection d'un fichier dans une autre fenêtre.
CTRL-x 4 d *directory*	Sélection du contenu d'un répertoire dans une autre fenêtre.
CTRL-x ^	Agrandissement de la fenêtre courante.
CTRL-x 0	Suppression de la fenêtre courante.
CTRL-x 1	Suppression toutes fenêtres sauf la fenêtre courante.

C.3.6.6 Autres commandes diverses

Séquence de touche	Description
CTRL-l	Rafraichissement écran.
ESC-!	Execution d'une commande shell. Le résultat apparaitra dans une fenêtre `"emacs"`.

C.3.7 Macros

Une séquence de traitements `"emacs"` peut être mémorisée afin d'être rejouée une ou plusieurs fois ultérieurement. Cette séquence peut être nommée, sauvegardée et restaurée au cours d'une autre session `"emacs"`.

CTRL-X (
Définition de la macro
liste des traitements emacs
CTRL-X)

Séquence de touche	Description
CTRL–x e	Execution de la dernière macro créée.
CTRL–u *n* CTRL–x e RETURN	Execution *n* fois de la dernière macro créée.
ESC–x *name-last-kbd-macro*	Association d'un nom à la macro dernièrement définie.

Pour pouvoir réutiliser cette macro au cours d'une autre session, on peut :

- Soit sauver sa définition dans un fichier externe qu'il faudra recharger avant utilisation (commande "ESC–x `load-file`").
- Soit sauver sa définition dans le fichier d'initialisation "`.emacs`". Il suffit de l'ouvrir, de s'y positionner, et d'entrer la commande :

  ESC–x `insert-kbd-macro` RETURN *nom_de_macro* RETURN

  Lors d'une prochaine session "`emacs`", la macro sera executée par la commande ESC–x *nom_de_macro*.

Remarque C.11 :

Pour annuler une modification de texte, il existe la commande "`undo`*" (représentée par CTRL–x u). Cette modification de texte est liée à un buffer. Elle peut être répétée plusieurs fois (manuellement ou par la sequence CTRL–u n CTRL–x u). Les modifications de texte sont mémorisables dans la limite de 8000 caractères par buffer.*

Annexe D

Licence associée à ce document

Version 2, June 1991

51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU General Public License

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".
   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.
   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if

the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.
Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.
In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.
12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of Terms and Conditions

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) <year> <name of author>
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989
> Ty Coon, President of Vice

Bibliographie

[1] Jeffrey E. F. Friedl *Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools* – 1st Edition January 1997 – O'Reilly & Associates Inc.

[2] Dale Dougherty & Arnold Robbinsx, *Sed & Awk Programming* – 2nd Edition March 1997 – O'Reilly & Associates Inc.

[3] Debra Cameron, Bill Rosenblatt & Eric Raymond, *Learning GNU Emacs* – 2nd Edition September 1996 – O'Reilly & Associates Inc.

[4] Bill Rosenblatt, *Learning the Korn Shell* – June 1993 – O'Reilly & Associates Inc.

[5] Randal L. Schwartz and Tom Christiansen, *Learning Perl* – Second edition, July 1997 – O'Reilly & Associates Inc.

[6] Larry Wall, Tom Christiansen and Randal L. Schwartz, *Programming Perl* – Second edition, September 1996 – O'Reilly & Associates Inc.

[7] Sriram Srinivasan, *Advanced Perl Programming* – March 1992 – O'Reilly & Associates Inc.

[8] Paul Albitz and Cricket Liu, *DNS and Bind* – Mars 1993 – O'Reilly & Associates Inc.

[9] Sylvain Baudry. *Les réseaux 802.3 et IP, Gestion avec SNMP* – Juin 1993 – RNUR.

[10] Sylvain Baudry. *Administration centralisée de plusieurs* Unixs – Septembre 1994 – INED.

[11] Sylvain Baudry. *Description des protocoles NFS et NIS – Applications et administration sur différents* Unix – Avril 1995 – INED.

[12] Sylvain Baudry. *Programmation réseau avec les BSD-Sockets* – Juin 1996 – ESME.

[13] Helmut Kopka and Patrick W. Daly. *A Guide to LaTeX2e, Document Preparation for Beginners and Advanced Users* – Second edition, 1995 – Addison-Wesley

D.1 Conventions et Notations

Dans toute la suite de ce document, les conventions suivantes seront adoptées :

`% commande`
: représente la commande saisie par l'utilisateur au niveau de l'invite (*prompt*) UNIX.
Cette commande obéira aux règles explicitées au niveau de la section 1.3.

« ␣ » ou SPACE
: représente le caractère « *espace* ».

TAB
: représente le caractère « *Tabulation* ».

ALT
: représente la touche « *ALT* ». En général, la touche « *ALT* » se trouve de part et d'autre de la barre d'espace du clavier alpha-numérique sur les claviers étendus 102 touches.

SHIFT
: représente la touche « *SHIFT* » ou « ↑ », ou encore "*Maj*". Cette touche permet de passer en mode "*majuscule*".
En général, la touche « *SHIFT* » se trouve de part et d'autre du clavier alpha-numérique sur les claviers étendus 102 touches.

CTRL
: représente la touche « *CONTROL* » ou « *CTRL* », ou encore "*ctrl*".
En général, la touche « *CTRL* » se trouve de part et d'autre en bas du clavier alpha-numérique sur les claviers étendus 102 touches.

RETURN
: représente la touche « *RETURN* » ou « *ENTRÉE* » ou encore « ↩ ».
En général, la touche « *RETURN* » se trouve à droite du clavier alpha-numérique.

ESC
: représente la touche « *ESCAPE* » ou « *esc* ».
En général, la touche *ESCAPE* se trouve en haut à gauche sur les claviers étendus 102 touches.

`x`
: représente la touche du clavier permettant d'obtenir le caractère « `x` ».

`X`
: représente la combinaison des touches SHIFT et `x`.

X–*Y*
: représente l'appuie simultané sur les touches "*X*" et « *Y* » du clavier.

Index

#, 93
$, 194
&, 59, 89
&&, 89
fp ', 56
', 57
fp ", 56
", 57
(), 107
*, 54, 55, 57, 106
+, 106
-, 54
;, 89
<, 50, 57
>, 50, 51, 56
?, 54, 57, 106
[], 54, 55, 57, 105
\, 57, 105
$, 49, 56, 105
^, 105
‘, 60

at, 62
awk, 95, 105, 131, 189
  action, 137
  ARGC, 142
  ARGV, 142
  champ, 131
  enregistrement, 131
  exit, 141
  fonctions prédéfinies, 138
    exp, 139
    index, 139
    int, 139
    length, 139
    log, 139
    print, 139
    printf, 139
    sprintf, 139
    sqrt, 139
    substr, 139
  fonctions utilisateur, 140
  opérateurs arithmétiques, 137
  structures de contrôle, 141
    break, 141
    continue, 141
    for, 141
    if, 141
    next, 141
    while, 141
  sélecteur, 132
    BEGIN, 132
    BEGIN, 132
    syntaxe, 134
  tableaux, 137
  variables, 135
    $0, 135
    $*i*, 135
    FILENAME, 135
    FS, 135
    IFS, 135
    IRS, 135
    NF, 135
    NR, 135
    OFMT, 135
    OFS, 135
    ORS, 135
    RS, 135
  variables pré définies, 135

batch, 62
bg, 61
break, 102, 141

case, 100, 202
cat, 19, 28
cd, 15
  équivalence, 15
chemin d'accès
  absolu, 10
  relatif, 11
chmod, 26

commande
    externe, 49
    format, 6
    interne, 49
    liste, 89
commentaire, 93
`continue`, 102, 141
`cp`, 20
crontab, 65
`cut`, 31

`egrep`, 111
`emacs`, 210, 241
`env`, 48
`exit`, 5, 102, 141, 206
`export`, 48, 197
`expr`, 86, 195
expression régulière, 87, 105

`fg`, 61
`fgrep`, 112
fichier
    attributs, 18
    chemin d'accès à un, 10
    lien symbolique, 19, 20
    modes d'accès, 24
    nom de, 9
    propriétaire, 23
    spécial, 11, 19
filtre, 13, 28
fonction, 91
`for`, 101, 102, 141, 204
`ftp`, 34
    commandes internes à, 35
    comparaison avec `rcp`, 41
    options, 34

GID, 23
`grep`, 29, 105, 109
    `egrep`, 111
    `fgrep`, 112
    utilisation, 110

`if`, 99, 141, 201

`jobs`, 59

`kill`, 60, 103

lien
    logique, 21
    recherche des liaisons entre, 22
    symbolique, 20
    visualisation du nombre de, 22
`ln`, 20
login, 5
`ls`, 16
    équivalence, 16
    options, 16

`man`, 8
manuel
    de référence, 7
    page du, 7
    paragraphes du, 8
`mkdir`, 17
    équivalence, 17
`more`, 19, 28
`mv`, 20
métacaractère, 54

`next`, 141

PID (*Process IDentifier*), 59, 61
pipe, 53, 89
`printenv`, 48
`printf`, 189
`pwd`, 15
    équivalence, 15

`rcp`, 38
    comparaison avec `ftp`, 41
`read`, 85, 196
`remsh`, 40
`return`, 92
`rlogin`, 37
    comparaison avec `telnet`, 41
`rm`, 22
`rmdir`, 17
    équivalence, 17
`rsh`, 40
répertoire
    `..`|hyperpage, 17
    `.`|hyperpage, 17
    système, 11

`sed`, 95, 105, 115
    commandes, 117
    fonctionnement, 116
    requête, 116
    symboles, 119
`set`, 48, 81, 193

setenv, 197
shell, 45
    arguments, 79
    bash, 46, 74
    csh, 45, 73
    environnement, 46
    ksh, 46, 74
    mécanisme d'évaluation, 45, 87
    script, 73
        appel, exécution, 94
    sh, 45, 73
    tcsh, 46
shift, 79, 85
signal, 103
    trap (définition), 103
    trap (commande), 104
sort, 29
stderr (sortie d'erreurs standard), 12, 51
stdin (entrée standard), 12, 50, 53
stdout (sortie standard), 12, 50, 51, 53

telnet, 6, 33
    commandes internes à, 33
    comparaison avec rlogin, 41
test, 97
tests, 97
    chaînes de caractères, 98
    numériques, 99
    opérateurs logiques, 99
    sur les fichiers, 98
trap, 104, 206

UID, 5, 23
unset, 48
until, 101, 102

variable
    #, 196
    $, 196
    *, 81, 196
    -, 81
    ?, 81, 92, 196, 198
    #, 79, 81
    $, 81
    DISPLAY, 197
    EXINIT, 198
    expression d'initialisation, 82
    gestion, 48
    HISTSIZE, 48
    HOME, 48, 197
    LANG, 48, 197
    MAIL, 197
    PATH, 48, 49, 73, 197
    positionnelle, 79
    PS1, 48, 198
    PS2, 48
    TERM, 197
    TZ, 197
    USER, 197
    usuelle, 48
    visualisation, 49
vi, 115, 210, 212

wait, 61, 81
wc, 30
while, 101, 102, 141, 204
wildcard, 54

Table des figures

1.1 Organisation de l'arborescence UNIX avec les systèmes de fichiers 4
1.2 Structure arborescente du système UNIX 10
1.3 Entrées/Sorties d'un processus 12
1.4 Principe des filtres sous UNIX 13

2.1 Liens des répertoires "." et ".." 18
2.2 Différence entre les liens symboliques et les liens logiques 21
2.3 Algorithme de vérification des droits d'accès sous UNIX 25
2.4 Rappel sur les filtres 28
2.5 Fonctionnement de la commande "cat" 28

3.1 Terminologie pour la description de "ftp". 35
3.2 Etablissement d'une connexion ftp 35
3.3 Envoi/Réception de fichier(s) de "server-host" vers "local-host" avec "ftp". 36
3.4 Exemple d'utilisation de la commande "rcp". 40

1.1 Exécution d'une commande sous UNIX 47
1.2 Enchainement de commandes, mécanisme du *pipe*. 53

5.1 Enchaînement de commandes et modèle d'exécution 90

C.1 Résumé des commandes d'insertion de texte de "vi". 217
C.2 Commandes de déplacement du curseur avec "vi". 220
C.3 Commandes de suppression de texte avec "vi". 222

Liste des tableaux

1.1 Équivalences Unix et OpenVms pour la déconnexion et le changement de mot de passe. 6
1.2 Équivalences OpenVms, MS-DOS et Unix pour accéder à l'aide . 9
1.3 Équivalences des noms des canaux d'entrées/sorties entre Unix et OpenVms 13

2.1 Équivalence des commandes de déplacement dans l'arborescence sur le système 15
2.2 Équivalences entre systèmes pour afficher la liste des fichiers d'un répertoire 16
2.3 Équivalences entre systèmes pour la gestion des répertoires . . . 17
2.4 Exemples d'equivalences entre systèmes pour la manipulation des répertoires "." et ".." 18
2.5 Équivalence entre `cat`et `more` et d'autres systèmes 20
2.6 Équivalence des commandes `cp`, `mv` et `ln` entre Unix,OpenVms et MS-DOS 22
2.7 Équivalences de la commande `rm` entre Unix,OpenVms et MS-DOS. 23
2.8 Équivalences pour les notions d'identités entre Unix et OpenVms 24
2.9 Actions possibles en fonction du masque de protection 24
2.10 Valeurs associées aux différents droits d'accès 26
2.11 Exemple d'affectation d'un masque en octal 26
2.12 Abréviations utilisées par la commande `"chmod"` 27
2.13 Exemples de modifications des protections par rapport à celles déjà actives 27

3.2 Comparaisons `telnet`/`rlogin` 41
3.3 Comparaisons `ftp`/`rlogin` 42

1.1 Liste des variables les plus usuelles. 48

1.2 Équivalence pour l'utilisation des métacaractères entre UNIX, OpenVMS et MS-DOS. 55

11.4 Fonctions acceptant un argument de type numérique. 139

11.5 Fonctions acceptant des arguments de type alphanumérique. . . . 139

12.1 Description des fichiers en entrée ou en sortie pour la converions des utilisateurs OpenVMS vers UNIX. 160

C.21 Commandes de base de `"emacs"` 241

C.22 Principales commandes accédant aux rubriques d'aide d'`"emacs"`. 245