Université Pierre et Marie Curie
Paris VI

Master de Sciences et Technologie :

— **Mention Physique et Applications**

— **Mention Sciences de l'Univers, Environnement, Écologie**

Méthodes numériques et informatiques
UE 4P009
Unix et programmation

L'environnement Unix

Sylvain.Baumont@lpnhe.in2p3.fr
Christophe.Boitel@lmd.polytechnique.fr
Marine.Bonazzola@lmd.jussieu.fr
julien.brajard@locean.upmc.fr
Jacques.Lefrere@upmc.fr

avec des contributions de :

Frédéric Bernardo
Albert Hertzog
Frédéric Meynadier

2017–2018

Notations

— La police « machine à écrire », à espacement fixe, est utilisée pour indiquer les éléments du langage UNIX.

— Les crochets `[...]` délimitent généralement [1] les éléments *optionnels* de la syntaxe ou les arguments *optionnels* des commandes ; ces symboles ne font pas partie de la syntaxe.

— Lors de la description de la syntaxe, l'usage de la police italique à espacement fixe indique où l'utilisateur doit substituer les identificateurs, expressions, ou valeurs qu'il a choisis.

— Les combinaisons de touches impliquant la touche de contrôle sont parfois notées plus brièvement avec l'accent circonflexe : par exemple la combinaison `Ctrl C` peut être notée simplement sous la forme `^C`.

Exemple

La syntaxe décrite comme suit :

syntaxe

```
grep [-option] motif [fichier]
```

peut être utilisée comme dans les exemples suivants :

```
grep -i septembre /etc/motd
```

avec l'option `-i`, le motif `septembre` et le fichier `/etc/motd`,

```
grep toto
```

sans option, avec le motif `toto`, sans fichier donc en lisant les données saisies au clavier.

Indications de lecture

♠ Sous-section facultative

Les sections (ou sous-sections) qui peuvent être omises en première lecture sont indiquées par le symbole ♠ placé en marge comme ci-dessus.

Conseils pratiques

♥ ⇒ Les règles de « bon usage » du langage, qui, au delà de la norme, sont motivées par des objectifs de lisibilité, de portabilité ou de robustesse du code source, sont repérées par le symbole ♥ dans la marge extérieure du texte, comme pour ce paragraphe.

Difficultés

⚠ ⇒ Les points présentant des difficultés particulières ou des risques d'erreur sont indiqués par le symbole ⚠ dans la marge extérieure du texte, comme pour ce paragraphe.

Ce document a été mis en page grâce au traitement de texte LaTeX.

1. Quand les crochets font partie de la syntaxe, leur signification est décrite explicitement, par exemple pour définir des ensembles de caractères dans les générateurs de noms de fichiers (11.2.1), ou dans les expressions rationnelles (6.3.2) ou dans les structures de contrôle de type `case` (14.1.2).

Table des matières

Chapitre 1

Introduction

1.1 Bref historique d'unix et linux

Le système d'exploitation (operating system) UNIX date des années 1970. L'essentiel du système est écrit en langage C [1], ce qui lui a assuré une large diffusion dans laquelle les milieux universitaires ont joué un rôle essentiel. Plusieurs branches parallèles d'UNIX se sont très rapidement développées (notamment BSD et System V), conduisant à quelques différences dans les commandes et leur syntaxe.
Des efforts ont ensuite été entrepris en vue d'une synthèse des meilleurs aspects des différents UNIX avec la définition de la norme POSIX (Portable Operating System Interface). L'implémentation de cette norme est encore incomplète et quelques différences subsistent pour certaines commandes [2] selon les versions de système UNIX. Mais ces différences concernent beaucoup moins l'utilisateur que l'administrateur du système.
Noter enfin que les systèmes UNIX s'appuient largement sur les programmes du projet GNU dont l'ambition est de développer un système complet (Gnu is Not Unix). Ces programmes d'une grande qualité sont distribués sous la licence dite GPL (General Public License) de la Free Software Fondation.

1.2 Principales caractéristiques du système unix

UNIX est un système interactif, multi-tâches et multi-utilisateurs capable de partager des ressources (fichiers, authentification, ...) selon une hiérarchie de droits d'accès. Il comporte une documentation en ligne (`man`) des commandes. Ce système s'intègre facilement dans le réseau pour échanger des informations et développer des applications réparties entre plusieurs calculateurs. UNIX comporte un mécanisme très puissant de chaînage des processus par les tubes (pipes) qui permet d'accomplir des tâches complexes par simple assemblage d'outils élémentaires. Enfin, l'interpréteur (shell) intègre un langage de programmation permettant d'automatiser les tâches répétitives.
Le système UNIX est un système ouvert, implémenté sur diverses architectures, du portable au super-calculateur :

— propriétaires (aix d'IBM, hp-ux de HP, solaris de SUN, OS-X d'APPLE, ...) ;

— libres (linux, net-bsd, free-bsd, ...).

Le système UNIX est aujourd'hui installé sur la plupart des gros calculateurs scientifiques. Mais c'est le système linux, une des versions d'UNIX du domaine public, qui a permis de rendre UNIX de plus en plus populaire dans le domaine des ordinateurs personnels et des petits serveurs. Linux est aujourd'hui dominant dans le calcul intensif avec plus de 95 % des calculateurs du TOP 500 (*cf.* http://www.top500.org/statistics).
Linux est disponible dans de nombreuses distributions issues de quelques grandes branches qui diffèrent notamment par les outils de maintenance du système et des « packages » logiciels et des orientations serveur ou grand public : Slackware, Redhat et ses dérivées (mandrake puis mandriva et aujourd'hui mageia, centOS, Scientific Linux, Fedora), qui utilisent le système de gestion de paquets RPM (Redhat Package Manager), debian dont dérivent knopix et ubuntu, ... (pour avoir une idée du foisonnement, consulter par

1. Moins de 10 % du noyau UNIX est écrit en assembleur.
2. Par exemple : affichage des processus avec `ps`, impression avec `lpr`/`lp`, ...

exemple l'arbre généalogique d'UNIX http://www.levenez.com/unix/ ou celui de linux http://fr.wikipedia.org/wiki/Distribution_Linux).

1.3 Présentation du document

Le système UNIX est un système d'exploitation que l'utilisateur découvre tout d'abord au travers de l'*interpréteur de commandes* que constitue le *shell*. L'utilisation d'UNIX en interface ligne de commande nécessite un certain effort d'apprentissage. Mais cet investissement permet d'accéder à des outils très efficaces de manipulation de données (transformation de fichiers textes, gestion de la hiérarchie de fichiers, ...) et de contrôle des procédures (UNIX ou issues d'autres langages), d'autant plus puissants que le shell est aussi un *langage de programmation*.
L'organisation du document respecte cette progression en présentant d'abord les outils de base avant d'aborder la programmation en shell.

1.3.1 Unix interpréteur de commandes

La première partie de ce document décrit les principales commandes qui permettent d'éditer des fichiers texte, de modifier leur contenu (par l'intermédiaire de filtres notamment) et de les manipuler. Après une brève présentation des premiers contacts avec UNIX (chapitre 2), le système de fichiers est décrit (chapitre 3), puis les commandes d'édition (chapitre 4). La notion de filtre est ensuite abordée avec des commandes élémentaires (chapitre 5), puis la notion de motif générique est introduite avec les expressions régulières (chapitre 6), afin de décrire trois filtres paramétrables très puissants :

- `grep` qui permet de rechercher des motifs (chapitre 7),
- `sed` qui est un éditeur de flux (chapitre 8),
- et enfin `awk` qui permet de programmer dans une syntaxe proche de celle du langage C et possède des fonctions de tableur (chapitre 9).

Cette partie s'achève sur les processus et le mécanisme des redirections et tubes, permettant l'assemblage des outils de base, au cœur du système UNIX (chapitre 10).

1.3.2 Unix langage de programmation

La deuxième partie de ce document aborde le shell en tant que langage de programmation. Sont présentés ici les variables du shell (chapitre 11), qui permettent de paramétrer les procédures shell (chapitre 12), ainsi que les tests (chapitre 13), utilisés dans les structures de contrôle (chapitre 14). Les notions avancées du shell sont évoquées rapidement au chapitre 15. Quelques conseils de bonne programmation sont fournis en guise de conclusion. Il est conseillé de les lire avant d'entreprendre l'écriture de shell-scripts.

1.3.3 Résumés aide-mémoire

Enfin, des fiches aide-mémoire ont été regroupées à la fin du document, pour faciliter la consultation rapide. Ces résumés concernent notamment :

- Les structures de contrôle du shell comparées à celles d'autres langages tels que C, fortran et scilab (p. 119) ;
- Les éditeurs de texte `vi` (p. 120) et `emacs` (p. 122), après l'index et la bibliographie.

1.4 Les différents shells : avertissement

Il existe de nombreuses versions de l'interpréteur de commandes UNIX ou shell, qui présentent des différences à la fois en termes de syntaxe et de possibilités offertes (fonctions internes, interactivité, fichiers de configuration, ...). Suivant la syntaxe des structures de contrôle, on peut les regrouper essentiellement en deux familles :

— les shells de type BOURNE, dérivés du shell historique `sh` : `ksh`[3], `pdksh`[4], `mksh`[5], `bash`, `zsh`, ...

— les shells du type C-shell, dont les structures de contrôle sont plus proches de celles du langage C : `csh`, `tcsh`, ...

Ce document n'aborde que les shells de type BOURNE, dans leurs versions modernes : `ksh` (KORN-shell) et `bash` (BOURNE again shell), shell par défaut sur les postes de travail sous `linux`.
Pour plus de détails sur les différents shells, on se reportera à NEWHAM et ROSENBLATT (2006) ou RAMEY et FOX (2006) sur `bash`, ROSENBLATT et ROBBINS (2002) sur `ksh`, KIDDLE *et al.* (2004) sur `zsh` et DUBOIS (1995) sur `csh` et `tcsh`.
Par ailleurs, l'essentiel des commandes traitées sont communes aux différents systèmes UNIX, dont `linux` fait partie. Mais compte tenu de la popularité des systèmes `linux`, certaines commandes spécifiques de `linux` sont abordées et la majorité des exemples testés sous `linux`.

1.5 Linux à partir de windows

Partant d'un poste de travail personnel sous le système d'exploitation `windows`, de nombreuses possibilités sont actuellement disponibles pour accéder au monde `unix`. Nous n'en évoquerons que quelques-unes, en commençant par les plus faciles à mettre en œuvre.

MobaXterm (*cf.* `http://mobaxterm.mobatek.net/`) permet simuler localement une partie des commandes du système unix. Il fournit aussi des outils pour se connecter à des serveurs `unix`, pour y ouvrir des sessions texte, voire graphiques, et échanger des fichiers. C'est l'outil que nous conseillons et dont l'installation est décrite en annexe (A, p. 111).

Cygwin Le logiciel `cygwin` est une émulation assez complète du système linux dont l'installation, un peu plus lourde, est décrite en annexe (B, p. 115).

wubi pour Windows-based UBuntu Installer permet d'installer ubuntu sur une machine windows sans modifier les partitions (*cf.* `http://doc.ubuntu-fr.org/wubi`).

Machines unix sur support dédié Si on dispose d'un support physique bootable : disque, clef-usb, que l'on peut entièrement consacrer au système unix, cela peut constituer une solution prudente, sans interférence avec le système existant. C'est par exemple ce que propose le logiciel open-source `linuxlive USB Creator` pour un grand choix de distributions linux (*cf.* `http://www.linuxliveusb.com`).

Machines virtuelles On peut aussi installer un système unix assez complet comme application particulière de windows : on parle alors de machine virtuelle simulée par la machine physique, en s'appuyant sur ses ressources. Les machines virtuelles sont des systèmes d'exploitations invités (guest en anglais) alors que le système de la machine physique est le système hôte (host en anglais) qui les héberge. Avant d'installer une machine virtuelle, il faut tout d'abord installer le logiciel d'hébergement des machines virtuelles dit logiciel de virtualisation, par exemple `virtualbox` (`https://www.virtualbox.org`), `VMware` (`http://www.vmware.com`). Il sera alors possible d'installer plusieurs systèmes d'exploitation invités sur la machine hôte.

Signalons enfin la technique du dual-boot : on installe en parallèle, éventuellement sur le même disque, le système linux à côté du système windows ; c'est au démarrage que l'on choisit le système d'exploitation à activer. L'installation d'unix selon cette méthode requiert des compétences d'administration et comporte des risques de pertes d'information, notamment lors du partitionnement du disque ; c'est pourquoi nous la déconseillons.

3. `ksh`, développé par David Korn et initialement logiciel propriétaire, est maintenant passé dans le domaine public.
4. `pdksh` est une version de `ksh` du domaine public.
5. `mksh`, MirBSD Korn Shell est un des successeurs de `pdksh`.

Chapitre 2

Session Unix, environnement réseau

⚠⇒ **N.-B. :** le système UNIX distingue minuscules et majuscules : on dit qu'il est sensible à la casse (case sensitive). La plupart des commandes UNIX s'écrivent en minuscules.

2.1 Session utilisateur

2.1.1 Compte utilisateur

Pour pouvoir ouvrir une session sur une machine, un utilisateur doit y posséder un compte ; cela signifie que l'administrateur lui a attribué :

- un identifiant (ou login) associé à un numéro unique (`uid`) (*cf.* 2.2.4 p. 8) ;
- un mot de passe (ou password) qui doit rester confidentiel ;
- un groupe (associé à son numéro unique `gid`) parmi ceux définis sur la machine [1] (*cf.* 2.2.4 p. 8) ;
- un répertoire d'accueil personnel (ou home directory) destiné à héberger tous les sous-répertoires et fichiers qui lui appartiennent (*cf.* 3.1, p. 15) ;
- un « interpréteur de commandes » (ou shell) : `sh`, `ksh`, `bash`, `zsh`, `csh` ou `tcsh`.

L'ensemble de ces informations est stocké dans un fichier système (souvent `/etc/passwd`), mais le mot de passe est mémorisé [2] sous une forme cryptée ; c'est pourquoi l'administrateur du système ne peut retrouver un mot de passe oublié par un utilisateur (mais l'administrateur peut le modifier).
Par exemple, la ligne définissant le compte `p6m1mni` dans le fichier `/etc/passwd` prend la forme (le mot de passe crypté n'est pas stocké ici et est remplacé par le caractère `!`) :

```
p6m1mni:!:40369:2055:Jacques LEFRERE:/home/p6pfal/p6m1mni:/usr/bin/ksh
```
login : : uid : gid : identité de l'utilisateur : répertoire d'accueil : shell

Les ressources informatiques étant limitées, un quota d'espace disque est bien souvent spécifié : s'il est atteint, l'utilisateur ne peut plus écrire [3] dans son espace personnel.

2.1.2 Session texte et session graphique

L'ouverture d'une session sur une machine UNIX peut s'effectuer selon deux modes différents :

en mode texte sur une console ou par une connexion à distance à partir d'une machine locale via `slogin` ou `ssh` [4], par exemple
`slogin` *`user`*`@sappli1.datacenter.dsi.upmc.fr`

1. Les groupes et leurs `gid` sont stockés dans le fichier système `/etc/group`
2. Pour des raisons de sécurité, les mots de passe cryptés sont souvent stockés en dehors du fichier `/etc/passwd`, par exemple dans le fichier `/etc/shadow/` aux droits plus fermés.
3. Cette interdiction peut empêcher l'ouverture d'une session en mode graphique, qui nécessite généralement l'écriture d'informations dans l'espace personnel. Il faut alors ouvrir une session en mode texte (*cf.* 2.1.2, p. 5) pour libérer de l'espace disque.
4. `ssh` assure des communications sécurisées par cryptage. L'usage de `telnet` où les informations circulent en clair sur le réseau et sont donc susceptibles d'être interceptées est aujourd'hui abandonné sur la majorité des serveurs.

en mode graphique dans une fenêtre d'accueil dont la présentation dépend du gestionnaire de fenêtres (window manager) : `fvwm`, `kde`, `gnome`, `icewm`, `lxde`, `xfce`[5]... Ce gestionnaire, qui constitue souvent un véritable environnement de bureau (Desktop Environment) définit l'interface graphique (icônes, menus déroulants, boutons divers, ...) qui permet de lancer et de contrôler les applications et finalement de fermer la session.

Mais les deux types de session s'initient de façon similaire par l'identification puis l'authentification de l'utilisateur qui doit saisir[6] successivement :

— son identifiant à l'invite `login`

— son mot de passe à l'invite `password`[7]

Une fois le nom d'utilisateur et le mot de passe reconnus, le comportement diffère suivant le type de session.

En mode texte : Le système affiche le mot du jour[8] et la date de la dernière connexion. La configuration personnelle est activée par l'exécution des fichiers de type `.profile` dans les shells de type `sh` ou `.login` dans les shells de type `csh` (*cf.* 15.1.6, p. 103). Ces fichiers peuvent positionner certaines options du shell, initialiser des variables d'environnement (par exemple : chaîne de caractères d'invite, type de terminal utilisé ...) et lancer des commandes qui seront exécutées à chaque début de session. S'inscrit ensuite à l'écran une chaîne de caractères (invite ou prompt en anglais) invitant l'utilisateur à entrer des commandes pour interagir avec le `shell`. Une fois le travail terminé, la commande `exit` permet de fermer la session texte.

En mode graphique : Le contrôle est passé au gestionnaire de fenêtres qui permet de lancer des applications via des menus et des icônes sans identification ni authentification complémentaires : parmi ces applications, des consoles peuvent être activées qui permettent de passer des commandes en mode texte. Mais, une fois toutes les applications terminées, il faut signifier au gestionnaire de fenêtres la fin de la session graphique pour éviter que des applications puissent être lancées sans authentification par quelqu'un d'autre sur le poste graphique. Une fois la session terminée, on doit retrouver la fenêtre graphique d'accueil initiale.

♠ Commutation entre mode graphique et mode console sous linux

Sous linux, par exemple en cas de problème en mode graphique (quota disque atteint ou difficulté avec la vidéo), on peut passer du mode graphique au mode texte en frappant simultanément Ctrl | Alt | F1 par exemple pour travailler sur la console 1. Six pseudo-consoles sont disponibles de `F1` à `F6` qui permettent d'ouvrir des sessions en mode texte seul.
Le retour en mode graphique s'effectue par la frappe simultanée de Ctrl | Alt | F7[9]
Ne pas oublier de fermer toutes les sessions en mode texte ouvertes sur les pseudo-consoles non affichées en fin de session graphique.

5. Avec son gestionnaire graphique de fichiers, `Thunar`.

6. Ne pas utiliser le pavé numérique lors de la saisie et vérifier que la touche majuscule (CapsLock) ne soit pas verrouillée.

7. Dans certains cas, aucun écho n'est affiché lors de la saisie du mot de passe, de façon à ne pas dévoiler aux témoins le nombre de caractères saisis.

8. Appuyer sur la barre d'espacement pour parcourir le mot du jour page par page ; taper `q` pour terminer son affichage (voir les filtres `more` et `less` 2.3.2, p. 9).

9. Sur certaines distributions linux, dont Mandriva 2010, il faut utiliser la touche F8 au lieu de la touche F7. À l'inverse, sur la distribution mageia, ou Scienfic-Linux, le mode graphique est accessible via Ctrl | Alt | F1 alors que Ctrl | Alt | F2 jusqu'à F5 permettent d'accéder à des pseudo-consoles.

2.2 Les commandes unix

2.2.1 Syntaxe élémentaire des commandes unix

Les lignes de commandes UNIX sont interprétées par un shell selon une syntaxe bien précise, qui, dans sa forme élémentaire, peut être décrite comme suit :

Une ligne de commande unix est découpée par le shell en une suite de mots séparés par des blancs [10]. Le premier mot est le nom de la commande ; les mots suivants constituent les paramètres de la commande. Chaque mot représente un paramètre, dont le rôle est déterminé par sa position dans la phrase.

Certains paramètres facultatifs, commençant généralement par le caractère « - », et précédant les opérandes (qui sont très souvent des noms de fichiers), sur lesquels agit la commande, permettent de passer des options à la commande pour contrôler finement son comportement.

commande␣[-*options*]␣[*liste_d_opérandes*]

Exemple : `ls␣-l␣/tmp`

1. `ls` est la commande (affiche la liste des fichiers)
2. `-l` est l'option (liste longue, avec les attributs des fichiers)
3. `/tmp` est le nom de répertoire passé en argument (répertoire dont on liste les fichiers)

Les commandes de type GNU respectent aussi deux conventions complémentaires :

1. La fin des paramètres optionnels peut être indiquée par « `--` », permettant ainsi d'interpréter correctement des noms de fichiers commençant par « - » sans les confondre avec des spécifications d'options.
2. Si le nom du fichier d'entrée est remplacé par « - », les données d'entrée sont prises sur l'entrée standard (*cf.* chap. 10, p. 72).

⚠⇒ **Remarque :** Le shell interprétant les espaces comme des séparateurs, on évitera leur emploi dans les noms de fichiers.

2.2.2 Aides à l'interactivité

Les shells modernes disposent de fonctionnalités facilitant la saisie des commandes : l'historique et l'édition en ligne des commandes d'une part, la complétion automatique des noms de commandes et de fichiers d'autre part.

Historique des commandes et édition en ligne

Le shell mémorise la suite des commandes qui ont été effectuées. La primitive (builtin, *cf.* 15.1.1, p. 101) `history` permet d'afficher la liste numérotée des dernières commandes saisies. Il est d'autre part possible de rappeler les précédentes commandes et de naviguer dans cette liste avec les flèches verticales : on peut alors éditer une ligne pour l'adapter à de nouveaux besoins avant de la relancer.
Sous `bash`, cette édition se fait par défaut dans le mode `emacs` [11] (*cf.* 4.4, p. 36) ; les déplacements peuvent se faire avec les flèches horizontales et les raccourcis d'`emacs` utilisant la touche Ctrl sont utilisables (*cf.* 4.4.3, p. 38) :

— Ctrl E aussi noté `^E` (end) positionne en fin de ligne ;

10. Les caractères qui séparent les mots pour le shell sont définis par la variable `IFS` Internal Field Separator qui comporte par défaut l'espace, la tabulation et la fin de ligne. D'autres caractères spéciaux soumis à interprétation par le shell peuvent intervenir dans ce découpage (*cf.* la section 15.1.5, p. 102 pour une analyse plus précise de la ligne de commande), notamment les opérateurs de redirection (*cf.* chapitre 10).

11. Le mode `vi` (*cf.* 4.3, p. 33) est aussi disponible pour éditer la ligne de commande, mais il est moins intuitif. Le choix du mode se fait par la commande : `set -o vi` ou `set -o emacs` (*cf.* 15.1.7, p. 103).

— Ctrl A aussi noté `^A` (begin donnerait `^B`, déjà utilisé pour back) positionne en début de ligne ;

— Ctrl W aussi noté `^W` (word) efface le mot précédent le curseur ;

— Ctrl T aussi noté `^T` (transpose) échange le caractère sous le curseur avec le précédent.

Complétion automatique des noms de commandes et de fichiers

Le shell comporte un puissant mécanisme de complétion automatique qui permet d'alléger et surtout de valider la saisie : ⇐ ♥

— des commandes ;

— des noms de fichiers et de leurs chemins d'accès.

L'utilisateur peut se contenter d'écrire les premières lettres de la commande ou du nom de fichier et demander au shell de compléter en frappant la touche Tab de tabulation. S'il n'y a aucune ambiguïté à partir de ces lettres, le shell complète lui-même, et sinon, il propose (si les possibilités sont en nombre raisonnable) la liste des possibilités ; il est alors possible, quitte à réitérer le processus, de compléter jusqu'à lever toute ambiguïté. Cette possibilité permet de choisir des noms de fichiers longs, à la signification explicite, car ils ne devront être saisis qu'une fois, lors de leur création. Elle permet aussi d'éviter les fautes de frappe en demandant au shell de vérifier la validité des chemins.

2.2.3 La documentation en ligne avec `man`

Présentation de la commande `man`

La documentation en ligne de la commande *`cmd`* est affichée par `man` *`cmd`* (`man` étant l'abréviation de manual). Elle permet en particulier de connaître la syntaxe d'une commande et la liste des options attachées à cette commande. Chaque « page » de manuel comporte habituellement les rubriques suivantes :

— `NOM`/Name qui indique brièvement la fonction de la commande (c'est cette définition qui est affichée si on lance `whatis` suivie du nom de la commande) ;

— `SYNOPSIS`/Synopsis qui résume la syntaxe de la commande ;

— `DESCRIPTION`/Description qui décrit la commande de façon plus détaillée, précisant en particulier le type de paramètres et d'options (flags) admis ;

— `EXEMPLES`/Examples qui présente quelques exemples d'utilisation de la commande ;

— `OPTIONS`/Flags qui détaille chacune des options de la commande et précise leur syntaxe ;

— `FICHIERS`/Files qui indique les éventuels fichiers en rapport avec la commande ;

— `VOIR AUSSI`/See also qui liste les autres commandes éventuellement en rapport avec cette fonction.

♠ **Les sections des manuels**

Les manuels en ligne sont regroupés en « sections » numérotées, parmi lesquelles :

1 pour les commandes usuelles ;

3 pour les fonctions des bibliothèques hors appels système (notamment les fonctions des bibliothèques du langage C) ;

8 pour l'administration et le système.

Le numéro de la section est indiqué entre parenthèses en haut de la page. Certaines pages existent dans plusieurs sections et il est possible de spécifier la section à consulter, par exemple :

— `man 3 printf` permet d'accéder à la fonction `printf` du langage C

— alors que `man 1 printf` ou simplement `man printf` affiche le manuel de la commande UNIX `printf`.

Enfin, l'option `-a` (**a**ll) de `man` permet d'afficher successivement les manuels trouvés dans toutes les sections. De même, `whatis printf` affiche une brève définition de toutes les versions disponibles de `printf`.

Recherche par mot-clef

De manière générale, le nom d'une commande UNIX est une abréviation (en anglais) de son action. Mais comment déterminer le nom des commandes qui peuvent réaliser une action donnée ? Une première indication peut être fournie en faisant appel à la base d'indexation (cette base, constituée des définitions des commandes affichées par `whatis` est stockée dans des fichiers comme `/usr/share/man/whatis` par exemple [12]) des mots-clefs des commandes :

`man -k` *`mot-clef`* ou encore `apropos` *`mot-clef`*

permet d'afficher la liste des commandes dont la description comporte un mot-clef (keyword) déterminé.

Sous linux, une partie importante de la documentation est fournie sous un format texte permettant la navigation grâce à l'utilitaire `info`.

Par ailleurs, les commandes qui respectent les conventions de syntaxe GNU affichent un mode d'emploi résumé si on les invoque avec l'option `--help`.

Enfin, on ne peut aujourd'hui négliger l'importante source d'information que peut fournir l'usage averti d'un moteur de recherche sur le web.

2.2.4 Exemples de commandes

— se repérer parmi les utilisateurs :

`whoami`	affiche l'identifiant (souvent le nom) de l'utilisateur
`id`	affiche les noms et numéros d'utilisateur et de groupe
`who`	affiche la liste des utilisateurs connectés
`hostname`	affiche le nom du calculateur
`uname`	affiche le nom du système d'exploitation

— changer son mot de passe avec la commande `passwd` (*cf.* 3.1.4, p. 19) : une fois la commande ci-dessus lancée, le système demande de saisir le mot de passe actuel, puis le nouveau mot de passe qui devra être confirmé (ceux-ci ne seront pas affichés).

— affichage de la date : par exemple en français [13] avec le format par défaut

```
date                          ven. sept. 14 15:36:45 CEST 2012
```

ou en imposant le format, par exemple

```
date "+%A %d %B %Y"                    vendredi 14 septembre 2012
```

De nombreux formats sont disponibles y compris pour afficher des dates différentes de la date courante comme `date -d tomorrow` (*cf.* 10.5.5, p. 78).

— affichage de texte grâce à la commande `echo`

```
echo Bonjour
echo Comment vas-tu
```

— déconnexion (logout) par la commande `exit`

2.3 Commandes de gestion de fichiers et de répertoires

Avant d'entrer dans plus de détail, examinons les principales commandes de manipulation de fichiers, leurs options et quelques exemples élémentaires. On se reportera au manuel en

12. Il existe autant de fichiers `whatis` que de composantes dans la variable `MANPATH` qui contient la liste des chemins des répertoires contenant les manuels (*cf.* 15.1.6, p. 103).
13. L'affichage dépend de la langue de travail (*cf.* 15.4, p. 106).

ligne pour une étude plus exhaustive. Ne pas oublier que les arguments des commandes sont d'abord interprétés par le shell.

2.3.1 Lister les noms et attributs des fichiers avec `ls`

`ls` **l**ist : affiche la liste des fichiers du répertoire courant
`ls -a` **a**ll : liste aussi les fichiers « cachés », dont le nom commence par « `.` »
`ls -l` **l**ong : affiche aussi les attributs (droits, propriétaire, taille, date, ...) des fichiers
affiche le fichier pointé dans le cas d'un lien symbolique
`ls -lh` **h**uman-readable : affiche la taille de façon plus lisible (kilo, Méga, ...)octets
`ls -R` **R**ecursive : liste tous les sous-répertoires
`ls -F` **F**lag : indique la nature de certains fichiers par un caractère ajouté au nom :
par ex. : répertoires (`/`), exécutables (`*`), liens symboliques (`@`)
`ls -d` **d**irectory : affiche les répertoires au lieu de leur contenu
`ls -t` **t**ime : classe la liste par ordre de date des fichiers
`ls -r` **r**everse : classe la liste par ordre inverse

Ne pas oublier que, par défaut, `ls rep` liste le contenu du répertoire `rep`, d'où la nécessité de l'option `-d` pour s'intéresser au répertoire lui-même. ⇐⚠
Sauf spécification de classement explicite, les fichiers sont affichés dans l'ordre lexicographique déterminé par les variables de contrôle local (*cf.* 15.4.1, p. 106 et 15.4.2, p. 107).
L'affichage des attributs d'un fichier nécessite le droit `x` sur le répertoire où il est situé (*cf.* 3.1.4, p. 20).
Le marquage des fichiers produit par l'option `-F` de `ls` en fonction de leur type et de leurs droits [14] peut être renforcé par une « colorisation » de l'affichage via l'option `--color=auto`, par exemple : fichiers exécutables en vert (`*`), liens symboliques en bleu clair (`@`), répertoires en bleu foncé (`/`), fichiers avec suid bit en rouge.

♠ Format d'affichage de la date avec `ls -l`

Le format par défaut d'affichage de la date des fichiers dépend de la distribution UNIX [15] :

1. avec deux champs `AAAA-MM-JJ` et `HH:MM` séparés par un blanc :
   `-rw-r--r-- 1 jal latmos 5159 2013-09-03 18:36 arbre_unix.fig`
2. ou trois champs `Mmm`, `JJ` et `HH:MM` où le mois est en lettres et dépend de la variable de contrôle local :
   `-rw-r--r-- 1 jal latmos 5159 Sep 3 18:36 arbre_unix.fig`
   dans ce cas, l'année n'apparaît pas, sauf pour les fichiers anciens (plus de 6 mois) où elle prend la place des heures et minutes :
   `-rw-r--r-- 1 jal aero 5159 Sep 24 2010 arbre_unix.fig`

Si le second format est plus lisible, le premier, parfaitement hiérarchisé, se prête mieux au traitement automatique par les filtres UNIX [16]. Il est aussi possible de spécifier le format de la date avec l'option longue [17] `--time-style=` de `ls -l` : outre la valeur `'long-iso'` (qui correspond au premier format illustré plus haut), cette option admet aussi les mêmes formats que la commande `date` (*cf.* 15.4.1, p. 106).

2.3.2 Afficher le contenu d'un fichier texte avec `cat`, `more` ou `less`

`more f1` affiche le fichier `f1` page par page
la touche Espace permet d'avancer d'un écran
la touche Entrée permet d'avancer d'une ligne

14. Les propriétés ainsi soulignées sont données par les attributs affichés en tête de ligne par `ls -l`.
15. En particulier, la distribution `mandriva` utilise le premier format par défaut, alors que celle `CentOS` du serveur `sappli1` utilise par défaut le second.
16. Prendre garde que le nombre de champs de la sortie de `ls -l` peut donc être de 8 ou 9 selon le format de la date, et en particulier que le numéro du champ du nom de fichier en dépend.
17. La documentation complète de cette option n'est disponible pas sous `man`, mais sous `info`.

la touche `q` (**q**uit) permet de terminer l'affichage
la touche `/` suivie d'un motif permet de rechercher ce motif en avançant
la touche `n` (**n**ext) permet de rechercher l'occurrence suivante

`less f1` sous linux, plus puissant [18] que `more`, permet de « remonter » dans le texte

⚠⇒ Comme `less` permet de remonter dans le fichier, au contraire de `more`, il ne se termine pas automatiquement en fin de fichier : il affiche `END` sur la ligne d'état, mais il faut lui demander explicitement par la requête `q` de s'arrêter.

`cat f1` con**cat**enate : affichage (avec défilement) du contenu du fichier de nom `f1`
`cat f1 f2` concatène les deux fichiers `f1` et `f2` et affiche le résultat
`cat -n f1` affichage du contenu du fichier de nom `f1` avec ses lignes numérotées

Remarque : noter que, bien qu'acceptant une liste de fichiers en paramètres d'entrée, `cat` est aussi un filtre, le filtre identité (*cf.* chap 10) dont on peut rediriger la sortie dans un fichier. Cette commande permet donc de concaténer plusieurs fichiers texte en seul fichier, par exemple avec `cat f1 f2 f3 > fichier_resultat`.

2.3.3 Afficher le début (`head`) ou la fin (`tail`) d'un fichier texte

`head f1` affiche le début du fichier `f1` (par défaut les 10 premières lignes)
`head -n L f1` affiche les `L` premières lignes de `f1`

`tail f1` affiche la fin du fichier `f1` (par défaut les 10 dernières lignes)
`tail -n L f1` affiche les `L` dernières lignes de `f1`
`tail -n +L f1` affiche les lignes de `f1` à partir de la `L`-ième
`tail -f f1` **f**ollow : affiche la fin de `f1` en suivant ses mises à jour [19]

2.3.4 Détruire un fichier avec `rm`

`rm f1` re**m**ove : détruit le fichier `f1`
`rm -i f1` **i**nteractive : demande une confirmation avant de détruire
`rm -r rep` **r**ecursive : détruit toute la branche sous le répertoire `rep`

2.3.5 Copier un fichier avec `cp`

`cp f1 f2` co**p**y : recopie le fichier `f1` sous le nom `f2`
`cp f1 f2 ... fn rep` recopie les fichiers `f1`, `f2`, ..., `fn` dans le répertoire `rep`

2.3.6 Comparer le contenu de deux fichiers texte avec `diff`

`diff f1 f2` compare les contenus des fichiers `f1` et `f2`
et affiche à l'écran les lignes qui diffèrent
`diff -i f1 f2` permet d'ignorer les différences portant sur la casse
`diff -b f1 f2` permet d'ignorer les différences portant sur les espaces
`diff -y f1 f2` présente les lignes différentes dans 2 colonnes en parallèle [20]

2.3.7 Renommer ou déplacer un fichier avec `mv`

`mv f1 f2` **m**ove : renomme le fichier `f1` en `f2`
`mv f1 f2 rep/` déplace les fichiers `f1` et `f2` dans le répertoire `rep`

18. Dans ses versions récentes, la commande `less` active un préprocesseur déterminé par la variable d'environnement `LESSOPEN`, positionnée par défaut à `| /usr/bin/lesspipe.sh` . Ce filtre permet de convertir des fichiers que l'on ne pourrait pas afficher directement : entre autres, il affiche la liste des fichiers d'un répertoire, décompresse les fichiers compressés, extrait les fichiers archivés par `tar`, extrait le texte des `pdf`..., mais aussi analyse les fichiers d'image de type `pnm`, même en format ascii. On peut désactiver cette interprétation avec l'option `-L` ou `--no-lessopen`.
19. Cette option permet par exemple de surveiller l'avancement de l'écriture d'un fichier de résultats.
20. La commande `vimdiff` permet d'éditer en parallèle deux fichiers sous l'éditeur `vi`, *cf.* 4.3.3, p. 36.

2.3.8 Compresser/décompresser un fichier

`gzip fic` compresse le fichier `fic` en `fic.gz`
`gunzip fic.gz` décompresse le fichier `fic.gz` en `fic`

♠ **Autres outils de compression/décompression**

D'autres commandes de compression et de décompression de fichiers permettent, grâce à des algorithmes plus sophistiqués, d'obtenir de meilleurs taux de compression [21] : `bzip2`, `unlzma` et `xz` (qui est une version améliorée de `unlzma`).

`bzip2 fic` compresse le fichier `fic` en `fic.bz2`
`bunzip2 fic.bz2` décompresse le fichier `fic.bz2` en `fic`
`unlzma -z fic` compresse le fichier `fic` en `fic.lzma`
`unlzma -d fic` décompresse le fichier `fic.lzma` en `fic`
`xz -z fic` compresse le fichier `fic` en `fic.lzma`
`xz -d fic` décompresse le fichier `fic.lzma` en `fic`

2.3.9 Compter le nombre de lignes, mots, caractères et octets d'un fichier avec `wc`

`wc f1` **w**ord **c**ount : compte le nombre de lignes, de mots et d'octets du fichier `f1`
Les options `-l`, `-w`, `-m`, `-c` permettent de n'afficher respectivement que le nombre de lignes, de mots, de caractères ou le nombre d'octets [22]. On peut combiner ces options de `wc`, mais l'ordre des options est sans influence sur l'ordre d'affichage des nombres demandés, classés de l'entité la plus « grande » à la plus petite : nombre de lignes, puis nombre de mots, puis nombre de caractères et enfin nombre d'octets. ⇐⚠

Remarque générale

Les commandes `ls`, `cat`, `head`, `tail`, `more`, `less`, `wc` et `rm` peuvent s'appliquer à une liste de fichiers (par exemple, la commande `rm f1 f2` permet d'effacer `f1` et `f2`).

2.3.10 Se repérer (`pwd`) et se déplacer (`cd`) au sein de la hiérarchie de fichiers

Se reporter à 3.1.1, p. 15 pour les notions de répertoire de travail et d'accueil.

`pwd` **p**rint **w**orking **d**irectory : affiche le chemin absolu du répertoire courant
`cd rep1` **c**hange **d**irectory : choisit `rep1` comme répertoire de travail
`cd` permet de revenir dans le répertoire d'accueil d'où que l'on parte
`cd ..` permet de revenir au répertoire père du répertoire de travail
`cd -` permet de revenir au répertoire de travail précédent dans l'historique

2.3.11 Manipuler des répertoires avec `mkdir` et `rmdir` (*cf.* chap. 3)

`mkdir rep1` **m**a**k**e **dir**ectory : crée le sous-répertoire de nom `rep1`
`mkdir rep2 rep3` crée les sous-répertoires `rep2` et `rep3` dans le répertoire de travail
`rmdir rep1` **rm**ove **dir**ectory : supprime le répertoire `rep1` s'il est vide

Les commandes `cp` et `mv` admettent aussi des répertoires comme arguments.

`cp f1 f2 rep1` recopie les fichiers `f1` et `f2` dans le répertoire `rep1`
`cp -r rep1 rep2` recopie la branche partant de `rep1` vers `rep2` (créé par la copie)
`mv f1 rep1` déplace `f1` dans le répertoire `rep1` (existant) sous le nom `f1`

21. Ils sont généralement plus lents, mais restent rapides à la décompression notamment l'algorithme LZMA (Lempel-Ziv-Markov chain-Algorithm) utilisé par `unlzma` et `xz`, et sont parfois choisis pour distribuer des outils logiciels (par exemple les pages du manuel `man`).

22. Attention : dans les codages multi-octets, en particulier en unicode, on doit distinguer octet et caractère, qui peut s'écrire sur plusieurs octets. L'option `-m` permet d'afficher le nombre de caractères. L'option `-m` de la commande `wc` est sensible à l'environnement de contrôle local (*cf.* 15.4, p. 106) et plus précisément au codage des caractères (*cf.* 4.1.2, p. 30) .

`mv f1 rep1/toto` déplace `f1` dans le répertoire `rep1` (existant) sous le nom `toto`
`mv rep1 rep2` renomme le répertoire `rep1` en `rep2`
`mv rep1 chemin/` déplace la branche sous le répertoire `rep1` sous le répertoire `chemin`

2.4 Environnement réseau

Le système UNIX est conçu pour s'intégrer dans un environnement de machines reliées en réseau. Ce réseau, qu'il soit filaire ou sans fil (WIFI), est notamment exploité pour :

— échanger des messages avec des utilisateurs distants via le courrier électronique
— se connecter à distance sur un serveur via `slogin`
— échanger des fichiers avec une machine distante
— naviguer sur les serveurs web du réseau

2.4.1 Courrier électronique

L'envoi et la réception de messages par courrier électronique s'effectuent à l'aide d'un programme de gestion du courrier. Plusieurs clients de courrier en mode texte, ont été développés, parmi lesquels `mail`, `pine` et `alpine`. D'autres outils permettent de gérer le courrier en mode texte, comme l'éditeur `emacs`. Mais les clients en mode graphique comme par exemple `thunderbird` sont maintenant plus populaires. Enfin, certains sites disposent d'un service de `webmail` (par exemple https://webmail.etu.upmc.fr) qui permet d'accéder via un simple navigateur à sa boîte aux lettres personnelle après authentification.
La forme complète d'une adresse de messagerie électronique est en général [23]
prenom.nom`@`*domaine_internet*
Exemples d'adresses électroniques :
`Sofian.Teber@lpthe.jussieu.fr`
`Jacques.Lefrere@upmc.fr`
Adresse électronique officielle des étudiants à l'UPMC : *Prenom.Nom*`@etu.upmc.fr`.

2.4.2 Connexion à distance via `slogin`

Si un utilisateur est connecté sur une machine *localhost* qualifiée de locale, et s'il possède un compte sur une machine distante *dist_host* dans le domaine [24] *domain* dont l'accès est autorisé, il peut s'y connecter grâce à la commande sécurisée par cryptage `slogin`. Le mécanisme d'authentification sur la machine distante peut exiger une saisie du mot de passe ou s'appuyer sur un échange de clefs.

syntaxe

```
slogin user@dist_host.domain
```

Plus généralement, il est possible de passer des commandes sur la machine distante grâce à la commande [25] :
`ssh` *user@dist_host.domain dist_cmd*

2.4.3 Transfert de fichiers à distance via `scp` et `sftp`

L'utilisateur peut échanger des fichiers personnels entre deux machines, sans ouvrir de session sur la machine distante, via la commande `scp`, selon une syntaxe proche de celle de

23. Supprimer les signes diacritiques (accents, cédille, ...) éventuels dans les noms et prénoms et utiliser le tiret « - » pour les noms ou prénoms composés.
24. Par exemple `dsi.upmc.fr` désigne le sous-sous-domaine de la DSI inclus dans le sous-domaine de l'Université Pierre et Marie Curie, lui même appartenant au domaine géographique `fr`.
25. Bien noter que le shell *local* interprétera les caractères spéciaux non protégés de la commande ; si on souhaite que ceux-ci ne soient interprétés que par le shell distant, il faut les protéger soit individuellement par un « \ », soit par des délimiteurs de protection faible « " » ou forte « ' » (*cf.* 11.2.4, p. 83).

`cp`, en préfixant le chemin d'accès des fichiers distants par *`user@dist_host.domain`* : (le répertoire par défaut est alors le répertoire d'accueil de l'utilisateur distant). Le mécanisme d'authentification est le même qu'avec `slogin`, et les échanges sont aussi sécurisés par cryptage.

syntaxe

```
scp [[user1@]host1:]file1 [[user2@]host2:]file2
```

`scp` *`file1`* `[`*`user2@`*`]`*`host2:file2`* local vers distant
`scp` `[`*`user1@`*`]`*`host1:file1`* *`file2`* distant vers local

La commande `scp`, qui demande le mot de passe à chaque invocation, est mal adaptée pour transférer un nombre important de fichiers. On lui préfère `sftp` qui permet d'ouvrir avec une seule authentification une session de transfert de fichiers pendant laquelle il est possible de lister les fichiers distants, de se déplacer dans la hiérarchie et de d'effectuer des transferts dans les deux sens. Un utilisateur connecté sur une machine *`localhost`* qualifiée de locale, et qui possède un compte sur une machine distante *`dist_host`* peut aussi échanger des fichiers personnels entre ces deux machines en ouvrant une session `sftp` (**s**ecure **f**ile **t**ranfer **p**rotocol)

syntaxe

```
sftp user@dist_host.domain
```

Après l'authentification sur le serveur distant, l'utilisateur peut importer des fichiers distants grâce à la requête `get` *`dist_file`*, ou exporter des fichiers vers la machine distante par `put` *`local_file`* ; d'autres requêtes telles que `ls`, `cd`, `pwd` s'adressent au serveur distant alors que `lcd` (**l**ocal **c**hange **d**irectory) concerne la machine locale ; `exit` ou `quit` permet de terminer la session `sftp`.

2.4.4 Navigateur

Les explorateurs Web (`lynx`, `mozilla-firefox`, `opera`, `konqueror`, `amaya`, ...) permettent d'accéder à des ressources d'informations distribuées sur le réseau Internet. Cet accès se fait par l'intermédiaire de différents protocoles, comme `ftp` (**F**ile **T**ransfer **P**rotocol), `http` (**H**yper**t**ext **T**ransport **P**rotocol), ou sa version `https` sécurisée par cryptage.
Les ressources fournies à travers le protocole `http` sont appelées pages Web. Elles sont localisées grâce à une adresse dite URL (**U**niversal **R**esource **L**ocator).

Exemples d'URL :
`file:/home/lefrere/M1/Doc/Unix/` sur la machine locale
`http://www.formation.jussieu.fr/ars/2011-2012/UNIX/cours/`
`http://www.w3.org/TR/xhtml1`

Autres outils La commande `wget` est très efficace pour télécharger des fichiers ou (récursivement) des hiérarchies de fichiers depuis un serveur (`ftp` ou `http`) dès que l'on connait leur URL, qui peut être obtenue via un navigateur en utilisant l'option « copier l'adresse du lien » obtenue avec le clic droit de la souris. En effet `wget` est outil non interactif très robuste qui peut s'adapter à des liaisons de faible débit et gérer lui-même les reprises de transfert en cas d'échec. ⇐ ♥
Exemple [26] de téléchargement de l'introduction à LINUX de Machtelt GARRELS :
`wget "http://tldp.org/LDP/intro-linux/intro-linux.pdf"`
Noter que le système de gestion de paquets RPM de la distribution Red-Hat utilise un autre outil de transfert en ligne de commande, `curl` qui gère de nombreux protocoles (`ftp`, `http`, `https`, ...).

26. Ne pas oublier de protéger par des guillemets doubles les caractères spéciaux comme le « : » (*cf.* 11.4.4, p. 85) de l'interprétation par le shell (*cf.* 11.2.4, p. 83).

Chapitre 3

Hiérarchie de fichiers

3.1 Arborescence des fichiers

Sous UNIX, l'ensemble des fichiers est structuré sous la forme d'une hiérarchie de répertoires et de fichiers constituant un arbre unique. Un exemple partiel d'arborescence est donné dans la figure 3.1.

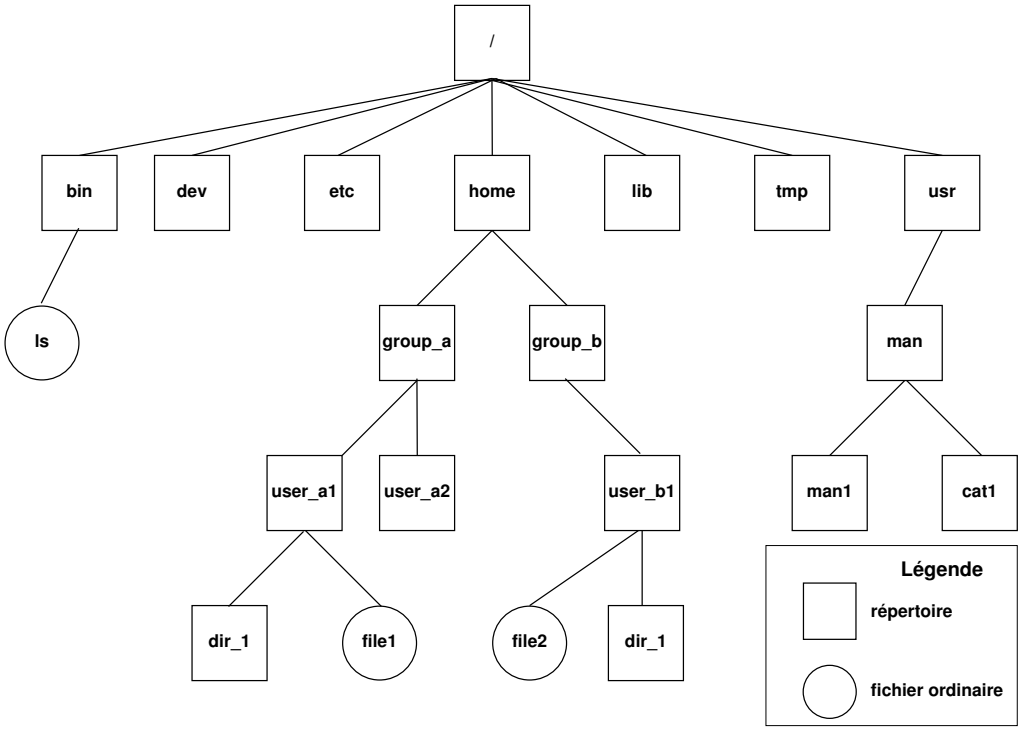


FIGURE 3.1 – Arborescence des fichiers UNIX

— sa racine (root) est le répertoire qui contient tous les autres fichiers ; on la désigne par `/`

— ses nœuds sont des sous-répertoires : `/bin` pour les exécutables, `/dev` pour les périphériques, `/etc` pour le système, `/home` pour les utilisateurs, `/tmp` pour les fichiers temporaires, `/usr` pour les outils, ... qui eux-mêmes contiennent des fichiers (feuilles de l'arbre) ou d'autres sous-répertoires (par exemple les groupes d'utilisateurs) qui...

— ses feuilles sont les fichiers.

Remarques : L'arbre unique d'UNIX est purement logique ; plusieurs périphériques amovibles (disques, clefs-USB, ...) peuvent y être « montés » temporairement : cette opération de montage peut être considérée comme la « greffe » de la branche de la hiérarchie de fichiers du périphérique amovible en un point de montage (par exemple `/media/cdrom`, `/media/removable`) de l'arbre UNIX.

Sous windows, le séparateur est la contre-oblique `\` (antislash) ; les périphériques sont désignés par une lettre préfixe suivie de « `:` », par exemple `C:\` ou `D:\` qui constituent chacun un arbre.
Noter que certains systèmes de fichiers utilisés sous windows ne distinguent pas minuscules et majuscules dans les noms de fichiers, ce qui peut être source de confusion en cas de montage d'une partie de disque windows par une machine unix. ⇐⚠

Répertoire d'accueil : Au sein de cette hiérarchie, chaque utilisateur se voit attribuer par l'administrateur (*cf.* 2.1.1, p. 4) un répertoire d'accueil (home directory) personnel, d'où part une branche de l'arbre qui contient les sous-répertoires et fichiers ordinaires qui lui appartiennent [1]. La variable d'environnement `HOME` contient le chemin absolu d'accès au répertoire d'accueil (*cf.* 11.1.4, p. 80).

3.1.1 Chemins d'accès des fichiers et répertoires

Tout fichier UNIX peut être référencé par son chemin d'accès (path), c'est-à-dire la description du chemin qu'il faut parcourir dans l'arborescence à partir d'un certain répertoire pour atteindre le fichier en question. Le chemin est spécifié par les noms des répertoires séparés par des `/` et suivis du nom du fichier.

Répertoire de travail :
Pour simplifier la désignation des fichiers, on introduit la notion de *répertoire courant ou de travail* (working directory) dans lequel les fichiers peuvent être nommés sans préciser de chemin. Le répertoire courant est désigné par un point « `.` ». La commande `pwd` permet d'afficher le chemin absolu du répertoire de travail ; le répertoire de travail peut évoluer en cours de session grâce à la commande interne `cd` (*cf.* 2.3.10, p. 11). Lors de l'ouverture de la session UNIX, le répertoire de travail est par défaut le répertoire d'accueil de l'utilisateur.
Ainsi, pour désigner un fichier, on peut indiquer, suivant que l'on part de la racine ou du répertoire courant :

— **un chemin absolu** : il comporte la liste complète des répertoires traversés depuis la racine, et commence toujours par `/` [2]
Exemples : `/usr/man/man1/ls.1`, `/home/group_a/user_a1`, (*cf.* figure 3.1, p. 14)

— **un chemin relatif** : il comporte la liste des répertoires à parcourir depuis le répertoire courant jusqu'au fichier ou répertoire choisi. Il ne commence jamais par `/` (on peut dire qu'il commence par « `.` ») et doit passer par un nœud commun à la branche de départ (répertoire courant) et la branche d'arrivée. La notation « `..` » permet de remonter d'un niveau dans la hiérarchie, c'est-à-dire au *répertoire père*.
Exemples, partant de `/home/group_a/user_a1` :
`dir_1`, `../`, `../user_a2`, `../../group_b/user_b1`, (*cf.* figure 3.1, p. 14)

Raccourcis pour les répertoires d'accueil

Enfin, le symbole « `~` » permet de référencer le répertoire d'accueil d'un utilisateur quelconque [3] sous la forme `~`*`user`*. Il se comporte donc comme un raccourci d'un chemin *absolu*. Par exemple, `~user_a2` est équivalent à `/home/group_a/user_a2/`. Enfin « `~` » sans nom d'utilisateur désigne votre propre répertoire d'accueil [4].

1. En général, un utilisateur n'a pas le droit de créer de fichier en dehors de cette branche du système de fichiers UNIX, sauf dans le répertoire `/tmp` (*cf.* 3.1.4, p. 20).
2. Si le chemin commence par `~`, c'est un raccourci de chemin absolu qui, une fois développé par le shell, débutera par `/`
3. Ce qui suit « `~` » doit impérativement être un nom d'utilisateur, dont le répertoire d'accueil est unique, mais surtout pas un nom quelconque de fichier ou de sous-répertoire dont il peut exister plusieurs versions dans des répertoires différents.
4. Donc `~` est équivalent à `${HOME}`, *cf.* 11.1.4, p. 80

3.1.2 Visualisation d'une branche avec `tree`

♥ ⇒ La commande `tree` permet de représenter[5] une branche de la hiérarchie de fichiers. Plusieurs options y ont le même sens que dans `ls` : en particulier, `-F` ajoute un « `/` » à la fin des noms de répertoires, `-a` permet d'afficher les fichiers cachés. L'option `-d` permet de n'afficher que les répertoires.

Par exemple, avec comme répertoire de travail le répertoire `/home` de l'arbre de la figure 3.1, p. 14, la commande `tree -F .` affiche :

```
.
|-- groupe_a/
|   |-- user_a1/
|   |   |-- dir_1/
|   |   `-- file1
|   `-- user_a2/
`-- groupe_b/
    |-- user_b1/
    |   |-- dir_1/
    |   `-- file1
    `-- user_b2/

8 directories, 2 files
```

L'option `-f` (**f**ull) permet d'afficher le chemin complet à partir du répertoire donné en argument de la commande. `tree -F -f .` affiche :

```
.
|-- ./groupe_a/
|   |-- ./groupe_a/user_a1/
|   |   |-- ./groupe_a/user_a1/dir_1/
|   |   `-- ./groupe_a/user_a1/file1
|   `-- ./groupe_a/user_a2/
`-- ./groupe_b/
    |-- ./groupe_b/user_b1/
    |   |-- ./groupe_b/user_b1/dir_1/
    |   `-- ./groupe_b/user_b1/file1
    `-- ./groupe_b/user_b2/

8 directories, 2 files
```

3.1.3 Changement de répertoire de travail : exemples

La commande interne `cd` (**c**hange **d**irectory) permet de changer de répertoire de travail. Sans paramètre, `cd` permet d'accéder au répertoire d'accueil, alors que `cd` *`rep`* permet d'accéder à un répertoire quelconque spécifié par son chemin (absolu ou relatif).
Les exemples illustrés ci-après partent du répertoire courant `/home/group_a/user_a1`.
La commande `cd␣.` ne modifie pas le répertoire courant car « `.` » désigne le répertoire courant.

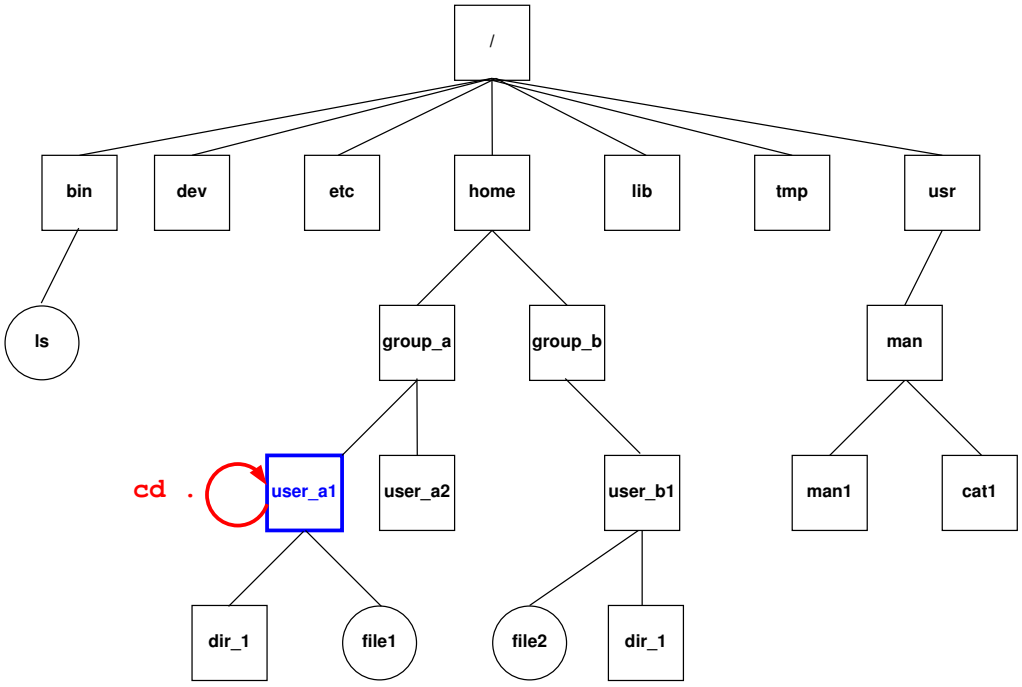


FIGURE 3.2 – La commande `cd .` laisse dans le répertoire courant `/home/group_a/user_a1`.

5. L'affichage avec `tree` respecte visuellement la structure d'arbre du système de fichiers, mieux que celui des gestionnaires graphiques de fichiers, comme par exemple `Thunar`, associé à `xfce`.

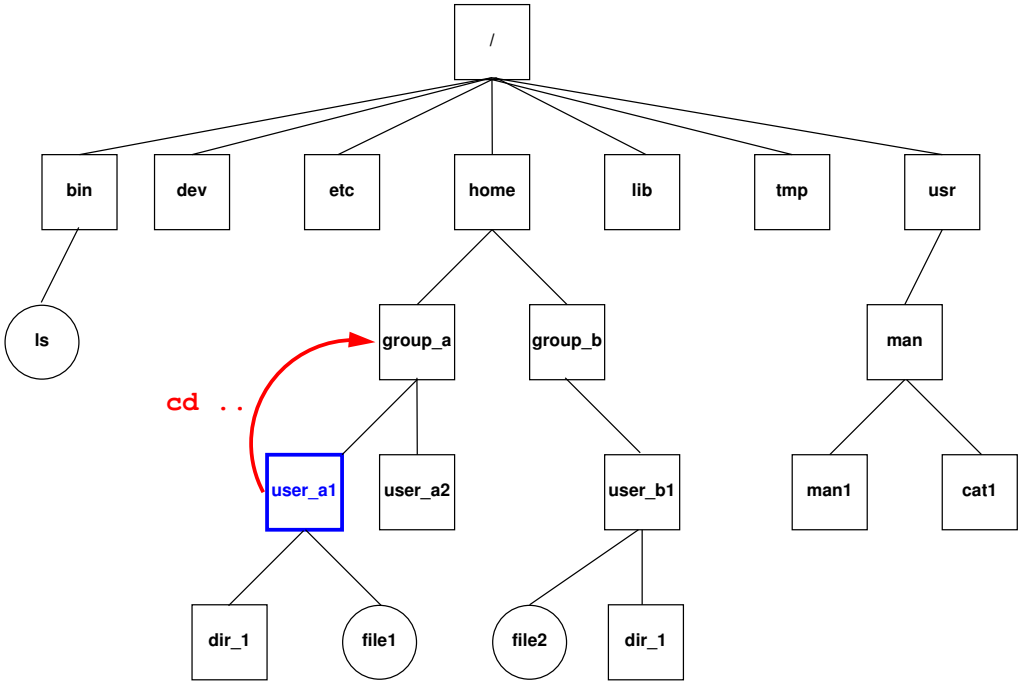


FIGURE 3.3 – Partant de `user_a1`, la commande `cd ..` déplace dans le répertoire père, soit `group_a`.

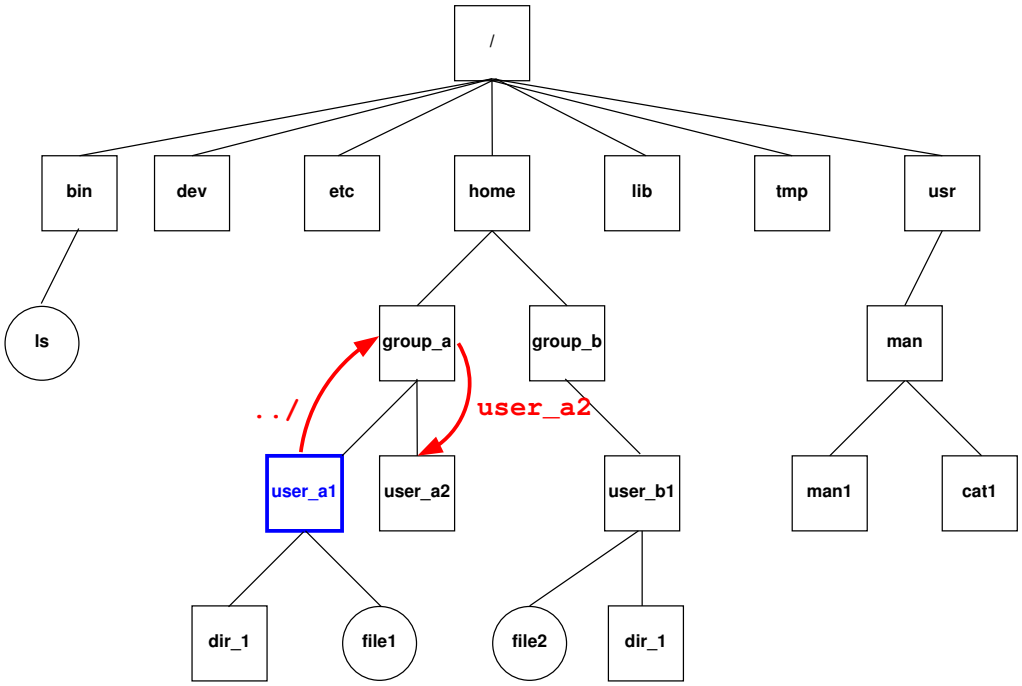


FIGURE 3.4 – Partant de `user_a1`, la commande `cd ../user_a2` déplace dans le répertoire `user_a2`

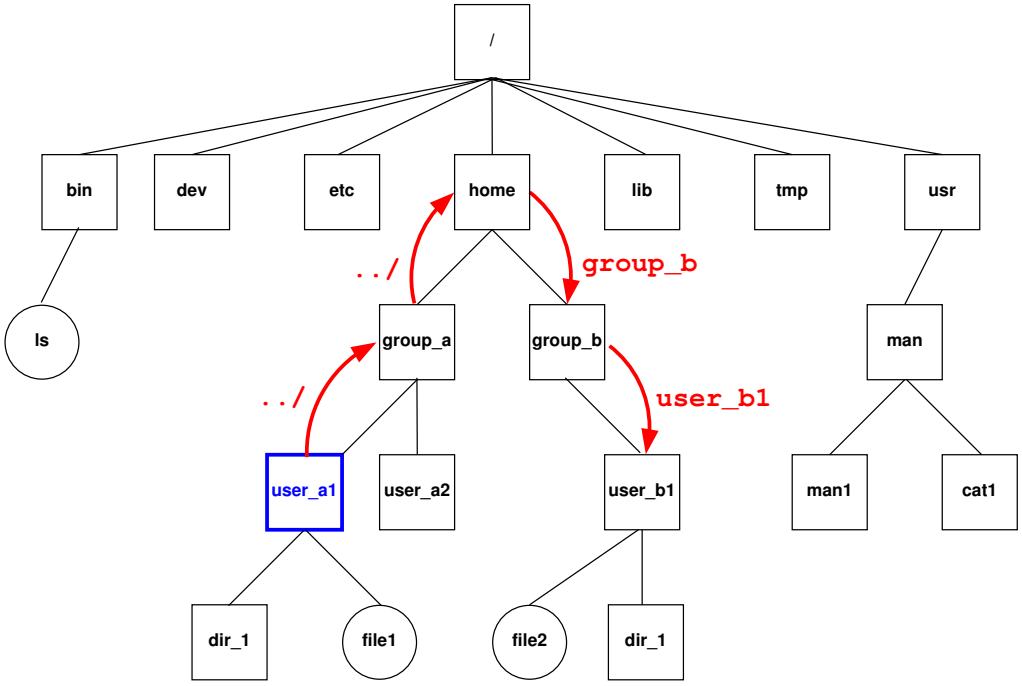


FIGURE 3.5 – Partant de `user_a1`, la commande `cd ../../group_b/user_b1` déplace dans le répertoire `user_b1`.

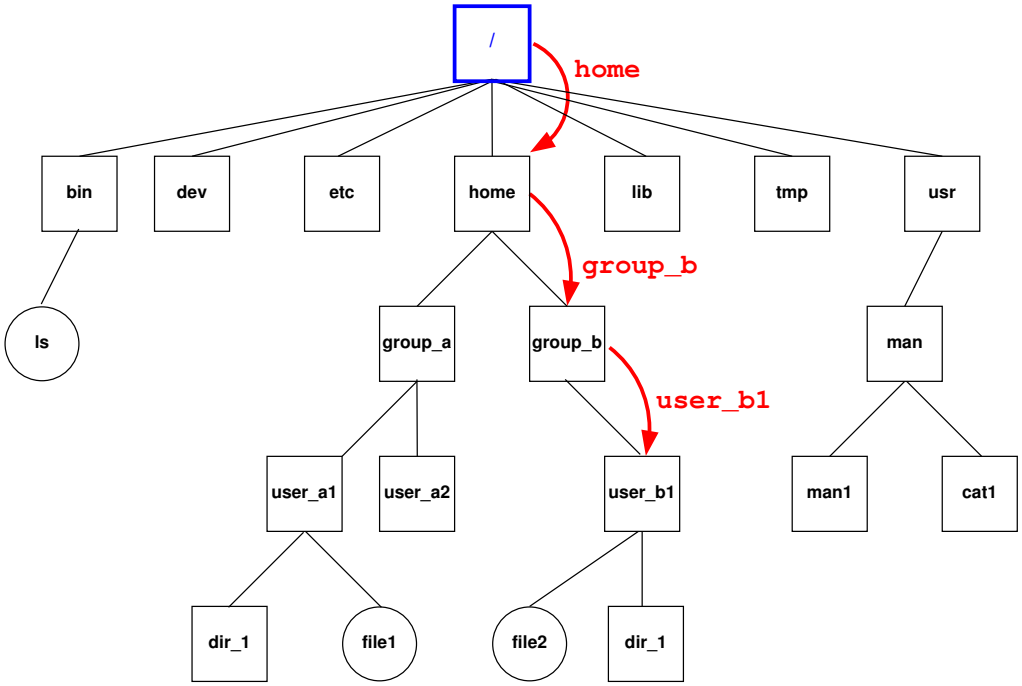


FIGURE 3.6 – La commande `cd /home/group_b/user_b1` fait du répertoire `user_b1` le répertoire courant quel que soit le répertoire de départ car elle utilise un chemin **absolu**.

3.1.4 Attributs des fichiers—droits d'accès

Affichage des droits avec `ls -l`

La commande `ls -l` permet d'afficher les attributs des fichiers, dans l'ordre suivant : type et droits d'accès, nombre de liens, nom du propriétaire, nom du groupe (*cf.* 2.2.4, p. 8), taille et date.

```
-rwxr-xr-x   1 p6m1mni  p6pfal          1346 Oct 08 18:24 MNI.profile
```

Le type et les droits d'accès sont affichés à l'aide de 10 caractères :

1. Le premier caractère représente le type du fichier :
   - `-` pour un fichier ordinaire,
   - `l` pour un lien (**l**ink),
   - `d` pour un répertoire (**d**irectory),
   - ...
2. Les neuf suivants doivent être interprétés par groupes de trois :
   - les trois premiers représentent les droits du propriétaire symbolisé par `u` (**u**ser),
   - les trois suivants représentent les droits du groupe auquel il appartient, `g` (**g**roup),
   - les trois derniers représentent les droits des autres utilisateurs `o` (**o**thers [6]).

   Pour chaque public [7], trois droits principaux peuvent être accordés, dans l'ordre :
   - lecture `r` (**r**ead),
   - écriture `w` (**w**rite),
   - exécution `x` (e**x**ecute).

   Si un droit n'est pas accordé, le signe moins « `-` » vient remplacer `r`, `w` ou `x`.

type	propriétaire	groupe	autres
`-/d/l`	**u**ser	**g**roup	**o**thers
`-`	`r w x`	`r w x`	`r w x`

♠ **Droits d'endossement**

À la place de l'attribut `x` sur un fichier exécutable, on peut positionner la permission `s` [8] : celui qui exécute le fichier acquiert alors temporairement les droits du propriétaire [9] (set uid bit), ou du groupe (set gid bit) lors de l'exécution du fichier [10].

Exemple : droits sur le fichier `/etc/passwd` Le fichier `/etc/passwd` [11] contient les mots de passe codés de tous les utilisateurs. Son propriétaire est l'administrateur du système (root). Il ne peut pas accorder de permission permanente en écriture sur ce fichier aux utilisateurs.

```
-rw-r--r--    1 root     root         1199 Sep  6 17:16 /etc/passwd
```

Mais chaque utilisateur doit pouvoir modifier son mot de passe grâce à la commande `passwd` dont le code est situé dans le fichier exécutable `/usr/bin/passwd`.
L'attribution de la permission `s` à ce fichier, dont le propriétaire est l'administrateur, permet d'accorder temporairement les droits du super-utilisateur à chaque utilisateur authentifié lors de l'exécution de la commande `passwd` : en particulier, cela lui permet d'écrire son mot de passe codé dans le fichier `/etc/passwd`.

```
-r-s--x--x    1 root     root        16084 Apr 27 23:37 /usr/bin/passwd
```

6. Ne pas confondre avec owner.
7. Noter que si un droit est refusé au titre du groupe par exemple, un utilisateur ne peut pas invoquer un droit sur une entité plus vaste (les autres ici) pour accéder à un fichier.
8. Pour des raisons de sécurité, certaines implémentations d'UNIX n'autorisent pas l'attribut `s` pour les shell-scripts.
9. On qualifie alors d'effective user le propriétaire du fichier exécuté, par rapport au real user qui lance la commande. La permission `s` est qualifiée de droit d'endossement.
10. Il existe un mécanisme individualisé de gestion des droits via la commande `sudo` (*cf* 15.2.7, p. 105).
11. Dans les systèmes sécurisés, les mots de passe codés sont en fait stockés dans un fichier `/etc/shadow`, qui n'est plus accessible qu'à l'administrateur y compris en lecture.

♠ Restriction au propriétaire du droit de destruction

Notons aussi qu'il est possible de placer sur un répertoire un droit qui interdit de détruire un fichier dont on n'est pas propriétaire. Cette restriction est en général imposée sur le répertoire `/tmp`. Elle est représentée par `t` à la place de `x` en dernière position :
`drwxrwxrwt 25 root root 4096 2013-09-15 22:02 /tmp/`

Cas des liens symboliques

Noter que les liens symboliques ont tous leurs droits ouverts : cela implique qu'ils ne limitent pas les droits qui sont en fait portés par la cible du lien.

Cas des répertoires

Rappelons que l'affichage des attributs d'un répertoire (et non des fichiers qu'il contient)
⚠⇒ se fait avec la commande `ls -ld`. La signification des droits sur les répertoires est un peu particulière :

- `w` permet d'ajouter, de renommer, de supprimer des fichiers dans le répertoire (mais ce droit n'est pas nécessaire pour modifier le contenu d'un fichier)
- `r` nécessaire pour afficher la liste des fichiers du répertoire
- `x` permet d'agir sur les fichiers du répertoire, donc de le traverser, d'en faire son répertoire de travail (même si on ne peut pas afficher la liste de son contenu)

Si on souhaite permettre aux autres d'explorer un répertoire, il faut donc leur accorder les droits `rx`.
Noter que la permission `x` sur le répertoire est nécessaire pour afficher les attributs des fichiers qu'il contient, notamment avec `ls -l` (*cf.* 2.3.1, p. 9), alors que c'est le droit `r` [12] qui permet d'afficher la liste sans les attributs.

Modification des droits avec `chmod`

Seul le propriétaire [13] d'un fichier peut modifier ses droits d'accès par la commande :
`chmod` *`mode fichier`*
où *`mode`* représente une concaténation de trois éléments :

le ou les publics concernés : `u`, `g`, `o` ou toute combinaison ou enfin `a` (**a**ll soit `ugo`),
suivi de l'opération à réaliser :
`=` (définition, à l'exclusion de tout autre droit pour un public donné),
`+` (ajout),
`-` (suppression),
suivie des droits considérés `r`, `w`, `x` ou une concaténation des ces droits.

Ainsi, la commande `chmod g+x toto` donne au groupe, en plus des droits d'accès actuels, celui d'exécuter le fichier `toto`. On peut enfin fournir une liste de modes selon la syntaxe précédente, en séparant les éléments par des virgules, sans espace.
Pour chacun des trois publics, les droits d'accès peuvent aussi être représentés de façon numérique par la valeur exprimée en base huit selon les poids des trois droits suivants : `r`=4, `w`=2, `x`=1. Les droits sont donc représentés par trois [14] chiffres en octal.

Ainsi les droits d'un fichier ouvert à toutes les opérations pour tous sauf celle d'écriture réservée au propriétaire seront caractérisés par le mode numérique 755.

public	u	g	o
symbolique	rwx	r-x	r-x
binaire	111	101	101
octal	7	5	5

12. Dans le cas où le répertoire comporte le droit `r`, mais pas `x`, la commande `ls -F` ou avec colorisation, qui fait appel aux attributs pour « décorer » le listing, va provoquer un message d'erreur, mais affichera cependant la liste. Si la commande `ls` est un alias imposant une de ces options, il faut lancer la commande native via `\ls` pour afficher la liste sans provoquer d'erreur.
13. ainsi que `root`, l'administrateur de la machine.
14. On doit ajouter à gauche un autre chiffre octal (0 zéro par défaut) si on prend en compte le suid bit `s` (poids 4), le guid bit (poids 2) et le sticky bit `t` ou le bit affecté aux répertoires restreignant les destructions aux fichiers dont on est propriétaire (poids 1).

Exemple : après `chmod 600 f1`, seul son propriétaire peut lire et écrire sur `f1`. Cette commande équivaut à `chmod u=rw,go= f1` en symbolique.

Droits par défaut : la commande interne `umask`

Les droits par défaut attribués lors de la création d'un fichier dépendent de l'outil de création [15], mais ils ne peuvent pas être plus étendus que ceux définis par un masque dont chaque bit à 1 bloque l'attribution du droit qui se situerait à sa position. La valeur de ce masque peut être affichée par la commande interne (*cf.* 15.1.1, p. 101) `umask` sans argument. Elle peut être modifiée [16] par `umask` *`mode`*. L'option `-S` de `umask` permet d'exprimer ce masque en termes symboliques. Le masque minimal consiste à bloquer le droit d'écriture sauf pour le propriétaire des fichiers, c'est à dire `umask -S u=rwx,g=rx,o=rx`, soit `umask 022`. Mais une valeur couramment adoptée est `umask 027`, qui bloque tous les droits en dehors du groupe. Cependant, une fois le fichier créé, la commande `chmod` (*cf.* 3.1.4, p. 20) permet de modifier ses droits indépendamment du masque courant. Pour qu'un changement de valeur du masque ne soit pas oublié en fin de session, il doit être effectué dans les fichiers d'initialisation du shell.

⇐♥

Cas des systèmes de fichiers non unix Le système de droits que nous venons de décrire est propre au système UNIX. Si on effectue le montage d'un disque externe ou d'une clef USB initialement sous windows, il est probable qu'ils soient vus sous UNIX avec tous les droits ouverts, y compris le droit d'exécution pour des fichiers où il n'a aucun sens.

⇐⚠

3.2 Autres commandes de gestion des fichiers

3.2.1 Rechercher récursivement des fichiers avec `find`

La commande `find` est une commande très puissante qui permet de rechercher dans la hiérarchie des fichiers selon de nombreux critères (nom, date, taille, droits, type, ...) qui peuvent être combinés entre eux, et d'appliquer des commandes aux fichiers sélectionnés. Elle répond à la syntaxe suivante :

syntaxe

```
find  répertoire  critère(s)  action
```

La recherche s'effectue **récursivement** dans toute la branche [17] située sous le répertoire indiqué en premier argument et les chemins affichés pour les fichiers trouvés partiront tous de *`répertoire`*.

Critères de recherche de `find`

Les **critères de sélection** les plus courants sont :

`-name` *`motif`*	nom selon un motif (`-iname` pour ignorer la casse)
`-size` *`entier`*`[ckM]`	taille (`+/-` entier plus grand/plus petit que l'entier spécifié) unité : octet (`c`), kilo-octet (`k`), méga-octet (`M`)
`-newer` *`fichier`*	plus récent qu'un fichier
`-type` *`T`*	de type donné (`f`=ordinaire, `d`=répertoire, `l`=lien symbolique)

15. Par exemple, un fichier créé par un éditeur de texte, n'est pas exécutable par défaut : s'il s'agit d'un fichier de commandes (*cf.* 12.1, p. 87), il faut le rendre exécutable par la commande `chmod +x`. En revanche, la compilation et l'édition de lien d'un fichier source fortran ou C produit un fichier `a.out` immédiatement exécutable sans avoir à utiliser `chmod`... sauf si le masque courant n'a pas permis d'attribuer ces droits !

16. Le changement de masque ne modifie en rien les droits des fichiers existants. Il limite seulement les droits pour les fichiers créés ultérieurement.

17. Il est possible de limiter la profondeur d'exploration de `find` dans les sous-répertoires par l'option `-maxdepth` suivie d'un nombre de niveaux.

Actions sur les fichiers trouvés

Les **actions** les plus usitées que `find` peut lancer sont :

`-print`	affichage de la liste des fichiers (un par ligne avec leur chemin relatif ou absolu selon le choix fait pour le premier argument de la commande)
`-ls`	affichage de la liste des fichiers avec leurs attributs (comme par la commande `ls -dils`[18])
`-exec` *cmd*	exécution de la commande UNIX *cmd* pour chacun des fichiers sélectionnés (syntaxe délicate)

L'usage le plus simple de cette commande est de rechercher dans une partie de l'arborescence les fichiers portant un nom donné :

`find . -name x.c -print` (*cf.* fig. 3.7, p. 22)

affiche la liste de tous les fichiers nommés `x.c` dans la branche située sous le répertoire courant.

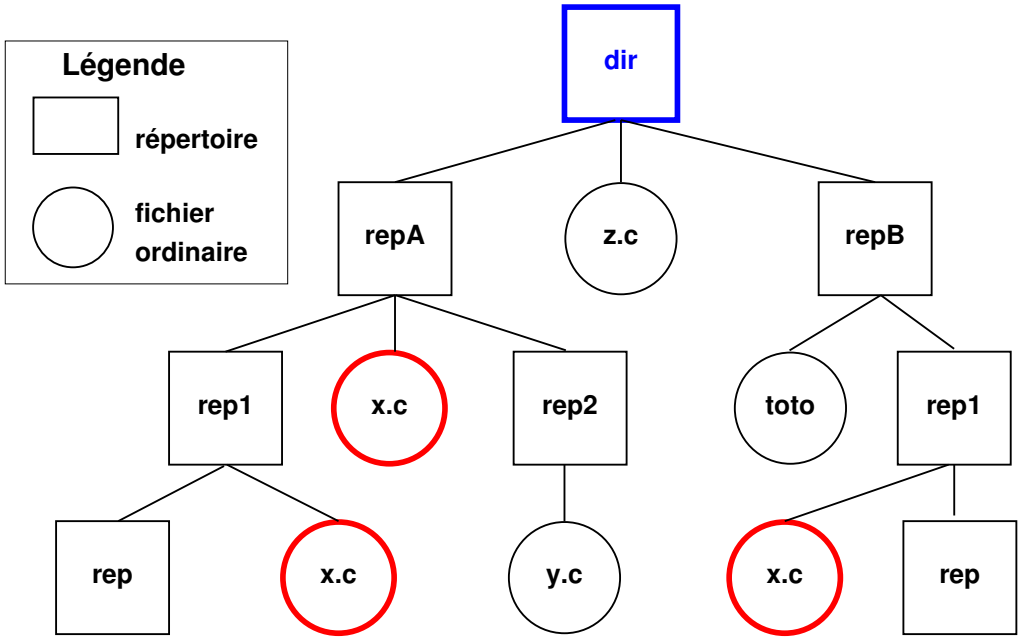


FIGURE 3.7 – Recherche des fichiers nommés `x.c` à partir du répertoire `dir` avec la commande `find . -name x.c -print` si `dir` est le répertoire de travail. Trois fichiers sont trouvés et leurs chemins affichés à partir du répertoire de travail sous la forme `./repA/x.c` par exemple. Si le répertoire de travail était `repA`, pour rechercher aussi à partir de `dir`, il faudrait lancer la commande `find ../ -name x.c -print`.

Interactions avec le shell

⚠⇒ Pour rechercher des fichiers dont le nom correspond à un motif générique, il faut protéger les caractères spéciaux du motif comme « `*` » de l'interprétation par le shell (*cf.* 11.2.4, p. 83), qui se ferait sinon dans le répertoire de travail quel que soit le répertoire de recherche.

La commande suivante recherche à partir du répertoire courant, tous les fichiers dont le nom se termine par `.c` (*cf.* fig. 3.9, p. 23 et fig. 3.10, p. 24).

```
find . -name '*.c' -print
```

Pour rechercher chez tous les utilisateurs à partir du répertoire `/home`, tous les fichiers dont le nom se termine par le suffixe `.f90`, il est préférable de se débarrasser des messages d'erreur qui surviennent à cause des permissions d'accès restreintes, en redirigeant la sortie d'erreur vers un fichier « poubelle ».

`find /home -name '*.f90' -print  2>/dev/null` (*cf.* 10.5.1, p. 76)

18. L'option `-d` est ici nécessaire pour afficher les attributs des répertoires et non celles des fichiers qu'il contiennent (*cf.* 2.3.1, p. 9).

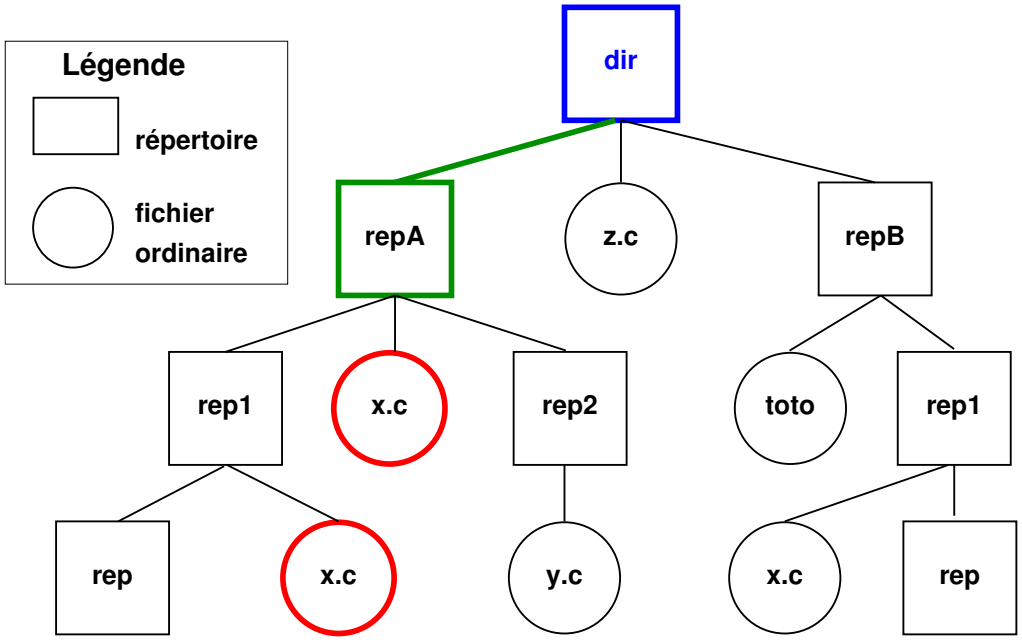


FIGURE 3.8 – Si `dir` est le répertoire de travail, la commande qui permet de rechercher les fichiers nommés `x.c` à partir du sous-répertoire `repA` est `find repA -name x.c -print` Seulement deux fichiers sont trouvés `./repA/x.c` et `./repA/rep1/x.c`

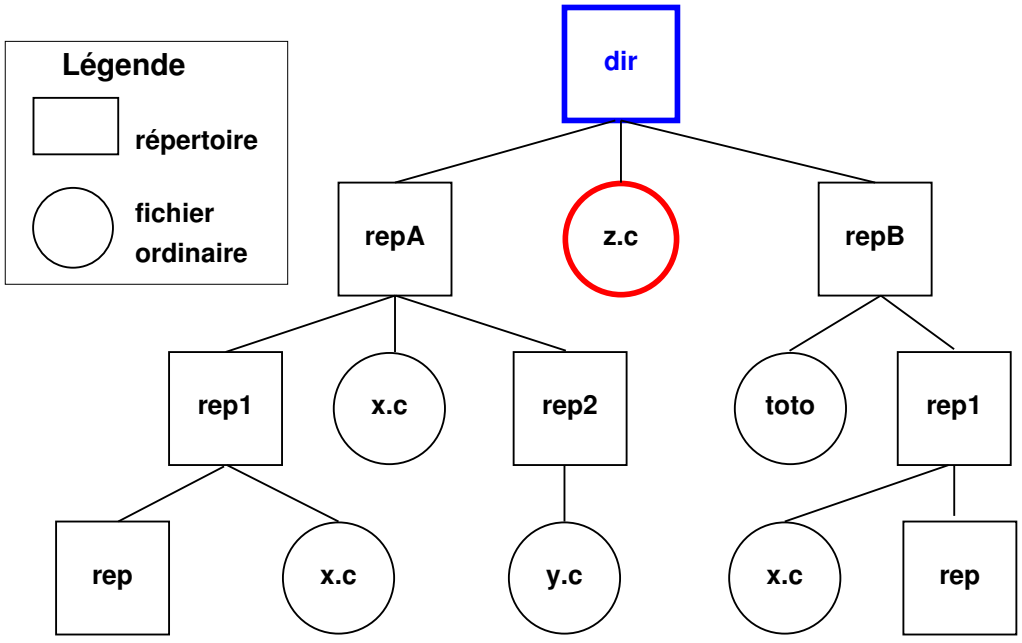


FIGURE 3.9 – Dans la commande `find . -name *.c -print`, le caractère « `*` » est interprété (dans le répertoire courant) par le shell et `*.c` est remplacé par `z.c` avant d'être passé à `find`. Seul le fichier `z.c` est donc trouvé dans la branche `dir` (*cf.* méthode correcte fig. 3.10, p. 24). Noter que si, sans changer de répertoire de travail, la recherche se faisait par `find repA -name *.c -print`, on rechercherait encore un fichier `z.c`, mais sous `repA` : on ne trouverait alors aucun fichier à ce nom.

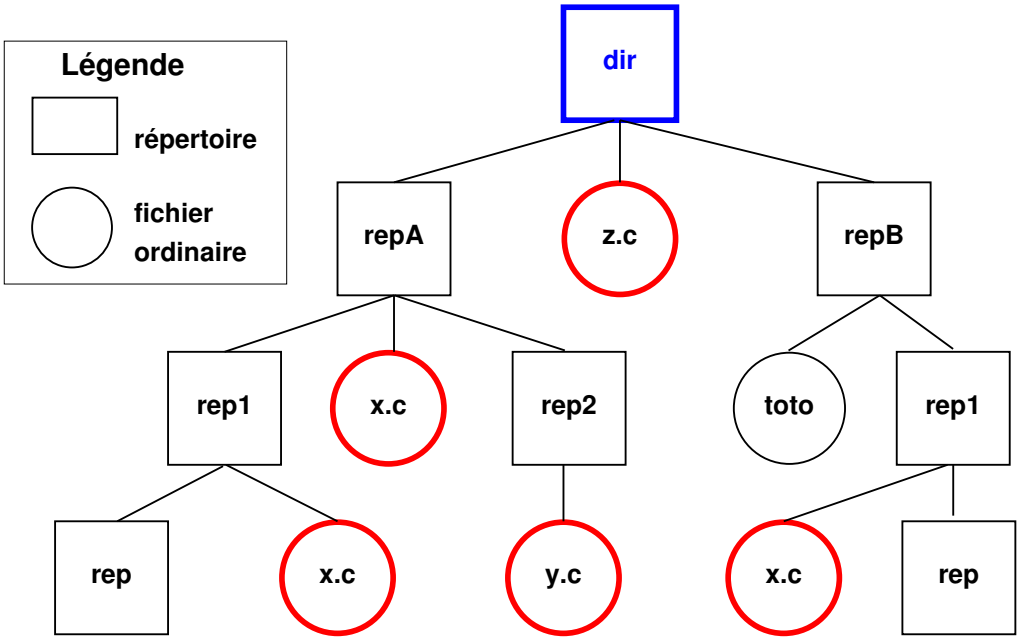


FIGURE 3.10 – Pour éviter l'interprétation du caractère « `*` » par le shell (*cf.* fig. 3.9, p. 23), on le protège en l'entourant par des « `'` ». Ainsi `find repA -name '*.c' -print` permet de trouver les 5 fichiers de suffixe `.c` de la branche `dir`.

Exemples

`find /tmp -size +1000c -size -2000c -print`
affiche la liste des fichiers de taille entre 1000 et 2000 octets sous `/tmp`
`find . -name a.out -exec rm {} \;`
recherche les fichiers `a.out` sous le répertoire courant et les supprime :

> `{}` désigne le nom de chaque fichier trouvé (avec son chemin d'accès)
>
> `\;` indique la fin de la commande [19] à appliquer à chaque fichier. Bien noter qu'indiquer la fin de la commande déclenchée par `exec` est indispensable car on peut par exemple faire suivre la commande `find` d'un tube.

`find ~/src -name "*.c" -exec grep -l math {} \; | wc -l`
permet de compter le nombre de fichiers de suffixe `.c` qui contiennent la chaîne `math` dans le répertoire `src` (situé dans le répertoire d'accueil de l'utilisateur).

3.2.2 Archiver des arborescences de fichiers avec `tar`

La commande `tar` [20] est utilisée pour archiver des hiérarchies de fichiers à des fins de sauvegarde ou de transfert entre deux machines (*cf.* par exemple Fig. 3.11, 3.12 et 3.13).

19. Il faut protéger le caractère `;` de l'interprétation par le shell où est lancée la commande `find` : c'est le rôle de la contre-oblique, qui sera « consommée » par le shell avant interprétation du `find` (*cf.* 11.2.4, p. 83).

20. Le nom de cette commande signifie tape archive, c'est-à-dire sauvegarde sur bande magnétique. Elle présente d'ailleurs des similitudes avec la commande `ar` de gestion des bibliothèques. Avec l'évolution des ressources disque et l'usage du réseau, les fichiers d'archives peuvent maintenant être stockés sur disque et il faut alors les nommer, d'où l'option `-f`.

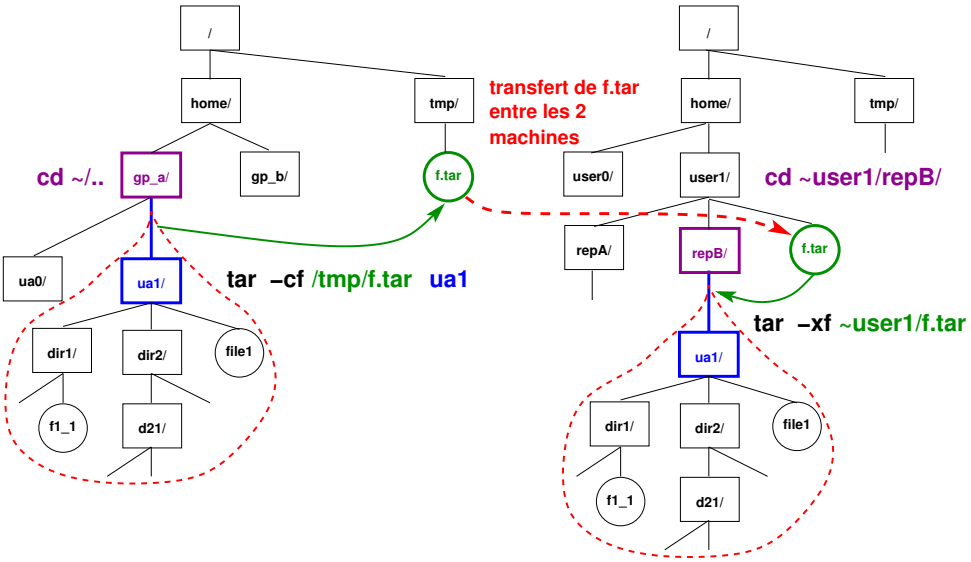


FIGURE 3.11 – Recopie d'une branche via `tar` : création de l'archive `f.tar` de la branche (*cf.* Fig. 3.12, p. 25), transfert de l'archive et extraction sous `repB` (*cf.* Fig. 3.13, p. 26)

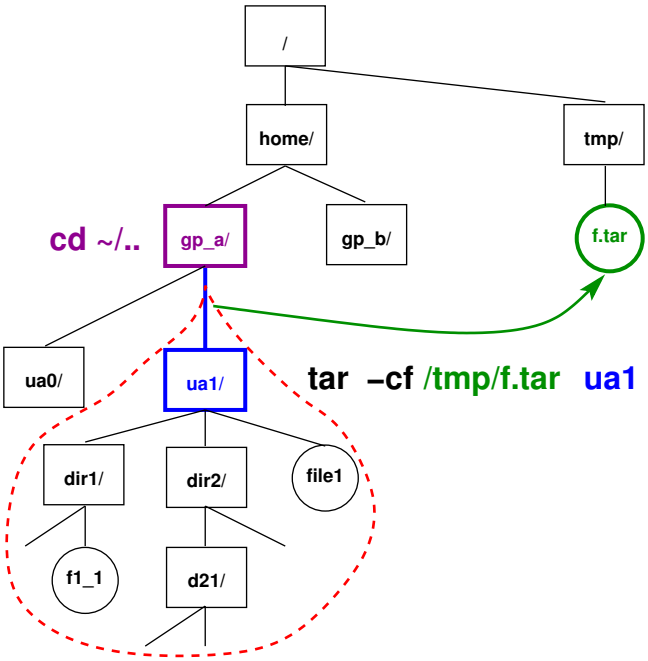


FIGURE 3.12 – Création (sous `/tmp`) de l'archive `f.tar` de la branche de l'utilisateur `ua1` :
1) `cd ~/..`
2) `tar -cf /tmp/f.tar ua1`
Noter que `f` est la dernière option et doit être suivie du nom du fichier d'archive avec son chemin et que le dernier argument doit être un chemin relatif.

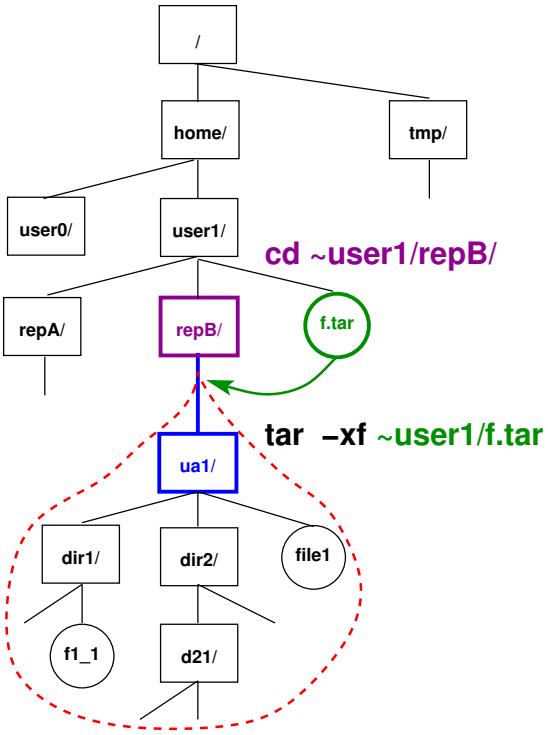


FIGURE 3.13 – Restauration de branche à partir de l'archive :
1) `cd ~/repB/`
2) `tar -xf ~/f.tar`
Noter que la restauration se fait à partir du répertoire courant.

Syntaxe de la commande `tar`

`tar` *`options [archive] répertoire`* pour créer l'archive de la branche *`répertoire`*
`tar` *`options [archive]`* pour exploiter l'archive (liste ou restauration)

Principales actions possibles[21] **(une et une seule par commande) :**
- `-c` (**c**reate) création de l'archive à partir de l'arborescence (*cf.* fig. 3.12) ;
- `-t` (li**t**) liste des fichiers tels qu'ils seront extraits ;
- `-x` (e**x**tract) extraction des fichiers pour restaurer l'arborescence (*cf.* fig. 3.13).

Autres options combinables :
- `-f` *`archive`* (**f**ile) option à argument pratiquement obligatoire qui permet de préciser le nom du fichier d'archive utilisé[22]
- `-v` (**v**erbose) affiche des informations complémentaires
- `-z` (**z**ip) compression (à la création) ou décompression (lors de la lecture) à la volée[23] du fichier d'archive

Exemple de duplication d'un compte

♥ ⇒ **création :** On se place généralement juste au-dessus de la branche à archiver lors de la création pour que la restauration puisse se faire dans un seul répertoire, facile à déplacer en cas d'erreur.

```
cd ~user1/../ ; tar -cvf /tmp/archive.tar user1
```

21. Noter que `tar` admet aussi une syntaxe plus ancienne où les actions ne sont pas précédées par le caractère `-`, introducteur d'options : `tar cvf archive.tar rep`.

22. Sans l'option `f`, l'archive est stockée sur une bande magnétique. Si l'archive est écrite sur la sortie standard, ou lue sur l'entrée standard, on spécifie `-f -` ; un exemple classique est la duplication d'une branche de l'arbre UNIX avec une commande `tar c` et une commande `tar x` reliées par un tube (*cf.* 10.3, p. 73) dans lequel est redirigée l'archive, qui n'apparaît donc plus en tant que fichier :
`cd rep_source; tar -cf - . | (cd rep_cible; tar -xf - )`
Noter que la seconde partie de la commande est lancée dans un sous-shell (*cf.* 10.5.4, p. 77) dans lequel s'effectue le changement temporaire de répertoire de travail.

23. Il existe aussi une option `j` pour compresser/décompresser avec `bzip2`/`bunzip2`, ainsi que des options `Y` et `J` pour effectuer ces opérations avec respectivement `unlzma` ou `xz`.

archive dans le fichier `archive.tar` (du répertoire `/tmp/`), toute l'arborescence de l'utilisateur `user1` en partant du répertoire père de son répertoire d'accueil

liste : `tar -tf /tmp/archive.tar`
affiche la liste des fichiers archivés dans `archive.tar`

extraction : `tar -xvf /tmp/archive.tar`
restaure l'archive dans le répertoire courant (en créant donc un répertoire `user1` sous le répertoire courant).

Autres options de `tar` :
`-X` *`motif`* ou `--exclude=`*`motif`* permet d'exclure de l'opération les fichiers dont les noms correspondent au motif spécifié ; si ce motif comporte des caractères spéciaux, il est nécessaire de les protéger d'une interprétation par le shell.
`-d` compare les fichiers archivés avec ceux du système de fichiers et signale les différences.
`-w` demande une confirmation pour les opérations potentiellement destructives comme le remplacement d'un fichier.

Remarque importante : Sauf si on est administrateur de la machine, il faut éviter de sauvegarder une hiérarchie définie par son chemin absolu [24], sinon elle ne pourra pas être restaurée dans un autre répertoire. D'ailleurs, seul l'administrateur pourra la restaurer. ⇐⚠

3.2.3 Copier des fichiers avec la commande `rsync`

La commande `rsync` constitue un outil très puissant de copie de fichiers et de hiérarchies à travers le réseau : très efficace, en particulier pour les mises à jour pour lesquelles seules les différences sont transférées, éventuellement compressées, elle permet notamment la synchronisation de répertoires distants et la mise à jour de miroirs. Les nombreuses options disponibles concernant notamment les droits, les dates et les règles de sélection des fichiers en font un outil précieux pour les administrateurs.
Elle permet de faire des copies sur la machine locale, de la machine locale vers une machine distante ou en sens inverse.

`rsync [options]` *`source [user@host:]dest`*
`rsync [options]` *`[user@host:]source dest`*

Parmi les options les plus usitées :

- `-v` (**v**erbose) prolixe
- `-n` affiche les transferts qui seraient effectués sans les déclencher
- `-r` (**r**ecursive) récursif : recopie de toute la branche de l'arborescence
- `-t` (**t**ime) préserve les dates
- `-p` (**p**ermissions) préserve les droits
- `-z` (**z**ip) compression pour le transfert
- `-l` (**l**inks) copie les liens symboliques en tant que liens symboliques
- `-u` (**u**pdate) mise à jour : ne copie pas les fichiers qui sont plus récents côté destinataire

`--exclude=`*`motif`* permet d'exclure les fichiers dont le nom correspond au motif indiqué

`--cvs-exclude` (ou `-C`) permet d'exclure les fichiers d'administration des systèmes de gestion de sources (comme `subversion`).

Exemple

```
rsync -rvtpu --exclude='*~' user@sappli1.dsi.upmc.fr:mni/unix/ ~/unix-mni
```

met à jour (`-u`) récursivement (`-r`) le répertoire `~/unix-mni` de la machine locale à partir du répertoire `~user/mni/unix/` du serveur `sappli1`, en préservant aussi les droits (`-p`)

24. Certaines versions de `tar` avertissent l'utilisateur qui tente d'archiver une branche définie par son chemin absolu et enlèvent le `/` initial, ce qui permet une éventuelle restauration de la branche sans droit administrateur ; mais l'utilisation d'un chemin absolu reste déconseillée, car la restauration va alors créer toute une hiérarchie de répertoires inutiles.

et les dates (`-t`), mais sans transférer (`--exclude`) les fichiers de suffixe `~` (sauvegardes de la version précédente faites par l'éditeur `vi`) et affiche (`-v`) les opérations effectuées.

♠ 3.2.4 Manipuler des chemins avec `dirname` et `basename`

Les commandes `dirname` et `basename` analysent des noms de fichiers [25] avec leur chemin d'accès et en extraient les composantes, en coupant au niveau du dernier séparateur `/`

- `dirname` affiche la partie correspondant au nom du répertoire d'attachement donc jusqu'au dernier `/`
- `basename` affiche la partie correspondant au nom du fichier seul [26] donc après le dernier `/`

Ces commandes sont employées en particulier dans des procédures automatiques de manipulation de fichiers écrites en shell (*cf.* chap. 12).

Exemples :

`dirname /home/p6pfal/p6m1mni/toto`	affichera `/home/p6pfal/p6m1mni`
`dirname toto`	affichera « `.` » (répertoire courant)
`basename /home/p6pfal/p6m1mni/toto`	affichera `toto`
`basename toto`	affichera aussi `toto`

La commande `basename` accepte en deuxième argument une chaîne de caractères dont le nom sera amputé du côté droit avant affichage. Cette chaîne est très souvent (mais pas nécessairement) le suffixe du nom du fichier.
Exemples :

`basename /home/p6pfal/p6m1mni/toto.c .c`	affichera `toto`
`basename toto.f90 .f90`	affichera aussi `toto`
`basename  data ta`	affichera `da`

♠ 3.2.5 Découper des fichiers avec `split` et `csplit`

Un fichier texte peut être découpé (sur des lignes entières) en plusieurs fichiers, en fonction soit du nombre de lignes avec `split`, soit du contexte avec `csplit`.

`split` découpage à nombre de lignes fixées (1000 par défaut), sauf éventuellement le dernier fichier plus petit :

`split [-l` ***nb_de_lignes*** `]` ***fichier***

`csplit` coupure lors de l'occurrence d'un motif défini par une expression rationnelle en parcourant les lignes du fichier d'entrée

`csplit` ***fichier*** `'/`***exp***`/`***offset***`' '{`***n***`}'`

où ***exp*** est une expression rationnelle (*cf.* chap. 6) qui définit la ligne de la coupure (au décalage optionnel ***offset*** près) et ***n*** le nombre de fois où la recherche est répétée (`*` signifiant autant de fois que nécessaire). Si ***n*** est différent de `*`, le nombre maximum de coupures est ***n+1*** et le nombre de fichiers de sortie est au maximum ***n+2***.

Les noms des fichiers résultants sont définis automatiquement avec comme préfixe par défaut `x` pour `split` ou `xx` pour `csplit` et un suffixe pris dans la série `aa`, `ab`, `ac`, ..., `ba`, `bb` ... pour `split` et dans la série `00`, `01`, `02`, ..., `10`, `11` ... pour `csplit`.

Exemples

Par exemple, pour découper une base de données bibliographique `base.bib` au format BibTeX, avec seulement des références de livres introduites par le début de ligne `@book{`, en un fichier par livre, on exécutera (l'option `-z` évite la création d'un fichier vide si le

25. Ces commandes analysent les chemins de fichiers indépendamment de l'existence de ces fichiers : elles servent souvent à constituer leur nom avant de les créer.
26. Le fichier en question peut être un répertoire, auquel cas l'affichage de son nom par `ls -F` ajoute un « `/` » terminal de « décoration » qui ne doit pas être pris pour un séparateur.

motif est trouvé dès la première ligne) :
```
csplit -z base.bib '/^@book{/' '{*}'
```
La commande `csplit` permet de séparer un fichier source en C ou fortran comportant un programme principal et l'ensemble de procédures qu'il appelle en autant de fichiers que de procédures.
Il est possible de spécifier la partie suffixe des noms des fichiers produits avec un format de conversion de l'entier qui les numérote habituellement grâce à l'option `-b`. Par exemple, pour découper le fichier source fortran unique `complet.f90` en un fichier par module et un fichier pour le programme principal, et nommer les fichiers `xx0.f90`, `xx1.f90`, etc, on pourra utiliser la commande [27] :
```
csplit -b '%d.f90' complet.f90 '/end  *module/+1' '{*}'
```
Le décalage `+1` fait que la coupure intervient après la ligne de fin de module. Le dernier fichier produit est le programme principal, obligatoirement placé après le dernier module.

27. Le motif utilisé pour rechercher les fins de modules, ici en minuscules, peut être généralisé pour accepter aussi les majuscules selon la syntaxe des expressions régulières (*cf.* 6.3.2, p. 53) :
`[eE][nN][dD]` au lieu de `end`

Chapitre 4

Édition, visualisation, impression

4.1 Les fichiers texte et leurs codes

4.1.1 Fichiers texte et fichiers binaires

Tous les fichiers sont avant tout des fichiers binaires dont certains peuvent représenter du texte. La commande `file` *`fichier`* permet d'avoir une idée du type du fichier considéré : elle reconnait entre autres les fichiers sources écrits dans les principaux langages, les fichiers binaires exécutables sur certains processeurs, les fichiers au format `pdf` (**p**ortable **d**ocument **f**ormat), les fichiers d'archives (de suffixe `.tar`), certains fichiers (texte ou binaires) d'images...
Bien que, sous UNIX, les noms de fichiers ne comportent pas obligatoirement d'extension (ou suffixe), et puissent contenir plusieurs caractères « `.` » comme `linux.tar.gz`, il est d'usage (et impératif pour utiliser certaines commandes) d'associer une extension à chaque type de fichier. Par exemple, les fichiers source C et fortran90 porteront respectivement le suffixe `.c` et `.f90`, les fichiers binaires objets résultant de la compilation de ces sources porteront l'extension `.o` et l'exécutable issu de l'édition de lien s'appellera par défaut `a.out`. Les types de fichiers sont très nombreux et on pourra s'en donner une idée en consultant par exemple le site `http://www.file-extensions.org/` et, pour les plus courantes la page d'url : `http://www.file-extensions.org/extensions/common`

Le tableau situé en annexe (p. 118), présente quelques suffixes usuels de fichiers, les types associés, leur signification et le nom d'un ou plusieurs logiciels permettant de les produire, de les lire ou de les transformer.
La commande `od` (octal dump) permet d'afficher sous diverses formes le contenu binaire d'un fichier. Elle permet entre autres d'afficher un aperçu du contenu de fichiers binaires grâce à une ou plusieurs options `-t` qui spécifient comment interpréter les octets (nombre d'octets à grouper et conversion suivant un type), par exemple :

- `od -t o2` pour des entiers en octal sur 2 octets ;
- `od -t c` pour des caractères ASCII ou des séquences d'échappement (sur 1 octet)
- `od -t d4` pour des entiers sur 4 octets (32 bits) ;
- `od -t f4` ou `od -t fF` pour des flottants sur 4 octets (32 bits).

Enfin, la commande `strings` permet d'extraire d'un fichier binaire les parties de texte affichable. Par exemple, si un programme C ou fortran comporte l'écriture d'un message, l'exécutable (`a.out`) produit par compilation du fichier source inclut le texte du message qu'il doit afficher. Ainsi la commande `strings a.out` affichera, parmi beaucoup de texte, le message spécifié dans le fichier source.

4.1.2 Codage des fichiers texte

Les fichiers dits texte sont eux-mêmes constitués de séquences binaires représentant les caractères selon un certain codage. Seuls les (128) caractères codés sur 7 bits ASCII (American Standard Code for Information Interchange) sont reconnus de façon identique sur la plupart des systèmes. Mais de nombreux codages sur 8 bits ont permis d'étendre à 256 caractères le code ASCII afin de représenter les caractères dits nationaux, notamment les caractères accentués. Ils se distinguent suivant les groupes de langues qu'ils permettent de représenter et les systèmes d'exploitation :

— La norme iso-8859 est déclinée en plusieurs variantes régionales dont le code iso-8859-1, ou iso-latin1 pour l'Europe occidentale utilisé sous UNIX [1].
— Les systèmes propriétaires utilisent parfois d'autres codages comme IBM ou Windows avec les codes CPxxx comme CP850 ou CP1252 (très proche du code iso-8859 [2], voir `man iso-8859-15`), MacRoman pour MacIntosh...

4.1.3 Le codage unicode

Enfin, un standard de représentation des caractères de l'ensemble des langages du monde, Unicode [3], aussi connu sous la norme iso-10646 a été défini et est en cours d'implémentation dans les systèmes et les logiciels. Il permet de représenter environ un million de caractères, numérotés en binaire de 0 à $2^{20} - 1$: ils ne peuvent donc être tous codés sur 2 octets et trois codages ont été associés au standard Unicode :

— UTF-32 où tous les caractères sont codés sur 4 octets ;
— UTF-16 où les caractères les plus courants sont codés sur 2 octets et les autres sur deux 16-uplets grâce à des codes d'indirection ;
— UTF-8 où les caractères les plus courants sont codés sur 1 octet et les autres sur un 2, 3 ou 4 octets. C'est sous ce codage qu'Unicode commence à être déployé. Si UTF-8 et iso-8859-1 coïncident dans le codage des caractères de l'ASCII, il n'en est pas ainsi pour les caractères pourvus de signes diacritiques du latin-1, codés en général sur deux octets en UTF-8, ce qui oblige à *distinguer octet et caractère*, autrefois confondus. Dans un environnement UTF-8, la commande `wc -cm` (*cf.* 2.3.9, p. 11 et 15.4.3, p.108) permet de détecter les caractères sur plus d'un octet mais aussi les combinaisons invalides issues de l'iso-8859-1. ⇐⚠

  En UTF-8, les premiers bits d'un octet permettent de déterminer son rôle dans la représentation des caractères :
  — s'il commence par `0`, c'est un code ascii pur sur un seul octet ;
  — s'il commence par `10`, c'est un octet de suite ;
  — s'il commence par `110`, c'est le premier octet d'un caractère sur 2 octets ;
  — s'il commence par `1110`, c'est le premier octet d'un caractère sur 3 octets ;
  — s'il commence par `1111`, c'est le premier octet d'un caractère sur 4 octets.

Pour plus de précisions, on pourra consulter le manuel de Desgraupes (2005).

4.1.4 Outils de transcodage des fichiers texte

Deux commandes, `recode` [4] et `iconv` [5], permettent de convertir des fichiers texte d'un code à l'autre. Par exemple, la commande suivante [6] convertit le fichier *`fic-iso.txt`* codé en iso-8859-1 vers le fichier *`fic-utf8.txt`* codé en UTF-8 :

```
recode 'ISO-8859-1..UTF-8' < fic-iso.txt > fic-utf8.txt
```

La même conversion peut être accomplie par `iconv` :

```
iconv -f ISO-8859-1 -t UTF-8 < fic-iso.txt > fic-utf8.txt
```

L'option `-l` (list), commune à ces deux commandes, permet d'afficher la liste des codages disponibles. Noter enfin que seul `recode` peut effectuer des transcodages irréversibles à ⇐♥

1. Le codage iso-8859-1 ne comporte pas le caractère Ÿ, ni les ligatures œ et Œ, pourtant nécessaires en français, ni le symbole de l'euro. On lui préfère la version plus récente iso-8859-15 (voir `man iso-8859-15`) ou iso-latin9 qui inclut ces caractères (œ à la place de 1/2 et Œ à la place de 1/4 par exemple).
2. Dans le code CP1252, les 32 caractères de contrôle de 128 à 159 de l'iso-8859-15 ont été remplacés notamment par le caractère Ÿ, les ligatures œ et Œ, le symbole de l'euro, ... La position de ces caractères n'est donc pas la même que dans le code iso-8859-15. Des apostrophes typographiques ont aussi été ajoutées, qui sont parfois mal reconnues des systèmes UNIX et traduites par des points d'interrogation.
3. *cf.* http://www.unicode.org/, http://hapax.qc.ca/
4. *cf.* http://recode.progiciels-bpi.ca/
5. *cf.* http://www.gnu.org/software/libiconv/
6. Prendre garde que `recode` travaille par défaut en place sur le fichier fourni en argument de la commande. Si on souhaite conserver le fichier d'origine dans le codage initial, il est donc prudent d'« alimenter » `recode` par redirection.

condition de préciser l'option `-f` (force). C'est le cas lorsque l'on souhaite supprimer accents et signes diacritiques pour obtenir de l'ascii « plat » :
`recode -f latin1..flat < fic-iso.txt > fic-ascii.txt`

Les fichiers texte peuvent être affichés par des commandes du type `cat` ou `more` (`less` sous linux), commandes de visualisation de fichiers vues précédemment. Un éditeur de texte permet de créer ou modifier un fichier texte, mais on verra que les filtres UNIX permettent aussi des manipulations sur les fichiers texte. Noter enfin que les éditeurs de texte modernes comme `vim`[7] et `emacs` peuvent prendre en charge le transcodage au vol[8] des fichiers textes.

4.2 Édition de fichiers texte

Sous UNIX, on distingue plusieurs types d'éditeurs de texte selon le mode d'interaction avec l'utilisateur :

- **ligne** : les éditeurs **`ex`** et **`ed`** de base d'UNIX, puissants et très robustes (utilisables avec un terminal quelconque), mais d'interface aujourd'hui un peu désuète ;
- **pleine page** : ces éditeurs doivent connaître le type de terminal utilisé, déclaré par la variable d'environnement `TERM` et ses capacités de gestion du curseur texte, ... informations stockées sous la forme d'une base de données de terminaux.
  - **`vi`** qui est une sur-couche de **`ex`** : très puissant, présent sur tous les UNIX, c'est aussi le seul disponible lors de l'installation du système.
    Une version améliorée, **`vim`** (*cf.* http://www.vim.org/), est diffusée comme logiciel libre notamment sous linux : doté d'une interface plus agréable (en particulier les commandes **`ex`**, passées sous l'invite **`:`** bénéficient d'un historique et sont éditables de façon similaire à la ligne de commande du shell), c'est un éditeur sensible au langage (C, fortran, LaTeX, ...) qui permet la mise en valeur de la syntaxe par des couleurs, la complétion des mots-clefs, ... Enfin, il permet d'éditer des textes dans différents codages de caractères et en particulier unicode (*cf.* 4.1.3, p. 31).
  - **`emacs`** (*cf.* http://www.gnu.org/software/emacs/emacs.html) est encore plus puissant, et permet aussi d'accomplir d'autres tâches (courrier, navigation, ...) mais s'avère plus gourmand en ressources.
- **environnement graphique multi-fenêtres** avec menus, gestion de la souris, ...

  Dans cette catégorie figurent à la fois des éditeurs élémentaires très simples d'emploi, mais aussi des outils très puissants comme **`gvim`** (version graphique de `vim`), **`xemacs`** ou **`gedit`**[9]. Ces derniers nécessitent alors des ressources importantes, qui ne les rendent pas toujours praticables à distance.

4.2.1 Les modes des éditeurs

Un éditeur présente deux modes principaux de fonctionnement :

- le mode **commande**[10] où les caractères saisis sont **interprétés** comme des ordres (requêtes), donc immédiatement exécutés ;
- le mode **insertion** où les caractères saisis sont directement **insérés** dans le fichier.

Le mode **par défaut** est :

⚠⇒ — le mode **commande** sous **`vi`**, ce qui s'avère très déroutant au premier abord. Le

7. La manipulation des caractères codés sur plusieurs octets nécessite une version de `vi` compilée avec l'option `+multi_byte`, ce que l'on peut vérifier en lançant `vi --version`.
8. Dans le cas d'un transcodage au vol, l'éditeur `vi` affiche alors l'avertissement `converti` sur la ligne d'état.
9. On préférera `gedit` à `nedit` qui ne gère pas le codage UTF-8.
10. Ne pas confondre les commandes internes de l'éditeur plus précisément appelées requêtes et les commandes au sens d'UNIX interprétées par le shell.

passage en mode insertion doit se faire par une requête (*cf.* figure 4.1, p. 33).

— le mode **insertion** sous `emacs`. Les requêtes doivent alors être introduites par des caractères de contrôle : `Ctrl`, `Échap`

4.3 L'éditeur `vi`

Sous `vi`, un certain nombre de requêtes permettent de passer du mode commande au mode insertion :

— `a` et `A` (**a**ppend), ajout après le curseur ou en fin de ligne ;

— `i` et `I` (**i**nsert), insertion avant le curseur ou en début de ligne ;

— `o` et `O` (**o**pen), ouverture d'une ligne après ou avant la ligne courante.

Il n'existe qu'une méthode pour passer du mode insertion au mode commande : appuyer sur la touche `Échap` (Escape).

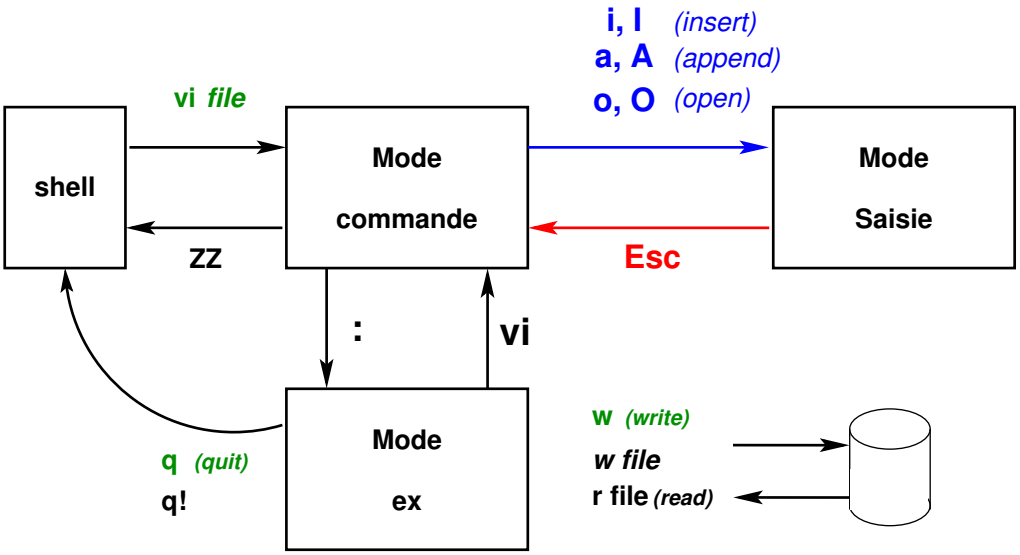


FIGURE 4.1 – Modes de fonctionnement de `vi`

Un troisième mode, le mode **dialogue** est accessible de façon temporaire depuis le mode commande de `vi` pour :

— passer des requêtes `ex` via la requête « `:` »

— rechercher de motifs dans le texte (via les requêtes « `/` » ou « `?` »)

Ces requêtes sont activées par la touche `Entrée`, et immédiatement après leur exécution, l'éditeur revient en mode commande. On peut enfin considérer un quatrième mode dit de remplacement (overwrite), initié notamment par les requêtes `R` ou `c`, où les caractères frappés viennent se substituer au texte déjà présent. La sortie de ce mode s'effectue via la touche d'échappement.

4.3.1 Principales requêtes de l'éditeur `vi`

Déplacements

Les déplacements se font en mode commande [11] en principe.

11. Si, sur les versions modernes de `vi`, il est possible de déplacer le curseur avec les flèches en mode insertion, il faut être conscient qu'en fait ce déplacement est interprété par une commutation du mode : sortie du mode insertion, puis déplacement et enfin retour au mode insertion comme avec la requête « `a` ». Si donc, on réitère la dernière commande ayant fait appel à ce mécanisme par la requête « `.` », on ne fera qu'ajouter le texte inséré après le déplacement. De même, l'annulation par la requête « `u` » ne portera que sur cette dernière phase d'insertion postérieure au déplacement.

← ↑ ↓ → les quatre flèches
`0` première colonne, `$` fin de ligne
par mots : `w` (**w**ord, début de mot vers l'avant), `e` (**e**nd, fin du mot suivant), `b` (**b**ackwards, début de mot vers l'arrière)
par écrans : `^F` (**f**orward, vers le bas), `^B` (**b**ackward, vers le haut)
en spécifiant le numéro de ligne *n* : *n*`G` (**G**o)

Modifications

`r c` (**r**eplace) remplace[12] **le** caractère pointé par le curseur par **le** caractère `c`
`cw` (**c**hange **w**ord) change un mot
`~` passe de majuscule à minuscule et réciproquement

Destructions

`x` détruit le caractère pointé par le curseur
`dw` (**d**elete **w**ord) détruit le mot à droite du curseur
`dd` détruit la ligne courante

Restitution de la mémoire tampon

`p` (**p**aste) restitue le contenu du tampon après le curseur ; `P` le restitue avant
Applications
`xp` permet de permuter deux caractères... en cas de dyslexie au clavier !
`ddp` permet d'échanger deux lignes (la courante et la suivante)

Divers

`.` réitère la commande précédente
`u` (**u**ndo) annule la précédente commande
`%` bascule le curseur d'un délimiteur à l'autre dans les paires comme `()[]{}`
`J` (**j**oin) concatène deux lignes (la ligne suivante à la ligne courante)
`^L` rafraîchit l'affichage

Recherche de chaînes de caractères

`/`*motif* recherche *motif* vers l'avant
`?`*motif* recherche *motif* vers l'arrière
`n` (**n**ext) recherche l'occurrence suivante dans le même sens
`N` (**N**ext) recherche l'occurrence suivante dans le sens inverse

Complétion automatique des mots

♥ ⇒ La complétion automatique des mots, disponible avec `vim`, permet de simplifier la saisie des textes. En mode insertion, après avoir écrit le début du mot, saisir `^P` (Previous) ou `^N` (Next)[13] pour rechercher un mot commençant ainsi et déjà présent dans le fichier en édition. Si plusieurs mots commencent par ce motif, `vim` affiche une mini-liste dans laquelle on peut se déplacer avec les flèches.

4.3.2 Requêtes `ex`

L'éditeur `vi` n'est qu'une sur-couche pleine page de l'éditeur ligne `ex`, dont il peut exploiter toutes les requêtes, à condition de passer dans le mode `ex` grâce au caractère « `:` » (*cf.* Fig. 4.1, p. 33).

12. Bien noter que dans cette requête il ne faut pas utiliser la touche d'échappement pour signifier la fin de la saisie.
13. La recherche rebouclant de la fin de fichier vers le début, choisir `^P` ou `^N` change seulement l'ordre dans lequel s'effectue la recherche.

Requêtes ex élémentaires

`:q` (**q**uit) sort de `vi`
`:q!` force une sortie sans sauvegarde
`:w` (**w**rite) sauvegarde dans le fichier courant
`:w fic` sauvegarde dans le fichier `fic`
`:r fic` (**r**ead) ajoute le contenu du fichier `fic` après la ligne courante

Adressage des commandes ex

La portée par défaut d'une commande `ex` est la ligne courante, sauf si la requête est précédée d'une adresse (numérique ou paramétrée).

`:`*`n1`*`,`*`n2`*	de la ligne *`n1`* à la ligne *`n2`*
`:.,`*`n`*	de la ligne courante (« `.` ») à la ligne *`n`*
`:`*`n`*`,$`	de la ligne *`n`* à la dernière ligne
`:`*`n1`*`,+`*`n2`*	`+` adressage relatif
`:`*`n1`*`,-`*`n2`*	`-` adressage relatif
`:%` ou `:1,$`	l'ensemble des lignes
`:/`*`motif1`*`/,/`*`motif2`*`/`	de la première ligne contenant *`motif1`* à la première ligne contenant *`motif2`* en partant de la ligne qui **suit** la ligne courante

Autres commandes ex

`d`	(**d**elete) détruit des lignes
`co` (ou `t`)	(**co**py) copie des lignes
`m`	(**m**ove) déplace des lignes
`s/`*`ch1`*`/`*`ch2`*`/`	(**s**ubstitute) substitue une chaîne à une autre
`g`	(**g**lobal search) recherche un motif

Exemples de commandes ex

`:r toto`	ajoute le fichier `toto` après la ligne courante
`:.,+10w toto`	écrit les 11 lignes partant de la ligne courante dans le fichier `toto`
`:.,$d`	détruit depuis la ligne courante jusqu'à la fin du fichier
`:1,5co32`	copie les lignes 1 à 5 après la ligne 32
`:1,3m$`	déplace les trois premières lignes à la fin de fichier
`:g/motif/d`	recherche toutes les occurrences de `motif` dans le fichier et détruit les lignes où il est présent

Substitutions élémentaires

`:s/motif1/motif2/`	le premier `motif1` éventuel sur la ligne courante est remplacé par `motif2`
`:s/motif1/motif2/g` (**g**lobal)	tous les `motif1` éventuels de la ligne courante sont changés en `motif2`
`:s/motif1/motif2/gc` (**c**onfirm)	*idem* mais une confirmation est demandée avant substitution

Expressions régulières

Pour construire des motifs génériques, on peut utiliser des caractères spéciaux en suivant la syntaxe des « expressions régulières » (voir chapitre 6). Dans les seconds membres des substitutions, le motif effectivement trouvé peut être référencé par `&`. Par exemple :

`:s/[-+=]/ & /g`	entoure les symboles `-`, `+` et `=` par des espaces, `&` représente le motif effectivement trouvé

4.3.3 Configuration de `vi`

Les nombreuses options de `vi`/`ex` sont accessibles par `:set`

`:set`	affiche l'état des principales options
`:set all`	affiche l'état de toutes les options (très nombreuses pour `vim`)
`:set nu` (`:set number`)	affiche les numéros des lignes
`:set nonu`	n'affiche plus les numéros des lignes
`:set list`	affiche `^I` pour les tabulations et marque les fins de ligne par `$`
`:set nolist`	affiche normalement les tabulations et ne marque pas les fins de ligne
`:set ic`	ignore la casse (majuscule/minuscule) dans les recherches
`:set encoding`	affiche le codage employé
`:set encoding=utf-8`	choisit le codage UTF-8 (*cf.* 4.1.3, p. 31)
`:set encoding=latin1`	choisit le codage ISO-8859-1
`:language`	affiche les variables de contrôle local (*cf.* 15.4.1, p. 106)

On peut stocker les choix d'options personnels dans le fichier de configuration de `ex` : `~/.exrc` (les options particulières de `vim` peuvent être définies dans le fichier `~/.vimrc`).

L'aide-mémoire en annexe page 120 présente un résumé des commandes de `vi`.
On pourra consulter la FAQ http://vimdoc.sourceforge.net/htmldoc/vimfaq.html et une URL enthousiaste pour l'éditeur `vi` : http://thomer.com/vi/vi.html

♥ ⇒ **Correction orthographique** `vim` propose des outils de correction orthographique s'appuyant sur des dictionnaires installés dans le répertoire `/usr/share/vim/spell/` [14] par défaut. À condition d'avoir installé le dictionnaire français dans le codage du fichier en édition, la commande `:setlocal spell spelllang=fr` par exemple active la vérification orthographique pour le français. Les mots inconnus sont alors affichés sur un fond coloré et si on saisit `z=` avec le curseur sur le mot, des suggestions sont proposées. Des dictionnaires pour les principales langues (en codage latin1 ou utf-8) sont disponibles sur le site http://ftp.vim.org/vim/runtime/spell/ [15].
Pour plus de détails, consulter par exemple le paragraphe `Spell checking` dans la documentation https://wiki.archlinux.org/index.php/vim.

♥ ⇒ **Comparaison et fusion de fichiers texte** On notera de plus la commande `vimdiff` qui permet, comme `diff`, de comparer deux fichiers texte, mais aussi de les éditer en parallèle. Le terminal est séparé verticalement en deux « fenêtres », et chaque fichier est disponible en édition dans une fenêtre. Les lignes communes sont masquées et les différences colorisées ; on change de fenêtre active en édition par la combinaison de touches `^W w` ; la sortie d'édition des 2 fichiers se fait par `:qa`
Noter enfin que `vim` permet de découper l'écran selon la verticale `^W V` ou l'horizontale `^W N` (comme avec `vimdiff`), pour éditer plusieurs fichiers dans un même terminal [16].

4.4 Les éditeurs `emacs` et `xemacs` [17]

`emacs` est l'un des éditeurs de textes les plus courants sous UNIX, après `vi`. C'est également l'un des plus puissants et l'un des plus complets, ce qui n'a pas que des avantages : ce logiciel est plus gourmand en ressources que `vi`, ce qui peut le rendre malcommode à utiliser sur des machines peu puissantes ou partagées entre un grand nombre d'utilisateurs.

14. Si nécessaire, on peut compléter à titre personnel par des dictionnaires placés sous le répertoire `~/.vim/spell/`
15. Par exemple, http://ftp.vim.org/vim/runtime/spell/fr.utf-8.spl pour le dictionnaire français au codage utf-8.
16. Cette possibilité existe aussi sous `emacs`, (*cf.* 4.4.5, p. 39).
17. Section rédigée par Frédéric Meynadier (Frederic.Meynadier@obspm.fr)

Son utilisation « basique » est plus simple que celle de `vi`, notamment parce qu'il n'y a pas de séparation entre l'édition et la saisie, et que les versions modernes disposent de menus déroulants comparables à ceux que l'on trouve dans les éditeurs de textes à la mode Windows. En parallèle, de nombreuses commandes et de nombreux raccourcis claviers en font un outil de choix pour l'édition de textes, qu'il s'agisse de codes de programmation ou de simples fichiers texte. On pourra notamment consulter Cameron et Rosenblatt (1997) en français, ou Cameron *et al.* (2004) plus récent en anglais pour explorer plus en profondeur les possibilités d'`emacs`.
`emacs` et `xemacs` sont deux « branches » d'un même logiciel : cette différence de dénomination correspond à des divergences d'opinions entre développeurs il y a quelques années, mais les deux logiciels sont essentiellement identiques et, sauf bug, les commandes utilisées dans l'un marchent aussi dans l'autre. La principale différence visible est la présence d'une barre d'outils cliquable sous la barre des menus de `xemacs`. C'est pourquoi les paragraphes suivants ne feront pas la distinction entre les deux.

4.4.1 Fonctions de base

Le démarrage du logiciel se fait simplement en tapant
`emacs monfichier &`
Une fenêtre s'ouvre alors, avec le fichier en question prêt à être édité. Si `emacs` est lancé sans argument, la page qui s'ouvre est un petit mode d'emploi qui s'efface lorsqu'on presse une touche, pour être remplacé par trois lignes d'avertissements (qui invitent à ouvrir ou créer un fichier avant d'écrire : menu `File/Open File`...).
Signalons qu'il est possible de demander à `emacs` de ne pas ouvrir de fenêtre supplémentaire [18], mais de s'exécuter dans le terminal d'où on le lance : il suffit pour cela de taper ⇐♥ `emacs -nw` (**no** **w**indow).
La frappe du texte peut démarrer directement (il n'y a pas de « mode insertion » spécifique à enclencher). Définition : `emacs` appelle les textes en cours d'édition des Buffers.

Une fois l'édition terminée, il faut enregistrer les modifications. Cela peut se faire à la souris (Menu : `File/save` (`current buffer`) ou `File/save buffer as`... puis taper le nom du fichier dans la ligne du bas, selon le cas). On quitte via le menu `File/Exit Emacs`. En cas d'oubli d'enregistrement, `emacs` rappelle que le buffer n'est pas enregistré et propose de le faire.

4.4.2 Principe des raccourcis clavier

L'une des principales caractéristiques de `emacs` est d'offrir une multitude de raccourcis clavier très utiles, et qui peuvent devenir des réflexes si on les utilise souvent. Malheureusement, en raison du nombre de fonctions disponibles, ces raccourcis sont plus compliqués que ceux auxquels on peut être habitué et nécessitent souvent plus de 2 touches...
Par exemple : l'enregistrement du buffer en cours se fait en maintenant appuyée la touche `Control` puis en pressant les touches `x` et `s`. Ce raccourci est rappelé dans le menu déroulant sous la forme `C-x C-s`. Pour quitter, le raccourci est `C-x C-c`. En fait toutes ces touches sont proches, ce qui fait que l'action « enregistrer puis quitter », extrêmement courante, se fait très vite et devient rapidement un réflexe. Certains raccourcis réclament une touche `M`, qui correspond à la touche `Meta` (qui existe sur les claviers des stations Sun, par exemple). Sur les PC, cette touche est généralement remplacée par le `Alt` situé à gauche du clavier [19]. Dans les paragraphes qui suivent, les raccourcis claviers seront indiqués autant que possible en complément des menus concernés.
L'aide-mémoire en annexe page 122 présente un résumé des commandes de raccourcis clavier de `emacs`.

18. Cette méthode est précieuse quand on travaille à distance sans serveur de fenêtres graphiques, ou via un réseau à débit insuffisant.

19. Humour d'informaticien : cette apparente complication a poussé les informaticiens à prétendre que emacs signifie `Esc-Meta-Alt-Control-Space` et qu'on n'arrive à rien sans appuyer simultanément sur toutes ces touches.

4.4.3 Fonctions d'édition de texte

Les fonctions de copier-coller à la souris sont disponibles : il est possible de surligner le texte à copier en glissant la souris avec le bouton gauche enfoncé, puis de coller à l'aide d'un clic du bouton du milieu (ou de la molette). Selon les configurations, le collage aura lieu à partir du curseur ou à partir de l'endroit où se trouvait la souris au moment du clic — faites l'expérience !
`emacs` introduit également des raccourcis d'éditions de ligne qui ont été, par la suite, introduits dans la plupart des shells (dont `bash`, *cf.* 2.2.2, p. 6). Citons `C-k`, qui efface toute une ligne et la copie en mémoire. Il est possible de stocker ainsi plusieurs lignes par appui répété, à condition de n'avoir pas déplacé le curseur entre deux appuis. Le collage se fait alors par `C-y`. Si pour une raison ou pour une autre les touches spéciales (retour au début de ligne, end, etc...) ne sont pas correctement configurées et ne marchent pas, les raccourcis suivants peuvent être utiles : `C-b` revient en arrière (**back**) d'un caractère, `C-e` va à la fin (**end**) de la ligne, `C-a` retourne au début de la ligne (le « b » de beginning était déjà pris...).
Dans tous les cas, si vous ne savez plus quoi faire et que la ligne du bas vous pose des questions auxquelles vous ne savez pas répondre, vous pouvez taper `C-g` qui annule toutes les commandes en cours et vous ramène à l'édition de texte.
Les fonctions de recherche et de remplacement automatique sont bien évidemment disponibles. `C-s` propose une recherche « incrémentale » : cela signifie que `emacs` va chercher tous les mots commençant par ce que vous tapez, au fur et à mesure que vous le tapez. Pour passer d'une occurrence à l'autre, tapez à nouveau `C-s`. Le traditionnel search and replace est accessible par le menu `Edit/Search/Replace`... (raccourci : `M-%`, soit pour la plupart des PCs : `Alt`(gauche)`-shift-%`). Il faut d'abord taper le mot que l'on souhaite remplacer, puis ce par quoi on souhaite le remplacer. La recherche ne se fait que sur le texte situé après le curseur. À chaque occurrence, `emacs` s'arrête et propose d'effectuer le remplacement. `y` ou `espace` signifie oui, `n` signifie non, `!` signifie « oui pour tous » et `q` signifie terminer.
Autres fonctions utiles pour la lisibilité du texte : s'il n'est pas activé par défaut, le retour à la ligne automatique se fait en tapant `M-x auto-f<tab>`, le `tab` complétant le nom de commande à `auto-fill-mode`. Ponctuellement, pour reformater un paragraphe, on peut utiliser `M-q`.
En cas d'erreur ou de manœuvre intempestive, il est toujours possible de revenir en arrière grâce à `dit/Undo`(ou `C-_`).

4.4.4 Aides à l'édition

`emacs` est très adaptable aux différents types de tâche qu'on lui demande. En devinant le type de texte que vous allez taper, grâce à l'extension (`*.c`, `*.f90`, `*.html`, `*.tex` par exemple), il se place automatiquement dans un mode prédéfini. Souvent, cela se traduit par l'apparition d'un menu spécifique dans la barre des menus. Normalement, cela active automatiquement aussi une gestion spécifique de l'indentation (c'est à dire les retraits obtenus par la touche `tab`) ainsi que la coloration syntaxique : les éléments spécifiques du langage, par exemple `program` en fortran ou `int` en C, apparaissent d'une couleur différente, choisie de façon à faire ressortir la structure du programme.
Exemple d'utilisation de l'indentation : à l'intérieur d'un programme en C, une boucle `for` commence comme suit :

```
for (i = 0; i < N, i++) {
```

Lors du retour à la ligne, `emacs` va se placer, de lui même, en léger décalage. Quand on tape la ligne suivante, le résultat ressemble donc à ceci :

```
for (i = 0; i < N, i++) {
   tab[i] = i;
```

Lors du retour à la ligne suivant, emacs s'aligne sur le même retrait. À la dernière ligne de la boucle, le seul caractère est « `}` » qui ferme le bloc : dès l'appui sur entrée, emacs

replace l'accolade fermante au même niveau que `for`, et surligne brièvement le début et la fin du bloc :

```
for (i = 0; i < N, i++) {
   tab[i] = i;
}
```

Si par la suite on ajoute une ligne au bloc, avec une mauvaise indentation, il suffit de placer le curseur n'importe où dans la ligne puis de taper `tab`, ce qui restituera l'alignement correct.
Cette fonctionnalité aide aussi à repérer les erreurs de syntaxe avant la compilation : en oubliant par exemple un point virgule au bout d'une ligne, le début de la ligne suivante sera placé à une indentation manifestement incorrecte car s'attendant à une continuation de la ligne précédente. Il suffit alors de corriger l'erreur.
Notez néanmoins que l'indentation automatique est une indication et pas une obligation [20] : il est toujours possible de placer ses lignes où on le souhaite à coup de `suppr` et de `space`. Néanmoins le respect d'une indentation correcte est une aide précieuse pour rédiger des programmes lisibles.

4.4.5 Ouverture de plusieurs fichiers simultanément

`emacs` est capable d'ouvrir simultanément plusieurs fichiers, par le menu `File/Open File` ou le raccourci `C-x C-f`. Cette commande suppose de taper le nom du fichier. On dispose alors de la complétion automatique comme en shell : tapez les premières lettres puis `tab` pour compléter ce qui n'est pas ambigu. S'il y a ambiguïté, un second appui sur `tab` affiche une liste des fichiers possibles. Il est alors possible de choisir le fichier concerné à la souris par un clic du bouton du milieu (ou de la roulette).
Cette option est de loin préférable à l'ouverture d'une seconde fenêtre `emacs`, par exemple en tapant `emacs fichier2` dans un autre shell. D'abord pour des raisons de performance : `emacs`, comme on l'a dit, est un logiciel relativement gourmand, il n'est donc pas souhaitable d'en avoir plusieurs qui tournent simultanément. Ensuite, emacs permet facilement de gérer plusieurs fichiers ouverts : le menu `Buffers` contient la liste des fichiers en cours d'édition. Une étoile signale les fichiers modifiés par rapport à la version enregistrée sur le disque dur. Le raccourci `C-x s` (pas de Control quand on appuie sur `s`, cette fois !) permet d'enregistrer tous les fichiers ouverts. Enfin, si on veut éditer deux fichiers simultanément, c'est tout à fait possible : la commande du menu `File/New Frame` permet d'ouvrir une seconde fenêtre ayant exactement la même liste de buffers que la première. La commande du menu `File/Split window` (`C-x 2`) permet de séparer la fenêtre en deux ; `Unsplit Window` (`C-x 1`) permet de revenir à une seule fenêtre.

4.4.6 Fonctions avancées

`emacs` est un logiciel extrêmement complet et particulièrement expansible. Les contributions de différents auteurs en font un logiciel très polyvalent [21].
Un inventaire complet de ces fonctionnalités serait impossible, aussi se limitera-t-on à quelques fonctions usuelles :

Sélections par rectangles

Cette fonctionnalité est particulièrement utile dans les fichiers de données, où les chiffres sont alignés selon des colonnes. On surligne à la souris une zone de texte telle que le début du surlignage corresponde au coin supérieur gauche de la colonne ou des colonnes que l'on souhaite sélectionner. La fin du surlignage doit correspondre au coin inférieur droit.

20. Sauf sous `python`, où l'indentation définit les blocs.
21. Humour d'informaticien : « `emacs` fait tout, même le café ». Pêle-mêle, emacs peut servir pour l'envoi et la réception de courrier électronique, les calculs de dates dans tous les calendriers (y compris les calendriers républicains et persans).

Le surlignage déborde du rectangle sur toutes les lignes intermédiaires, qui semblent intégralement sélectionnées.
Les commandes suivantes sont alors utilisables : `C-x r k` coupe le rectangle sélectionné, et `C-y` le colle. `C-x r r` copie le rectangle dans un registre, qu'il faut nommer après en frappant un caractère. Pour insérer ce registre ailleurs, la commande est `C-x r i` suivie du caractère choisi précédemment.
Cette procédure permet, par exemple, de changer l'ordre de colonnes, d'en supprimer, de compacter un tableau en supprimant une colonne de blancs inutiles... Attention néanmoins à ne pas tronquer les données involontairement.

Macros

Les macros servent à exécuter des opérations répétitives. La commande `C-x (` entame l'enregistrement de la macro (au clavier uniquement). Il faut ensuite taper la suite de touches exactement comme elles devront être exécutées. L'enregistrement est clôturé par `C-x )`. On peut alors exécuter la macro grâce à `C-x e`. Combinée à la commande de répétition (par exemple `C-u 50 ?` affiche 50 points d'interrogation à la suite les uns des autres), on peut exécuter 50 fois la même macro. La commande à taper est alors `C-u 50 C-x e`.
Exemple pratique : un fichier contient mille lignes, et on veut insérer la ligne « — » entre chacune d'entre elles. En se plaçant au début du fichier, la suite de commandes est donc :

```
C-x (
C-e <entrée> --- <flèche bas>
C-x )
C-u 998 C-x e
```

4.4.7 Internationalisation

L'éditeur `emacs` dans ses versions récentes dispose avec **Mu**lti**l**ingual **E**nvironment (mule) de nombreuses fonctionnalités facilitant le multi-linguisme. Le codage du fichier en édition détecté par `emacs` est indiqué sur la ligne d'information par un « `1` » pour `iso-8859-1` et un « `u` » pour `utf-8` avant le nom du fichier (*cf.* 15.4, p. 106). Il est possible de convertir le tampon d'édition dans un autre codage que l'on peut choisir de façon interactive grâce au mini-tampon...

4.4.8 URLs

Un aide mémoire : http://www.tuteurs.ens.fr/unix/editeurs/emacs_memo.html
Le site de gnu/emacs (en anglais) : http://www.gnu.org/software/emacs/emacs.html
Le wiki de gnu/emacs : http://www.emacswiki.org/cgi-bin/emacs-fr
Une fiche récapitulative des fonctions de base :
http://refcards.com/refcards/gnu-emacs/index.html

4.5 Impression

4.5.1 Gestion des impressions

Les commandes d'impression sont différentes suivant les systèmes UNIX et on distingue essentiellement les commandes BSD et System-V (comme `aix`) ; la commande `lp` relève de la norme POSIX.

fonction	BSD	System-V
impression	`lpr -P`*`imprim`* `[`*`fichier`*`]`	`lp -d`*`imprim`* `[`*`fichier`*`]`
état de la file	`lpq [-P`*`imprim`*`]`	`lpstat [-p`*`imprim`*`]`
annulation	`lprm -P`*`imprim num_requête`*	`cancel` *`num_requête imprim`*

Si on ne leur fournit pas de fichier d'entrée, les commandes d'impression `lp` et `lpr` impriment l'entrée standard : elles peuvent donc être alimentées par un tube (*cf.* 10.3, p. 73). À la demande d'impression par `lp` ou `lpr`, le système répond en attribuant un identificateur numérique à la requête, qui permet de la localiser dans la file d'attente et sera utilisé si on souhaite annuler cette requête avant l'impression.
Les imprimantes modernes sont généralement des imprimantes laser postscript, configurées de manière à imprimer des fichiers de type `postscript` (suffixes `.ps` en général, *cf.* p. 30) mais peuvent être aussi utilisées pour imprimer des fichiers texte.

4.5.2 Impression de fichiers non postscript

De façon assez générale, un fichier doit être converti en langage postscript pour pouvoir être imprimé. Les outils de conversion dépendent de la nature des fichiers. À titre d'exemple :

— Pour imprimer un fichier de format pdf (**p**ortable **d**ocument **f**ormat), il faut en général passer par un programme de conversion comme `pdf2ps` ou un programme d'affichage des `pdf` comme `acroread` ou `xpdf`, qui assure la conversion en postscript. Mais les imprimantes récentes sont souvent capables d'interpréter le format `pdf`, ce qui permet de lancer directement l'impression de fichiers `pdf` via `lpr`.

— Les fichiers `dvi`, produits par la compilations de sources LaTeX avec `latex`, doivent être convertis en postscript par l'utilitaire `dvips` pour être imprimés. Mais on peut aussi compiler les sources LaTeX avec `pdflatex` de façon à obtenir directement des fichiers `pdf`.

— Les fichiers d'images dans de très nombreux formats (`jpeg`, `png`, `pbm`, `ppm`...) doivent aussi être convertis en postscript ou pdf avant impression, par exemple avec l'outil `convert`.

Noter que les installations linux récentes fournissent des outils qui simplifient l'impression des fichiers grâce à des outils de conversion automatiques : en particulier la commande `a2ps` (**a**ll-to-**ps**) permet d'imprimer les fichiers texte[22] écrits par exemple en C ou en fortran en mettant en valeur la syntaxe de ces langages.

22. Attention, si `a2ps` gère correctement les caractères accentués français en codage iso-8859 sur un octet, il ne gère pas le codage UTF-8. Il faut donc recourir à un transcodage du texte via `iconv` ou `recode` (*cf.* 4.1.4, p. 31) pour le fournir à `a2ps`.

Chapitre 5

Introduction aux filtres

5.1 Notion de filtre

On appelle **filtre** une commande qui lit les données d'entrée saisies au clavier (entrée standard), les transforme et affiche le résultat à l'écran (sortie standard [1]). On indique la fin du texte saisi au clavier par un `Ctrl-D` (EOF **E**nd **O**f **F**ile) en début de ligne. Les commandes `cat`, `head`, `tail`, `wc`, `fold`, `cut`, `grep`, `sort` et `tr` sont des filtres [2]. Les filtres sont en général sensibles à l'environnement local (*cf.* 15.4, p. 106).

5.2 Classement avec `sort`

La commande `sort` trie, regroupe ou compare toutes les **lignes** des fichiers passés en paramètre. Si aucun nom de fichier n'est fourni après la commande, la lecture se fait depuis l'entrée standard, *i-e* le clavier.
Par défaut, `sort` effectue un tri sur les lignes d'entrée selon l'*ordre lexicographique* sur tous les caractères de chaque ligne (en commençant par la gauche) et affiche le résultat à l'écran. Mais il est possible de préciser des critères de tri plus précis : en découpant chaque ligne en champs séparés par un espace ou une tabulation, on peut spécifier avec l'option `-k` que le tri porte sur un ou plusieurs champs particuliers.

5.2.1 Principales options de `sort`

`-r` (**r**everse) pour trier selon l'ordre inverse ;

`-f` pour ignorer la casse (majuscule/minuscule) [3] ;

`-b` (**b**lank) pour ignorer les blancs en tête de champ ;

`-n` (**n**umeric) pour trier selon l'ordre numérique ;

`-u` (**u**nique) pour ne garder que le premier exemplaire des lignes ex-æquo du point de vue du classement (mais pas forcément identiques) ;

`-k`*`début`*`,`*`fin`* (**k**ey) classement selon les champs de numéro compris entre *`début`* et *`fin`* ;

`-t`*`délim`* permet de choisir le séparateur de champs *`délim`* (par défaut, il s'agit de la catégorie `[:blank:]` qui comporte les espaces et les tabulations).

♥ ⇒ **Recommandation** Si les champs sont alignés verticalement en utilisant des espaces multiples, il est prudent d'utiliser systématiquement l'option `-b` pour éviter que les espaces supplémentaires ne soient intégrés en tête de champ et influencent le classement.

1. Voir le chapitre 10 au sujet des entrée et sortie standard.
2. Pour faire travailler ces filtres avec des fichiers, on pourra rediriger leurs entrée ou sortie. Mais la plupart des filtres, notamment `cat`, `head`, `tail`, `wc`, `fold`, `cut`, `grep`, `sort` et `awk` (mais pas `tr`), admettent aussi le passage en paramètre d'un ou plusieurs noms de fichiers d'entrée. Noter que dans ce cas, ils peuvent exploiter le nom du fichier passé en paramètre, alors que ce n'est plus possible avec une redirection d'entrée. Cela se traduit par exemple avec `wc` par un affichage différent suivant la syntaxe utilisée : `wc -c fichier` affiche le nombre d'octets et le nom du fichier, alors que `wc -c < fichier` n'affiche que le nombre d'octets.
3. Avec une locale de type `C` ou `POSIX`, toutes les majuscules sont, par défaut, classées avant toutes les minuscules, selon l'ordre de la table des caractères ascii (*cf.* 15.4, p. 106). En revanche, avec une locale francisée, cette option n'est pas nécessaire.

Tris hiérarchisés Lorsque l'on souhaite classer d'abord selon un champ, puis classer les ex-æquo selon un autre champ, il faut utiliser plusieurs options `-k`. Bien noter que deux commandes `sort` reliées par un tube ne produiraient pas le même résultat : le deuxième classement détruirait le premier. ⇐⚠

Tris numériques Le tri numérique avec l'option `-n` fonctionne avec les entiers et les réels pour lesquels il faut que le séparateur décimal soit conforme à la variable de contrôle local `LC_NUMERIC` ou `LC_ALL` : la virgule en français et le point en anglais par exemple. ⇐⚠
Mais l'option `-n` ne permet pas de classer des valeurs numériques comportant le signe `+`[4], ni celles en virgule flottante (notation au format exponentiel). Il faut, pour cela, utiliser l'option `-g` (general numeric sort), qui n'est pas conseillée par ailleurs pour des raisons de performance. ⇐⚠

5.2.2 Exemples

1.
```
bhi az1
bhi au
cde 123
afh x
aaa 12 tuv
cdi
```
Si on saisit `sort`, puis les lignes ci-contre à gauche suivies de `Ctrl-D`, `sort` affichera alors les lignes de droite.
```
aaa 12 tuv
afh x
bhi au
bhi az1
cde 123
cdi
```

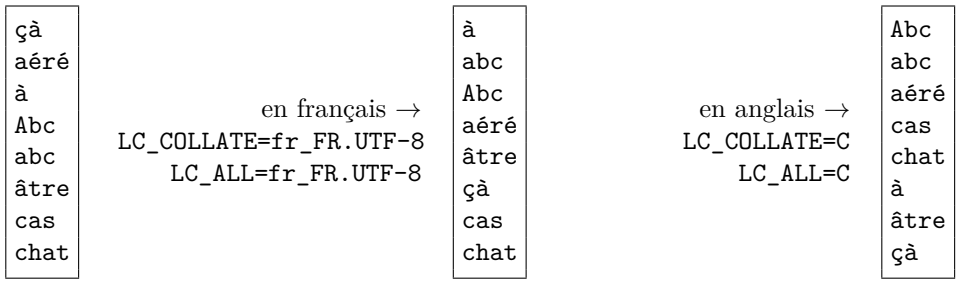
2. Dans l'exemple précédent, le tri porte sur la ligne d'entrée en intégralité. L'option `-k`*`début`*`,`*`fin`* où *`début`* et *`fin`* sont des entiers, permet d'effectuer le tri sur la portion de ligne comprise entre le champ numéroté *`début`* et le champ numéroté *`fin`* inclus, où les champs correspondent aux mots de la ligne d'entrée séparés par une ou plusieurs espaces (séparateur par défaut). Les options telles que `n` peuvent alors porter sur un champ particulier.
```
Albert 15
Benito  5
Chati  12
Dole   17
Estel   8.5
Valli  18.8
```
Si on saisit la commande `sort -b -k2,2n`, puis les lignes de gauche, suivies de `Ctrl-D`, `sort` affichera alors les lignes de droite, classées selon la valeur numérique du second champ.
```
Benito  5
Estel   8.5
Chati  12
Albert 15
Dole   17
Valli  18.8
```

3. `sort` permet de classer les fichiers listés par la commande `ls -l` avec leurs attributs (la taille est le 5e champ et le nom est supposé le 8e, *cf.* 2.3.1, p. 9) :

`ls -l | sort -b -k8,8`	classe les fichiers par ordre lexicographique
`ls -l | sort -b -k8,8r`	classe les fichiers par ordre lexicographique inverse de nom
`ls -l | sort -b -k5,5n`	classe les fichiers par ordre de taille croissante
`ls -l | sort -b -k5,5n -k8,8`	classe les fichiers par ordre de taille croissante puis les ex-æquo par ordre lexicographique
`ls -l | sort -b -k5,5n -u`	classe les fichiers par ordre de taille croissante et n'affiche qu'un exemplaire pour une taille donnée (fusion des ex-æquo)

4. Les variables de langue (`LC_ALL` et `LC_COLLATE`) ont une importance cruciale dans `sort`. Comparer (casse, accents, ...) les classements alphabétiques suivants :

4. Ainsi `sort -b -k1,1n` classe `+12` avant `+2` car le signe `+` non reconnu fait retomber en classement lexicographique.



5. `sort -t: -k3,3n /etc/passwd` affiche les lignes du fichier `passwd` (où le séparateur de champs est le caractère « : ») par numéro d'`id` (troisième champ) croissant.

5.3 Remplacement de caractères avec `tr`

`tr` (**tr**anslate) substitue (par défaut) dans le flux d'entrée à chaque caractère de l'ensemble fourni en premier paramètre son correspondant de l'ensemble du deuxième paramètre et affiche le résultat sur la sortie standard.

syntaxe

```
tr jeu1 jeu2
```

⚠⇒ **Restriction :** La commande `tr` n'admet pas de nom de fichier en paramètre : `tr` est un filtre « pur » [5]. Pour que `tr` lise les données d'entrée dans un fichier, il est nécessaire d'utiliser une redirection d'entrée (*cf.* 10.2.2, p. 73) :

```
tr jeu1 jeu2 < fichier
```

Exemples

La commande `tr 'b' 'B'` remplace les `b` par des `B` avant d'afficher le texte saisi sur l'entrée standard.

`le balbutiement des bebes` suivi de `Ctrl-D` est transformé en `le BalButiement des BeBes`

`tr 'abc' '123'` remplacera les `a` par des `1`, les `b` par des `2` et les `c` par des `3`.

5.3.1 Définition des jeux de caractères

Les jeux de caractères d'entrée et de sortie peuvent se recouvrir car `tr` travaille octet par octet : ainsi `tr 01 10` échange les `0` et les `1`.

À condition de les protéger de l'interprétation par le shell, des caractères de contrôle du code ascii désignés comme en langage C (`\r` pour carriage return, retour chariot par exemple) peuvent être utilisés dans les jeux d'entrée et de sortie.

La définition des jeux de caractères peut faire intervenir la notion d'intervalle lexicographique (`a-z` par exemple pour représenter les minuscules de l'ascii).

`tr '0-9' 'a-j'` remplacera les chiffres par les dix premières lettres de l'alphabet.

Mais les jeux de caractères peuvent être construits à partir de classes de caractères (*cf.* 6.3.3, p. 53), dépendant de la langue de travail, parmi lesquelles :

5. Si on lance la commande `tr` avec un troisième argument pour signifier le nom du fichier d'entrée, on obtient un message d'erreur du type `tr: opérande supplémentaire`.

`[:lower:]` les minuscules
`[:upper:]` les majuscules
`[:alpha:]` les lettres
`[:digit:]` les chiffres
`[:alnum:]` les lettres et les chiffres
`[:punct:]` les signes de ponctuation
`[:blank:]` les blancs horizontaux (espace et tabulation, *cf.* 5.5.3, p. 49)
`[:space:]` les blancs (espace, tabulation, fin de ligne, ...)

Ainsi `tr "[:lower:]" "[:upper:]"` remplace chaque lettre minuscule du flux d'entrée par la lettre majuscule correspondante et affiche le résultat à l'écran.
Malgré leurs similitudes, on ne confondra pas la syntaxe des jeux de caractères de `tr` avec celle des expressions régulières (*cf.* chapitre 6, p. 51).

5.3.2 Options de `tr`

L'option `-c` *jeu1* (**c**omplement) permet de définir le jeu de caractères à traiter comme le complément du jeu fourni en premier argument.
`tr -d` *jeu1* (**d**elete) permet de supprimer les caractères de cet ensemble au lieu de les transcrire.
`tr -s` (**s**queeze repeats) permet de remplacer les caractères *consécutifs* qui devraient être répétés en sortie par une seule occurrence. S'il n'y a qu'un jeu de caractères (`tr -s` *jeu1*), `tr` ne fait que supprimer les répétitions de caractères du jeu précisé ; si `tr` effectue une transformation d'un jeu dans un autre (`tr -s` *jeu1 jeu2*), la suppression des répétitions s'effectue après la transformation : elle porte donc sur le deuxième jeu.

Exemple 1

```
tr -d '\r'
```
permet de supprimer le retour chariot des fins de lignes des fichiers issus de WINDOWS.

Exemple 2

```
tr -cs '[:lower:][:upper:]' '[\n*]'
```
remplace tous les caractères non alphabétiques par un changement de ligne (`\n`). Le jeu de remplacement ne comprenant que le caractère `\n`, il est nécessaire de le faire suivre du multiplicateur `*` pour assurer un correspondant à chaque caractère du jeu d'entrée. C'est l'option `-s` qui permet finalement de fusionner les changements de ligne successifs en un seul. On crée ainsi une liste des mots présents dans le fichier d'entrée à raison d'un mot par ligne.

Remarque 1 : pour des raisons historiques liées aux deux branches d'UNIX, la commande `tr` admet des syntaxes différentes d'un système UNIX à l'autre et il est prudent de vérifier dans le manuel (`info tr`, plus complet que `man`) le détail des options disponibles.

Remarque 2 : on notera bien que `tr` travaille caractère par caractère et ne peut pas substituer un mot à un autre mot [6]. Plus précisément, `tr` travaille octet par octet [7] et il ne fonctionne pas sur les caractères multi-octets du codage UTF-8. Pour traiter par `tr` un fichier texte codé en UTF-8, on peut le transcoder via `iconv` ou `recode` en codage mono-octet comme ISO-8859-1, puis traiter le résultat par `tr` avec les variables de langage adaptées (cf. 15.4, p. 106), et enfin reconvertir la sortie en UTF-8. Ces transformations successives pourront s'effectuer à l'aide de redirections et de tubes, techniques décrites au chapitre 10, p. 72. Une autre solution pour les opérations simples portant sur des caractères codés en UTF-8 est d'utiliser la requête `y` de `sed` (*cf.* 8.2, p. 59). ⇐⚠

6. Pour substituer un mot à un autre, on utilisera `sed` (*cf.* Chap. 8, p. 59) avec la requête `s`
7. Au moins jusqu'à la version 7.1 de `tr`.

5.4 Autres filtres élémentaires

5.4.1 Aperçu d'un fichier avec `head` et `tail`

`head`/**`tail`** affiche les lignes du début/de la fin
Par défaut, **`head/tail`** affiche les 10 premières/dernières lignes de l'entrée standard ou du fichier passé en paramètre, mais
`head/tail -n` *p* **`[`***fic***`]`** affiche les *p* premières/dernières lignes du fichier *fic*.
Pour afficher les lignes à partir de la p^{e} incluse (sans avoir à calculer le nombre de lignes à afficher), il faut insérer le signe `+` devant le numéro p : `tail -n +p`.

5.4.2 Conversion des tabulations avec `expand` et `unexpand`

`expand/unexpand` traduit les tabulations en espaces et inversement. Par défaut, les tabulations sont placées tous les 8 caractères, mais on peut les choisir en précisant l'option `-t` suivie :

— soit de la liste de leurs positions, séparées par des virgules,
— soit, si elles sont sont équidistantes, d'un seul nombre indiquant leur espacement.

Par défaut, `unexpand` ne convertit que les blancs en début de ligne ; l'option `-a` permet de convertir potentiellement tous les blancs. Par exemple, partant du fichier `blancs` suivant,

`blancs`

```
123456781234567812345678123456781234567812345678123456781234
                  abc     ABCDEFGHI        zzz
```

le fichier `tabs` produit par la commande `unexpand blancs > tabs` ne comporte que deux tabulations représentées ici par `^I` [8] :

`visualisation de tabs sous vi avec option list`

```
123456781234567812345678123456781234567812345678123456781234$
^I^I abc     ABCDEFGHI        zzz$
```

alors qu'avec `unexpand -a blancs > tabs1`, on en obtient quatre :

`visualisation de tabs1 sous vi avec option list`

```
123456781234567812345678123456781234567812345678123456781234$
^I^I abc^IABCDEFGHI^Izzz$
```

comme on le voit aussi avec `od -t a` (`od` affiche ici 16 caractères par ligne [9], les fins de ligne sont marquées par `nl`, les blancs par `sp` et les tabulations par `ht`) :

`visualisation avec od -v -A n -t a`

```
   1   2   3   4   5   6   7   8   1   2   3   4   5   6   7   8
   1   2   3   4   5   6   7   8   1   2   3   4   5   6   7   8
   1   2   3   4   5   6   7   8   1   2   3   4  nl  ht  ht  sp
   a   b   c  ht   A   B   C   D   E   F   G   H   I  ht   z   z
   z  nl
```

5.4.3 Repliement de lignes avec `fold`

`fold` permet de replier les lignes de longueur supérieure à 80 colonnes, ou une longueur ♥ ⇒ spécifiée par l'option `-w`. L'option `-s` permet de ne couper que sur des espaces.

8. Pour visualiser les tabulations, on peut utiliser l'éditeur de texte `vi` avec l'option `:set list` (*cf.* 4.3.3, p. 36) : elles apparaissent alors sous la forme `^I`. Une autre méthode consiste à employer la commande `od -t c` ou `od -t a` (*cf.* 4.1.1, p. 30) : les tabulations sont alors affichées sous la forme `\t` avec l'option `-t c` ou `ht` (horizontal tabulation) avec l'option `-t a`.

9. L'option `-v` impose l'affichage explicite des lignes successives identiques, au lieu du seul caractère `*`, affiché par défaut à la place des lignes dupliquées.

Exemple

Repliement à au plus 30 caractères des lignes du fichier `lignes-longues`

lignes-longues
```
Ceci est un fichier avec des lignes dépassant les 30 caractères et
la commande fold le ramène à des lignes d'au plus 30 caractères,
plus ou moins intelligemment suivant l'option "-s"
```

`fold -w 30 lignes-longues > max30`

max30
```
Ceci est un fichier avec des l
ignes dépassant les 30 caractè
res et
la commande fold le ramène à d
es lignes d'au plus 30 caractè
res,
plus ou moins intelligemment s
uivant l'option "-s"
```

`fold -sw 30 lignes-longues > max30s`

max30s
```
Ceci est un fichier avec des
lignes dépassant les 30
caractères et
la commande fold le ramène à
des lignes d'au plus 30
caractères,
plus ou moins intelligemment
suivant l'option "-s"
```

♠ 5.4.4 Sélection de colonnes avec `cut`

```
cut -f liste [fichier_d_entrée]
```

affiche sur la sortie standard la partie de chaque ligne du fichier *`fichier_d_entrée`* (l'entrée standard s'il n'est pas précisé) constituée par les champs [10] dont les numéros figurent dans *`liste`*. La liste peut être explicite, le séparateur étant la virgule [11] « `,` », ou comporter des intervalles dont les bornes sont séparées par le signe moins « `-` ». Les champs sont délimités par défaut par des tabulations, mais le délimiteur peut être modifié grâce à l'option `-d`. La sélection peut aussi porter sur les octets ou les caractères avec les options `-b` (**b**yte) ou `-c` (**c**haracter) à la place de `-f` (**f**ield).

Exemple

On rappelle le format d'une ligne du fichier de mots de passe (`/etc/passwd`)
`p6m1mni:!:40369:2055:Jacques LEFRERE:/home/p6pfal/p6m1mni:/usr/bin/ksh`
`cut -d : -f 1,7 /etc/passwd` affiche les champs 1 et 7 du fichier de mots de passe (nom d'utilisateur et shell de login).

♠ 5.5 Fusion de fichiers texte

Les deux commandes suivantes ne sont pas des filtres, mais des outils pour fusionner ligne à ligne des données issues de fichiers texte : `paste` concatène les lignes dans l'ordre, alors que `join` s'appuie sur un champ commun pour fusionner les lignes en correspondance.

♠ 5.5.1 Concaténation de lignes avec `paste`

```
paste [liste_de_fichiers_d_entrée]
```

concatène ligne par ligne (« horizontalement ») les fichiers d'entrée spécifiés dans la liste et affiche le résultat sur la sortie standard. La ligne i de la sortie comporte donc les $i^{\text{èmes}}$ lignes de chacun des fichiers d'entrée, dans l'ordre et séparées par défaut par des

10. Remarque : la sélection par champ peut aussi être effectuée avec le filtre `awk`, *cf.* chapitre 9, qui permet aussi de modifier l'ordre des champs, alors que `cut` les restitue dans l'ordre initial, quel que soit l'ordre dans la liste.
11. Ne pas insérer d'espace après la virgule.

tabulations . Mais on peut fournir une liste de séparateurs grâce à l'option `-d`. La sortie comporte autant de lignes que le le fichier d'entrée qui en a le plus, quitte à imaginer dans des lignes vides dans les autres fichiers d'entrée.

Exemple 1

Concaténation des lignes des fichiers `notes-cc` et `notes-ex`

```
paste notes-cc notes-ex > notes-cc+ex
```

notes-cc

```
Durand  12 14
Leblanc 15 17
Rosier  09 11
```

notes-ex

```
Durand  10
Leblanc 13
Rosier  17
```

notes-cc+ex

```
Durand  12 14        Durand  10
Leblanc 15 17        Leblanc 13
Rosier  09 11        Rosier  17
```

Exemple 2

Concaténation des lignes des fichiers `notes-cc` et `notes2-ex`

```
paste notes-cc notes2-ex > notes2-cc+ex
```

notes-cc

```
Durand  12 14
Leblanc 15 17
Rosier  09 11
```

notes2-ex

```
Leblanc 13
Rosier  17
Untel   10
```

notes2-cc+ex

```
Durand  12 14        Leblanc 13
Leblanc 15 17        Rosier  17
Rosier  09 11        Untel   10
```

⚠⇒ Attention, *aucun* contrôle de cohérence entre les lignes n'est effectué avec `paste` : si on souhaite s'appuyer sur un champ commun pour associer les lignes, il faut utiliser la commande `join`.

♠ 5.5.2 Fusion de lignes selon un champ commun avec `join`

```
join fichier1 fichier2
```

fusionne les lignes ayant un premier **champ commun** dans les fichiers *fichier1* et
⚠⇒ *fichier2*, qui doivent être *préalablement triés* selon le champ qui pilote la fusion. Par défaut, seules les lignes pour lesquelles une coïncidence du champ de fusion est trouvée entre les deux fichiers sont affichées sur la sortie standard.

Par défaut, le séparateur de champs en entrée est l'espace (simple ou multiple), mais il peut être modifié en utilisant l'option `-t` suivie du ou des caractères séparateurs de champs. Les champs pilotant la fusion peuvent aussi être spécifiés pour chaque fichier via les options `-1` *n1* et `-2` *n2* où *n1* et *n2* concernent respectivement les deux fichiers d'entrée *fichier1* et *fichier2*.

Exemple 1

Fusion des fichiers `notes-cc` et `notes-ex` (selon le premier champ)

```
join notes-cc notes-ex > notes
```

notes-cc

```
Durand  12 14
Leblanc 15 17
Rosier  09 11
```

notes-ex

```
Durand  10
Leblanc 13
Rosier  17
```

notes

```
Durand 12 14 10
Leblanc 15 17 13
Rosier 09 11 17
```

Exemple 2

Fusion des fichiers `notes-cc+prenom` et `notes-ex` selon le champ du nom (2e champ dans le premier fichier, d'où l'option `-1 2`)

```
join -1 2 notes-cc+prenom notes-ex > notes+prenom
```

`notes-cc+prenom`

```
Jean Durand  12 14
Paul Leblanc 15 17
Yves Rosier  09 11
```

`notes-ex`

```
Durand  10
Leblanc 13
Rosier  17
```

`notes+prenom`

```
Durand Jean 12 14 10
Leblanc Paul 15 17 13
Rosier Yves 09 11 17
```

5.5.3 Mise en garde concernant les caractères non imprimables

La présence de caractères de contrôle dans les fichiers texte peut provoquer des résultats inattendus quand ces fichiers sont traités par des filtres. Sous un éditeur de texte ou lors d'un affichage, la plupart de ces caractères ne sont en effet pas visibles par défaut, ou peuvent être confondus. On peut citer par exemple, avec leurs séquences d'échappement en langage C, leur code ascii en octal et en décimal (*cf.* `man ascii`) : ⇐⚠

- la tabulation (horizontal tabulation, HT), notée `\t`, `011` en octal et `09` en décimal ;
- le retour chariot (carriage return, CR), noté `\r`, `015` en octal et `13` en décimal ;
- le saut de ligne (line feed, LF), noté `\n`, `012` en octal et `10` en décimal.
- le retour arrière (backspace, BS), noté `\b`, `010` en octal et `08` en décimal.
  Le retour arrière est le caractère le plus trompeur car il provoque le remplacement du caractère à sa gauche par le caractère à sa droite, d'où son utilisation comme touche de correction.

Codage des fins de ligne dans les fichiers texte

Les fichiers texte issus des outils DOS ou Windows utilisent la combinaison `\r\n` pour marquer les fins de ligne alors qu'unix se contente de `\n`[12]. Si on transfère des fichiers texte Windows en mode binaire vers un système unix (par exemple via une clef USB), les fins de ligne ne sont pas converties à la convention unix. Dans ce cas, la présence du `\r` avant le `\n` va provoquer des comportements surprenants si on cherche à ajouter des éléments en fin de ligne, car ils vont s'interposer entre `\r` et `\n`. Ils vont donc s'afficher en début de ligne en écrasant les caractères qui y figuraient auparavant. Par exemple, le filtrage `awk '{print $0 "texte"}'` va afficher le fichier de type DOS fourni en entrée avec les cinq premiers caractères de chaque ligne remplacés par `texte`. ⇐⚠

Alignement de colonnes avec des tabulations

Les tabulations sont considérées comme des séparateurs de champs par défaut par certains filtres (`cut`, `paste`, ...). Elles appartiennent à la classe des blancs `[:blank:]` (*cf.* 6.3.3, p. 53), qui comporte aussi l'espace (mais dépend de la langue de travail) et sert de séparateur par défaut pour d'autres filtres (`sort`, `awk`). Mais les tabulations ne sont pas un séparateur de champs par défaut pour `join`.
Les tabulations peuvent être confondues avec des espaces en affichage ou en édition. Mais il ne faut pas les confondre dans les motifs utilisés sous `grep`, `sed` et `awk`. Par exemple, `grep ' '` affiche les lignes comportant au moins un espace, mais pas celles où les seuls blancs sont produits par des tabulations. En revanche, `grep '[[:blank:]]'` affiche les lignes qui contiennent au moins un espace ou une tabulation.

12. Le système Mac-OS, jusqu'à la version 9, utilisait `\r`.

Visualisation des caractères non imprimables

Plusieurs outils d'affichage permettent de repérer certains caractères non imprimables dans un texte, parmi lesquels :

`cat :` `cat -v` affiche les caractères non imprimables avec la notation `^` pour contrôle, sauf pour les sauts de ligne et les tabulations : les retour chariot s'affichent `^M` et les retour arrière `^B`
`cat -E` (**E**nd) affiche les fins de ligne sous la forme `$`
`cat -T` (**T**abs) affiche les tabulations sous la forme `^I`
`cat -A` (**A**ll) est équivalent à `cat -vET`

`vi :` L'option `list` de l'éditeur `vi`, activée par `:set list`, affiche les tabulations sous la forme `^I` et les fins de ligne sous la forme `$`
Si `vi` affiche `[dos]` sur la ligne d'état, cela signifie que le type du fichier est repéré grâce à la présence de retour chariot en fin de ligne. Mais il faut lancer l'édition en mode binaire via `vi -b` pour voir apparaître ces caractères sous la forme `^M`

`od :` L'outil d'affichage de fichiers binaires `od -tc` affiche `\t` pour tabulation, `\r` pour retour chariot, `\n` pour le changement de ligne, `\b` pour retour arrière.

Suppression de caractères non imprimables

Le filtre `expand` permet de traduire les tabulations par le nombre d'espaces nécessaires.
Le filtre `tr` permet de transcrire ou supprimer les caractères non imprimables désignés par les séquences d'échappement du langage C : `\t` pour tabulation, `\r` pour retour chariot, `\n` pour saut de ligne, `\b` pour retour arrière...

Chapitre 6

Introduction aux expressions rationnelles

6.1 Introduction

Pour décrire l'ensemble des chaînes de caractères qui satisfont à un motif (pattern) générique, on utilise une syntaxe particulière qualifiée d'*expression rationnelle*[1] (Regular Expression). De nombreuses commandes et utilitaires UNIX font appel aux expressions rationnelles pour rechercher des motifs génériques :

— les éditeurs `ex`, `vi`, `emacs` et `asedit` ;
— les filtres `sed`, `grep` (**g**lobal **r**egular **e**xpression **p**rint) et `awk`.
— d'autres outils comme `perl`, `python`, `php`, `JavaScript`...

Suivant les commandes et les systèmes, deux versions de la syntaxe des expressions rationnelles peuvent être utilisées, mais de façon exclusive :

— les expressions rationnelles de base **BRE** : **B**asic **R**egular **E**xpressions
— les expressions rationnelles étendues **ERE** : **E**xtended **R**egular **E**xpressions

On décrira d'abord la syntaxe des BRE, utilisées par défaut par `grep`, `ex`, `vi` et `sed`.

Exemples d'application

Les exemples qui suivent utilisant les filtres `grep` et `sed` peuvent être testés avec l'entrée et la sortie standard (clavier et écran).

`grep 'chaine'`	affiche toutes les lignes contenant `chaine`
`grep '^a'`	affiche toutes les lignes commençant par `a`
`grep '^chaine'`	affiche toutes les lignes commençant par `chaine`
`grep '^a*b'`	affiche toutes les lignes commençant par `b`, éventuellement précédé par un nombre quelconque de `a`
`sed 's/[aA]/A+/g'`	remplace (**s**ubstitute) toutes les occurrences de `a` ou `A` par `A+`
`sed 's/[aA]\{3\}/A3/g'`	remplace (**s**ubstitute) toutes les suites de trois `a` (minuscule ou majuscule) par `A3`

6.2 Signification des trois caractères spéciaux de base

Pour construire des motifs génériques, il faut conférer un rôle particulier à certains caractères qui doivent être interprétés : ils sont qualifiés de caractères spéciaux ou méta-caractères.

`.` (point)	représente un caractère quelconque et un seul.
`\` (contre-oblique) (backslash)	sert à protéger le caractère qui le suit pour empêcher qu'il ne soit interprété : il retrouve son sens littéral. La contre-oblique permet aussi de rendre « spéciaux » certains symboles comme les parenthèses et les accolades dans les BRE.
`*` (étoile)	quantificateur qui représente un nombre d'occurrences quelconque (**zéro**, une ou plusieurs occurrences) du caractère ou de la sous-expression (en cas de groupement, *cf* 6.4, p. 54) qui précède.

1. On dit aussi expression régulière.

⚠⇒ **Mise en garde** Ne pas confondre les caractères spéciaux des expressions rationnelles avec les caractères génériques (wildcards) pour les noms de fichiers, `*` et `?` qui sont, eux, interprétés par le shell (*cf.* 11.2.1, p. 81).

Correspondance :	pour le shell	expr. rationnelles
un caractère quelconque	`?`	`.`
un nombre quelconque de caractères	`*`	`.*`

Noter aussi que, malgré certains points communs, `tr` utilise une syntaxe différente.

Exemples

`a*`	un nombre quelconque de fois le caractère `a` (y compris une chaîne vide)
`aa*`	une ou plusieurs fois le caractère `a`
`.*`	un nombre quelconque de caractères quelconques (y compris une chaîne vide)
`..*`	au moins un caractère
`\..`	un point suivi d'un caractère quelconque
`\\*`	un nombre quelconque (y compris zéro) de contre-obliques

6.3 Caractères devenant spéciaux dans certaines positions

6.3.1 Ancres

Les ancres (anchor) ne représentent aucune chaîne, mais permettent de spécifier qu'un motif est situé en début ou en fin de ligne :

`^` (accent circonflexe : caret)	spécial **en début** de motif, représente le début de ligne
`$` (dollar)	spécial **en fin** de motif, représente la fin de ligne

Les début et fin de mot peuvent aussi être représentés respectivement par les ancres `\<` et `\>`, permettant d'affiner les recherches de motifs.

Exemples

`^a`	une ligne commençant par un `a`
`a$`	une ligne finissant par un `a`
`^a$`	une ligne constituée d'un `a`
`^$`	une ligne vide
`^.*$`	une ligne quelconque, y compris vide
`^..*$`	une ligne non vide
`^a.*b$`	une ligne commençant par `a` et finissant par `b`
`^$.*$$`	une ligne commençant et finissant par `$` (contenant au moins deux fois `$`) seul le dernier `$` est spécial
`^^.*^$`	une ligne commençant et finissant par `^` (contenant au moins deux fois `^`) seul le premier `^` est spécial
`\<le\>`	l'article `le` mais pas la syllabe `le` à l'intérieur d'un mot

Applications

`sed 's/^/#  /'`	insère un `#` suivi de 2 espaces en début de chaque ligne
`grep '^..*$'`	affiche les lignes non vides

6.3.2 Ensembles de caractères

Les ensembles de caractères (parmi lesquels un caractère quelconque et un seul peut être choisi) sont spécifiés entre crochets : `[`*`ensemble-de-caractères`*`]`. À l'intérieur d'un tel ensemble, les caractères spéciaux sont :

`-` (signe moins)	utilisé pour définir des intervalles selon l'ordre lexicographique
`^` (accent circonflexe)	en tête pour spécifier le complémentaire de l'ensemble
`]` (crochet fermant)	qui délimite la fin de l'ensemble, sauf s'il est placé en première position

Dans un tel contexte, `.`, `*`, `\` et `$` sont des caractères ordinaires, considérés littéralement.

Exemples

`[a0+]`	un des trois caractères `a`, `0` ou `+`
`[a-z]`	une lettre minuscule (ascii)
`[a-z_]`	une lettre minuscule ou un souligné
`[0-9]`	un chiffre
`[^0-9]`	n'importe quel caractère qui n'est pas un chiffre
`[]-]`	un crochet fermant `]` ou un signe moins `-`

Remarque Bien noter que le caractère `-` permet de définir des intervalles uniquement à l'intérieur des crochets et que ses intervalles ne doivent pas être interprétés comme numériques. En particulier `[24-61]` ne désigne pas un nombre entre 24 et 61, mais un caractère parmi `2`, `4`, `5`, `6` ou `1`. Ici l'intervalle est entre 4 et 6. ⇐⚠

Applications

`grep '^[0-9]'`	affiche les lignes commençant par un chiffre
`grep '^[^0-9]'`	affiche les lignes ne commençant pas par un chiffre

6.3.3 Classes de caractères

La norme POSIX définit des classes de caractères (character class), notées entre les symboles `[:` et `:]` [2], dont le contenu dépend de la langue de travail choisie et de l'ordre lexicographique (variables `LANG`, `LC_CTYPE` et `LC_COLLATE`, *cf.* `man locale` et 15.4.1, p. 106). Citons simplement certaines classes dont les noms sont assez explicites :
`[:alpha:]` (lettres), `[:lower:]` (minuscules), `[:upper:]` (majuscules),
`[:digit:]` (chiffres), `[:xdigit:]` (symboles hexadécimaux de `0` à `f` ou `F`),
`[:alnum:]` (chiffres et lettres),
`[:print:]` (affichables), `[:punct:]` (ponctuation), `[:cntrl:]` (caractères de contrôle),
`[:blank:]` (espace et tabulation),
`[:space:]` (blanc, retour ligne, retour chariot, saut de page, ..., *cf.* 5.5.3, p. 49)

On préférera, par exemple, utiliser `[[:alnum:]]`, qui représente l'ensemble des caractères alphanumériques, y compris avec des signes diacritiques, au lieu de `[0-9A-Za-z]` qui ne comporte pas les caractères accentués. ⇐♥
Remarquer que les crochets dans les noms de classes font partie intégrante du nom symbolique, et qu'ils doivent donc être inclus en plus des crochets encadrant la liste. ⇐⚠

2. La norme POSIX définit aussi des classes d'équivalence rassemblant l'ensemble des caractères devant être considérés comme équivalents, par exemple `[=e=]` qui recouvre `e`, `é`, `è`... en français par exemple. Enfin elle définit la notation `[.ll.]` en espagnol par exemple qui, bien que constituée de deux caractères, doit être considérée comme une seule unité lexicographique, notamment pour les classements (il en est de même pour `[.ch.]`). Mais ces deux dernières notions ne sont pas encore pleinement implantées à ce jour dans les systèmes UNIX.

Exemples

`[[:lower:]+^$]`	une minuscule ou un des symboles `+`, `^` ou `$`
`[^[:alpha:]]`	un caractère non alphabétique
`[-+.[:digit:]]`	un chiffre, ou un des signes `+` ; `-` ou `.` employés pour écrire des nombres décimaux

6.3.4 Référence dans les substitutions

Dans `sed`, `ex` et `vi`, il est possible de référencer la chaîne de caractères du membre de gauche de la substitution qui correspond à la totalité de l'expression rationnelle par le symbole `&` en second membre de la substitution.

Applications

`sed 's/.*/&&/'`	duplique la ligne en une seule ligne
`sed 's/./&&/g'`	duplique tous les caractères
`sed 's/^[0-9][0-9]*$/(&)/'`	met entre parenthèses les lignes de chiffres

6.4 Groupement en sous-expressions et référence

Une partie d'expression rationnelle peut être définie comme sous-expression à condition de l'encadrer par les délimiteurs `\(` et `\)` (en effet, pour les BRE, les parenthèses ne sont pas des caractères spéciaux). La référence à la n-ième sous-expression se fait par `\n`.

Applications

`sed 's/0\([0-9]\)/200\1/'`	précise le siècle pour les années `200x`
`sed 's/19\([0-9][0-9]\)/\1/'`	supprime `19` devant les années `19xx`
`sed 's/\([0-9]\)\([0-9]\)/\2\1/'`	échange deux chiffres consécutifs

Spécification plus précise des répétitions

Le quantificateur « `*` » ne permet pas de préciser le nombre de répétitions d'une expression rationnelle qu'il autorise ; les syntaxes de quantification délimitées par les accolades (précédées de `\` car elles ne sont pas spéciales dans les BRE) `\{` et `\}` permettent de fixer des limites aux répétitions :

`\{m\}`	exactement `m` fois
`\{m,\}`	au moins `m` fois
`\{m,n\}`	entre `m` et `n` fois

Exemple

`[[:digit:]]\{2,\}`	désigne un nombre d'au moins deux chiffres

♠ 6.5 Règle en cas d'ambiguïté d'interprétation

Au cas où certains caractères spéciaux pourraient donner lieu à plusieurs interprétations, l'interprétation se fait toujours en essayant d'incorporer le plus grand nombre de caractères dans la correspondance avec l'expression rationnelle. Il est nécessaire de tenir compte de cette « avidité » dans les groupements avec délimiteurs.

Exemple :

Dans la ligne `a1!b2!c!`, extraire le premier champ, sachant que le séparateur est `!`
Avec `\(.*\)!\(.*\)!`, la première sous-expression `\1` sera `a1!b2` et la deuxième sera `c`
Pour que `\1` donne `a1`, il faudra préciser que la première sous-expression ne doit contenir aucun caractère `!` via `\([^!]*\)!\(.*\)!` ⇐⚠

Un exemple particulier : `*` signifie aussi zéro fois

`sed 's/@*/+/g'`

appliqué à `123z` donne `+1+2+3+z+`	plusieurs fois zéro occurrence de `@`
appliqué à une ligne vide donne `+`	zéro occurrence de `@`
appliqué à `@@@` donne `+`	à cause de l'avidité du `*`
appliqué à `@@@ab@@c` donne `+a+b+c+`	

♠ 6.6 Cas des expressions rationnelles étendues

Les expressions rationnelles étendues (ERE) sont utilisées dans `awk` (*cf.* chap. 9, p. 63) et éventuellement dans `sed` avec l'option `-r` (*cf.* 8.1.3, p 59) ou dans `grep` avec la version `egrep` ou l'option `grep -E` (*cf.* 7.4.2, p 57). Dans le passage des expressions rationnelles de base aux expressions étendues, les caractères `()` pour le groupement et `{}` pour la duplication deviennent spéciaux et doivent donc être protégés par `\` pour retrouver leur valeur littérale.
De plus, l'opérateur `|` permet d'exprimer une alternative (ou logique) entre deux sous-expressions. Enfin, deux opérateurs de multiplicité permettent de compléter le quantificateur trop général `*` :

`?` signifie zéro ou une occurrence de la sous-expression qui le précède (c'est un synonyme de `{0,1}`)

`+` signifie une ou plusieurs occurrences de la sous-expression qui le précède (c'est un synonyme de `{1,}`)

En conséquence, dans les ERE, les caractères littéraux `|`, `?` et `+` seront obtenus respectivement via `\|`, `\?` et `\+`.

Exemples d'expressions rationnelles étendues

`(ab)*`	signifie zéro, une ou plusieurs fois `ab` et correspond donc à une chaîne vide, `ab`, `abab`, ou `ababab`, ...
`a{2,4}`	signifie `aa`, `aaa` ou `aaaa` mais `\{2a` correspondra à la chaîne `{2a`
`a|b`	signifie `a` ou `b`
`(aa)|(bc)`	signifie `aa` ou `bc`
`[^[:alpha:]]*[[:lower:]]+[^[:alpha:]]*`	permet de détecter un mot comportant au moins une lettre en minuscule
`^[+-]?[[:digit:]]+`	représente une ligne qui commence par un nombre entier avec éventuellement un signe

Ce chapitre ne donne qu'un très bref aperçu de l'usage des expressions régulières, qui constitue le sujet de plusieurs ouvrages. On pourra notamment consulter le manuel de Desgraupes (2001), qui aborde aussi leur emploi dans d'autres utilitaires (`perl`, `tcl`, `python` et `JavaScript`).

Chapitre 7

Le filtre grep

7.1 Présentation de la commande grep

Le filtre **grep** permet d'afficher toutes les lignes d'un fichier contenant un motif donné. Ce motif peut être défini explicitement ou sous la forme d'une expression régulière (*cf.* chap. 6) représentant un motif générique (**grep** = **g**lobal **r**egular **e**xpression **p**rint).

syntaxe

```
grep motif [fichier_1 [fichier_2 ...]]
```

Exemple : `grep p6m1mni /etc/passwd`
affiche la ligne de l'utilisateur `p6m1mni` figurant dans le fichier de mots de passe.

⚠⇒ Dès que le motif recherché comporte des caractères que le shell peut interpréter, il est nécessaire de les protéger par exemple par des « ' » (*cf.* 11.2.4, p 83).

Il est courant d'utiliser la commande **grep** pour sélectionner des lignes particulières dans un affichage trop long pour une recherche visuelle fiable : on redirige le flux de sortie vers le flux d'entrée de la commande **grep** grâce à un tube (*cf.* chap. 8).

Exemple : `ps -ef | grep clock`
affiche les processus qui comportent `clock` dans la ligne de description.

Le code de retour (*cf.* 11.3, p. 84) de **grep** vaut `0` si la commande trouve le motif cherché.

Remarque :
La commande **grep** n'est elle-même qu'un cas particulier d'utilisation de l'éditeur ligne **ex**. Le chapitre 8 présente l'utilisation du filtre programmable **sed** qui s'appuie sur l'éditeur ligne **ed**. Le travail sur la ligne de commande de **vi** permet donc de se familiariser avec une syntaxe commune à plusieurs commandes UNIX.

7.2 Principales options de grep

-i (**i**gnore case) rend la recherche insensible à la casse du motif (pas de distinction entre majuscules et minuscules)

-v (in**v**erse) affiche les lignes ne contenant pas le motif indiqué ;

-l (**l**ist) affiche la liste des fichiers contenant le motif au lieu des lignes le contenant ;

-n (**n**umber) affiche les lignes contenant le motif précédées de leur numéro ;

-c (**c**ount) affiche les noms des fichiers [1] et pour chaque fichier, le nombre de ses lignes [2] qui contiennent le motif ;

-C *n* (**C**ontext) affiche *n* lignes de contexte avant et après la ligne comportant le motif (extension GNU).

1. Sauf quand **grep** ne s'applique qu'à un fichier ou quand il est alimenté via l'entrée standard.
2. Ne pas confondre avec le nombre d'occurrences du motif recherché dans le fichier ; voir à ce propos l'option **-o**, *cf.* 7.4.3, p. 58.

7.3 Exemples

`grep '^!' test.f90`	affiche les lignes commençant par `!` dans `test.f90`
`grep '^ *!' test.f90`	affiche les lignes dont le premier caractère non blanc est `!` dans `test.f90`
`grep -v '^ *!' test.f90`	affiche les lignes qui ne sont pas des commentaires en fortran90
`grep '[Ee]xemple' texte`	recherche le mot `exemple` éventuellement capitalisé (premier caractère en majuscule)
`ls -l | grep ^d`	permet d'afficher la liste des sous-répertoires du répertoire courant avec leurs attributs
`grep -l printf *.c`	affiche la liste des fichiers source en C qui appellent la fonction `printf`

7.4 Autres options

7.4.1 Recherche de plusieurs motifs avec l'option `-e`

Si on doit rechercher les lignes contenant un motif parmi plusieurs, il suffit de faire précéder chacun des motifs par l'option `-e` :
`grep -e` *`motif1`* `-e` *`motif2 fichier`*
affiche les lignes qui contiennent *`motif1`* ou[3] *`motif2`*
N.-B. : la recherche des lignes contenant à la fois *`motif1`* et *`motif2`* se fait par l'intermédiaire d'un tube : `grep` *`motif1 fichier`* `|` `grep` *`motif2`*
L'option `-e` constitue aussi un moyen de spécifier un motif commençant par le signe « - », évitant ainsi la confusion entre motif et option.

♠ 7.4.2 Variantes de `grep` : options `-F` et `-E`

La commande `grep` attend par défaut une expression régulière de base (BRE) pour définir le motif recherché. Elle admet deux variantes pour lesquels le motif peut être :

— un motif statique avec la commande l'option `-F` de `grep`[4] (**fast**) plus rapide dans ses recherches. Avec cette option, les caractères habituellement spéciaux du motif ne sont plus interprétés.

— une expression régulière étendue (ERE, *cf.* 6.6, p. 55) avec l'option `-E` de `grep`[5].

`motif``+`	une ou plus d'une occurrence de *`motif`*
`motif``?`	zéro ou une occurrence de *`motif`*
`motif1``|`*`motif2`*	*`motif1`* ou *`motif2`* (protéger le symbole « `|` » de l'interprétation comme un tube par le shell)
`(`*`motif`*`)`	mise entre parenthèses pour regrouper des motifs

Exemples :

`grep 'a.b'`	recherche un `a` suivi d'un caractère quelconque suivi d'un `b`
`grep -F 'a.b'`	recherche la suite de trois caractères `a.b`
`grep 'ab|AB'`	recherche le motif unique `ab|AB`, mais
`grep -E 'ab|AB'`	recherche le motif `ab` ou le motif `AB`
`grep -E '(ab)+c'`	recherche `abc`, `ababc`, `abababc`, ...
`grep -E 'a(x|y)b'`	recherche `axb` ou `ayb` (comme `grep -e axb -e ayb`)

3. La syntaxe étendue des motifs de recherche disponible avec `grep -E` (*cf.* 7.4.2) permet de formuler la même recherche grâce à l'alternative `|`.
4. La commande `fgrep`, équivalente à `grep -F`, est obsolète.
5. La commande `egrep`, équivalente à `grep -E`, est obsolète.

♠ 7.4.3 Affichage des chaînes correspondant au motif : option -o

Ajuster une expression régulière complexe est une tâche parfois délicate qui peut être clarifiée en testant le motif générique sur des données saisies au clavier (entrée standard). Avec l'option `-o` (`--only-matching`), `grep` n'affiche plus les lignes complètes comportant le motif, mais seulement chacune des chaînes de la ligne qui correspondent au motif spécifié pour `grep`, chaque chaîne étant affichée sur une ligne distincte.

Exemple :
`grep -o '[0-9][0-9]*'` suivi de la saisie

```
5septembre2009
```

puis `Ctrl-D`, va afficher

```
5
2009
```

On remarquera que le multiplicateur `*` correspond à zéro fois dans la première coïncidence, et à trois fois dans la seconde coïncidence (le multiplicateur cherche à englober le plus de caractères possibles : on le qualifie d'avide).
À titre pédagogique, on peut préférer la colorisation des chaînes correspondant au motif spécifié par l'expression régulière dans les lignes affichées par `grep` avec l'option `--color=auto`. Avec la commande `grep --color=auto '[0-9][0-9]*'` suivie de la saisie précédente, la ligne saisie est entièrement affichée et les chaînes `5` et `2009` sont colorisées.

```
5septembre2009
```

L'option `-o` de `grep` fournit un moyen de compter le nombre de motifs présents dans un fichier au lieu du nombre de lignes comportant ce motif :
`grep -o` *`motif fichier`* `| wc -l`

Chapitre 8

Le filtre sed

8.1 Présentation de sed

8.1.1 Principes de fonctionnement et intérêt

sed (sed = stream editor) est un filtre programmable de manipulation de fichiers texte, directement issu de l'éditeur ligne ed, dont il emprunte les requêtes [1]. Il lit chaque ligne du fichier d'entrée (ou à défaut, de l'entrée standard), effectue les transformations spécifiées par les requêtes et affiche le résultat sur la sortie standard, mais ne modifie jamais [2] le fichier d'entrée.

8.1.2 Les deux syntaxes possibles

syntaxe

```
sed -f fichier_d_instructions.sed [fichier_de_données]
```

Les instructions ou requêtes stipulées dans *fichier_d_instructions*.sed (introduit par l'option -f) sont exécutées sur le contenu du *fichier_de_données*. En l'absence de fichier d'entrée, sed traite l'entrée standard.
Dans le cas d'instructions courtes, il est possible de saisir ces dernières sur la ligne de commande. Le premier paramètre de la commande constitue alors la liste des instructions fournies à sed :

```
sed instructions fichier_de_données
```

Cependant, pour éviter l'interprétation par le shell des métacaractères éventuellement présents dans les instructions sed, il est souvent nécessaire de protéger globalement les instructions en les entourant par des « ' » (*cf.* 11.2.4 p. 83) : ⇐⚠

```
sed 'instructions' fichier_de_données
```

8.1.3 Autres options de sed

- **-e** Il est préférable d'introduire les instructions sed par l'option -e. Cette option est d'ailleurs nécessaire pour introduire sur la ligne de commande plusieurs mots (au sens du shell) contenant des instructions sed (et éviter la confusion avec le nom du fichier d'entrée) : ⇐ ♥

  ```
  sed -e 'instruction1' -e 'instruction2' fichier_de_données
  ```

- **-n** Il est parfois utile de supprimer l'affichage automatique du résultat notamment quand sed fait appel à la requête p d'impression
- **-r** Le filtre sed travaille par défaut avec les expressions rationnelles de base : l'option -r permet d'utiliser la syntaxe des expressions rationnelles étendues (*cf.* 6.6, p. 55).

8.2 Requêtes principales

8.2.1 Substituer

s/motif1/motif2/ (susbstitute)

1. ed est lui-même très proche de l'éditeur ligne ex, sur lequel est construit l'éditeur pleine page vi.
2. Sauf si on utilise l'option -i, qui est déconseillée dans les phases de mise au point.

remplace la première occurrence de `motif1` par `motif2` dans chaque ligne où `motif1` est présent.
`s/motif1/motif2/g` remplace toutes les occurrences de `motif1` par `motif2` grâce à l'option `g` (**g**lobal).

Remarque Le délimiteur « `/` » n'est pas imposé : c'est simplement le caractère qui suit la requête `s`. Il devient alors un caractère spécial dans le motif recherché et la chaîne ⚠⇒ de substitution : s'il doit y figurer, il faut le protéger par un « \ ». En particulier, quand ♥ ⇒ on travaille sur des chemins UNIX, ce délimiteur est fortement déconseillé. Comparer par exemple les deux requêtes suivantes qui remplacent `/usr/local/bin` par `/usr/bin` :

```
s/\/usr\/local\/bin/\/usr\/bin/
s+/usr/local/bin+/usr/bin+
```

Exemples

`s/^/!/` permet d'insérer l'introducteur `!` de commentaires en tête de chaque ligne d'un fichier source fortran
`s/@*/+/g` permet de remplacer un nombre quelconque d'occurrences de `@` par le caractère `+`. Ne pas oublier que le multiplicateur `*` signifie aussi zéro occurrence, donc si `@` n'est présent nulle part, la substitution va introduire un `+` entre chaque lettre ainsi qu'en début et fin de ligne (*cf.* 6.5, p. 54).

8.2.2 Transcrire caractère par caractère

`y/jeu1/jeu2/` remplace le premier caractère du `jeu1` par le premier caractère du `jeu2`, puis le deuxième par le deuxième... comme avec `tr` (*cf.* 5.3, p. 44). Les deux jeux doivent posséder le même nombre de caractères, ceux du premier jeu doivent être tous différents, mais pas nécessairement ceux du second.
⚠⇒ Ne pas confondre avec la substitution de motifs ; ici le remplacement des caractères se fait un par un, aucun contexte n'étant pris en compte. L'intérêt de cette requête est que, contrairement à `tr`, elle peut être restreinte à certaines lignes du fichier et utilisée avec des caractères codés sur plusieurs octets, par exemple en UTF-8.

Exemple

`sed -e 'y/aeiouy/AEIOUY/'` passe en majuscule les voyelles
`sed -e '3,$y/01/10/'` échange les `1` et les `0` à partir de la ligne 3.

8.2.3 Supprimer des lignes

`ligne1,ligne2d` (**d**elete) supprime de la ligne `ligne1` à la ligne `ligne2` incluses
Exemples (les lignes non détruites sont affichées) :
`sed '1,12d' fic` imprime de la ligne 13 à la fin du fichier `fic`
`sed '12,$d' fic` imprime les lignes 1 à 11 du fichier `fic`
`sed '/^#/d' fic` imprime les lignes du fichier `fic` qui ne commencent pas par `#`

8.2.4 Imprimer des lignes

`ligne1,ligne2p` (**p**rint) imprime de la ligne `ligne1` à la ligne `ligne2` incluses

Exemples

On précise l'option `-n` pour ne pas tout afficher par défaut.
`sed -n '1,12p' fic` imprime les 12 premières lignes du fichier `fic`
`sed -n '12,$p' fic` imprime de la ligne 12 à la fin du fichier `fic`
`sed -n '/^#/p' fic` affiche les lignes du fichier `fic` qui commencent par `#`

8.2.5 Quitter

`ligne1 q` (**q**uit) termine `sed` à la ligne spécifiée, ce qui arrête la recopie des lignes.

Exemples

`sed '12q' fic` imprime les 12 premières lignes du fichier `fic`
`sed '/^#/q' fic` imprime les premières lignes du fichier `fic` jusqu'à la première ligne commençant par un « `#` »

8.3 Remarques importantes

1. Par défaut, `sed` agit sur toutes les lignes du fichier[3], mais il est possible de restreindre la portée d'une requête à un ensemble de lignes spécifiées par adressage explicite (`12` par ex.) ou grâce à la recherche d'un motif (`/^#/` par ex.) ; la requête peut aussi porter sur les lignes comprises dans un intervalle (fermé) (`ligne1,ligne2`, `/p6m1mni/,/^#/`).
2. Lorsqu'on utilise l'instruction `p` qui provoque l'affichage de lignes particulières, il faut empêcher l'affichage global du résultat de l'édition, grâce à l'option `-n` (sous peine de voir les lignes affectées par la requête `p` affichées deux fois).
3. Des expressions régulières peuvent être utilisées pour définir des motifs génériques (sélection de lignes ou requêtes de substitution).

   Dans les requêtes de substitution, la chaîne de caractères complète qui correspond à l'expression régulière recherchée peut être référencée dans le second membre de la substitution par le caractère spécial `&` ; il est aussi possible de découper les motifs recherchés en sous-expressions que l'on peut ensuite référencer dans le second membre de la substitution par `\1`, `\2` ... (*cf.* chapitre 6 sur les expressions régulières).

   Exemples :
   `sed -e 's/[0-9]/(&)/g'` permet d'entourer chaque chiffre de parenthèses.
   `sed -e 's/[0-9][0-9]*/(&)/g'` permet d'entourer chaque nombre entier de parenthèses.
   `sed -e 's/^\([0-9]\)\([0-9]\)/\2\1/'` permet d'échanger deux chiffres consécutifs en début de ligne
   `sed -e 's/\([0-9][0-9]*\)\.\([0-9]*\)/\1/g'` supprime la partie décimale des nombres

8.4 Fichier d'instructions

Un fichier d'instructions `sed` (ou sed-script) a pour structure générale :

```
action                              l'action est appliquée à toutes les lignes
ligne action                l'action est appliquée à la (les) ligne(s) désignée(s)
ligne1,ligne2 action              l'action est appliquée à l'intervalle de lignes
ligne1,ligne2{                 les actions sont appliquées à l'intervalle de lignes
                    action1
                    action2
                }
```

Dans un sed-script, il est possible d'insérer des commentaires, introduits par le caractère `#` : tout ce qui suit `#` (hors motif de recherche ou de substitution) jusqu'à la fin de la ligne est considéré comme un commentaire.

3. Au contraire, sous `ex`, une requête sans adresse portera sur la *ligne courante* (notion qui n'a pas de sens sous `sed`) et il faudra préciser l'adresssage `%` ou `1,$` pour que la requête s'applique à toutes les lignes du fichier.

Exemple

```
s/^#/!/
/p6m1mni/{
            s/p6m1mni/(&)/
            s/[0-9]/x/g
          }
```

♠ 8.5 Complément : notion d'espace de travail

Le filtre `sed` travaille ligne à ligne. Il peut être utile de regrouper les lignes entre elles lorsque le document est organisé en paragraphes ou en blocs autonomes, par exemple, les instructions en fortran (ou en langage C) qui s'étalent sur plusieurs lignes. Il est possible de procéder à un traitement « multilignes » avec `sed` en forçant la lecture de tout le bloc dans l'espace de travail avant de le traiter. Les coupures de lignes restent repérables dans l'espace de travail par l'expression `\n` (embedded new-line). On se reportera à des ouvrages spécialisés pour plus de détails (Dougherty et Robbins, 1999, par exemple).

Chapitre 9

Le filtre awk

`awk`[1] est un filtre de manipulation de fichiers texte. Son utilisation ressemble formellement à celle de `sed`. `awk` explore les fichiers ligne à ligne en découpant chaque ligne en ses différents champs. Il est donc très adapté pour un traitement des fichiers par colonnes et possède des fonctions de tableur. Bien au-delà, `awk` est un filtre programmable très puissant doté d'opérateurs de calcul, de nombreuses fonctions et de structures de contrôle dont la syntaxe est proche de celle utilisée dans le langage C.

9.1 Syntaxe générale

La syntaxe générale, formellement identique à celle de `sed`, est :

syntaxe

```
awk -f fichier_d_instructions.awk [fichier_de_donnees]
```

Les instructions écrites dans *`fichier_d_instructions`*`.awk` sont exécutées sur le contenu de *`fichier_de_donnees`*. Il est possible de remplacer *`fichier_de_données`* par une liste de fichiers. Si aucun fichier n'est indiqué, c'est l'entrée standard qui est prise par défaut. Sauf redirection de sortie[2], le résultat est envoyé vers la sortie standard.
Dans le cas d'instructions courtes, il est possible de saisir ces dernières en ligne. La syntaxe est alors :
`awk` *`instructions`* *`[fichier_de_données]`*
Comme les instructions comportent généralement des caractères interprétés par le shell (ne serait-ce que l'espace, mais aussi les `$` désignant les champs, *cf.* 11.2, p. 81), on entoure habituellement les instructions `awk` d'une protection forte « ' » sauf si on souhaite précisément que le shell les transforme avant que `awk` ne les interprète. ⇐ ♥

9.2 Programmation

9.2.1 Structure des données : enregistrement, champ

Dans le langage de `awk`, chaque ligne de `fichier` s'appelle un enregistrement (record). Dans un enregistrement, tout « blanc » (au sens de un ou plusieurs espaces ou tabulations) est un séparateur qui délimite un « champ » (field). L'option `-F "separ"` permet de redéfinir le séparateur[3] si nécessaire.
Lorsque `awk` lit une ligne (enregistrement), il l'éclate en ses différents champs :

`$0` représente toute la ligne,

`$1` représente le premier champ,

`$2` représente le deuxième champ,

....

`NF` (**n**umber of **f**ields) contient le nombre de champs de la ligne courante.

1. Le nom de cette commande est constitué des initiales de ses inventeurs : Aho, Weinberger et Kernighan, *cf.* Aho *et al.* (1995).
2. La redirection peut être requise au niveau du shell, mais elle peut aussi être stipulée dans le programme `awk`, (*cf.* 9.5.5, p. 71).
3. Il est aussi possible de redéfinir le séparateur de champs via la variable `FS`, *cf.* 9.2.3, p. 65.

`$NF` représente le dernier champ de la ligne courante,

`NR` (**number of record**) contient le numéro de la ligne courante.

⚠⇒ N.B. : ne pas confondre les variables désignant les champs sous `awk` avec les paramètres positionnels de l'interpréteur de commande (*cf.* 12.1.2, p. 87).

9.2.2 Principes de base

Programme `awk`

Un programme `awk` est une suite d'instructions. Une instruction est de la forme :
`sélecteur {action}`

`awk` lit chaque ligne du *`fichier_de_donnees`* et, pour chaque ligne évalue chaque sélecteur. Si le sélecteur est vrai, l'action est exécutée.

- si aucune action n'est précisée, la ligne est simplement recopiée sur la sortie (l'action par défaut est `print`, c'est-à-dire `print $0`)
- en l'absence de sélecteur, l'action est exécutée sur toutes les lignes de du fichier.

⚠⇒ Les lignes de données sont donc automatiquement parcourues, et les actions déclenchées par défaut pour chaque ligne, sans avoir à programmer explicitement une boucle sur les enregistrements. En revanche, le parcours des champs d'une ligne nécessite une boucle explicite du champ 1 au champ `NF`.
N.B. : comme sous `sed`, sur une ligne de programme `awk`, tout ce qui suit le symbole dièse « `#` », hors expression régulière ou chaîne de caractères, constitue un commentaire.

Sélecteurs sous `awk`

Un **sélecteur** restreint donc la portée des actions de `awk` aux seules lignes pour lesquelles il est vrai. Il peut prendre les formes suivantes :

- vide et alors l'action est appliquée à toutes les lignes ;
- une expression rationnelle étendue [4] (extended regular expression, ERE, *cf.* 6.6, p. 55) entre `/` et `/` : le sélecteur est vrai si le motif représenté par l'expression rationnelle est présent dans la ligne
- une condition logique telle que, par exemple :

`$i == `*`chaîne`*	le champ `i` coïncide avec la chaîne *`chaîne`*
`$i != `*`chaîne`*	le champ `i` est différent de la chaîne *`chaîne`*
`$i ~ /`*`ERE`*`/`	le champ `i` correspond à l'expression régulière étendue *`ERE`*
`$i !~ /`*`ERE`*`/`	le champ `i` ne correspond pas à l'expression régulière étendue *`ERE`*
`$i <= `*`valeur`*	le champ `i` (numérique) est inférieur ou égal à *`valeur`*
`NR > 10`	le numéro d'enregistrement (de ligne) est supérieur à 10
`NF == 3`	le nombre de champs est de 3

- une combinaison logique (via `&&`, `||` ou `!`) de sélecteurs des deux types précédents
- un intervalle de lignes sous la forme : `sélecteur1, sélecteur2`
- `BEGIN` ou `END` qui introduisent des actions exécutées avant ou après le traitement ligne à ligne des données d'entrée. Ces deux sélecteurs sont facultatifs [5].

Exemples de sélecteurs

`/motif/`	la ligne contient la chaîne `motif`
`$1 =="juin"`	le champ 1 est la chaîne `juin` (*cf.* 9.2.1, p. 63 pour la définition d'un champ)
`$2 == 333`	le champ 2 est égal à `333`

4. L'usage des intervalles dans les expressions rationnelles n'est pas autorisé par défaut, sauf avec l'option `--posix` ou l'option plus spécifique `--re-interval`

5. Ne pas croire qu'ils constituent des délimiteurs nécessaires du programme `awk`. D'ailleurs `END` est utilisé beaucoup plus souvent que `BEGIN`, car il n'est pas nécessaire d'initialiser les variables sous `awk`. Enfin, rien n'empêche de placer le sélecteur `END` en première instruction et `BEGIN` en dernière instruction !

`$3 ~ /^nom/`	le champ 3 commence par `nom`
`$2 != 333 || $3 !~ /^nom/`	le champ 2 n'est pas égal à `333` ou le champ 3 ne commence pas par `nom`

Action `awk`

Une **action** est une suite d'instructions séparées par des « `;` » qui incluent des affectations de variables, des calculs ou des opérations sur des chaînes de caractères. La syntaxe des actions est calquée sur celle du langage C, par exemple :

- les opérateurs de `awk` (`= + - * / % += -= *= /= ++ -- ... || && ...`)[6] ; l'espace sert d'opérateur de concaténation entre chaînes ;
- les contrôles de flux (`if`, `while`, `for`, `break`)
- les fonctions numériques et d'impression.

Les variables sous `awk` : Toutefois, les variables ne sont pas déclarées et leur typage dépend de leur première utilisation : numérique (initialisée à 0), chaîne de caractères (initialisée à la chaîne vide) ou tableau (*cf.* 9.4 p. 67). La référence à une variable se fait simplement à l'aide de son nom, contrairement à la syntaxe du shell qui utilise le « `$` »[7] ; par conséquent les constantes chaînes de caractères doivent être délimitées par des guillemets doubles « `"` ». `"mot"` désigne une chaîne de caractères constante, alors que `mot` désigne une variable `awk`, qui peut être numérique ou de type chaîne et dont la valeur peut être une chaîne. ⇐⚠

9.2.3 Variables spécifiques

Il existe bien d'autres variables internes à `awk`, et en particulier :

`FILENAME` contient le nom du fichier en cours de traitement (mais modifier cette variable ne modifie pas le fichier en entrée ; elle est positionnée automatiquement lors de la lecture des données ; ainsi, une instruction `nextfile`, *cf.* 9.5.3, p. 71, la modifie).

`FS` (**f**ield **s**eparator) est le séparateur de champs en entrée ; par défaut, c'est un blanc, sauf s'il a été modifié par l'option `-F` de awk ; la modification peut intervenir avant la lecture des données, sous le sélecteur `BEGIN` (ce qui équivaut à utiliser l'option `-F`), mais aussi en cours de lecture, par exemple : `'NR>10{FS = ":"}'` ;

`OFS` (**o**utput **f**ield **s**eparator) est le séparateur de champs en sortie (par défaut, un blanc).

`OFMT` (**o**utput **f**or**m**at for numbers) : `%g` (au sens de `printf` en C) par défaut.

`RS` (**r**ecord **s**eparator) est le séparateur d'enregistrement en entrée (par défaut, le changement de ligne)

`ORS` (**o**utput **r**ecord **s**eparator) est le séparateur d'enregistrement en sortie (changement de ligne par défaut).

9.2.4 Fonctions incorporées

Il existe dans `awk` des fonctions incorporées (internes), par exemple :

- mathématiques : `cos(expr)`, `sin(expr)`, `exp(expr)`, `log(expr)`, `int(expr)` ;
- sorties :
  `print` affiche l'enregistrement courant ;
  `print` *`expr1,expr2, ...exprn`* affiche les expresssions séparées par la chaîne `OFS` (espace par défaut) suivies d'un changement de ligne ;
  `printf("format",...)` et `sprintf("format",...)` analogues au C (les changements de ligne doivent être spécifiés par `"\n"` dans le format) ;

6. Le langage `awk` comprend l'opérateur `^` d'élévation à la puissance et l'opérateur composé `^=` qui ne sont pas disponibles en langage C.

7. Mais les contenus des champs `awk` sont désignés par `$i`, que ce soit pour leur affectation ou leur référence.

— chaînes de caractères :
`tolower(s)` / `toupper(s)` : passe la chaîne `s` en minuscules/majuscules ;
`index(s1, s2)` renvoie la position de la chaîne `s2` dans `s1` et 0 si `s2` n'apparaît pas dans `s1` ;
`length(s)` : nombre de caractères de la chaîne `s` ;
`substr(s, m, n)` sous-chaîne de `s` de `n` caractères commençant à la position `m`.

♠ 9.2.5 Internationalisation

Quand la commande `awk` prend en compte le contrôle local (*cf.* 15.4.1, p. 106), par exemple avec l'option `--posix`, le séparateur décimal passe de « `.` » en à « `,` » en français,
⚠⇒ comme le prouve l'exemple suivant :

— avec `LC_ALL=C` (ou seulement `LC_NUMERIC=C`), la commande
`echo '1.25' | awk '{print 2*$1, 2/3}'`
affiche `2.5 0.666667` avec des points

— alors qu'avec `LC_ALL=fr_FR` (ou seulement `LC_NUMERIC=fr_FR`), la commande
`echo '1,25' | awk '{print 2*$1, 2/3}'`
affiche `2,5 0,666667` avec des virgules

Avec un environnement francisé, les nombres décimaux écrits avec un point décimal sont alors tronqués : `echo '1.25' | awk '{print 2*$1, 2/3}'` affiche donc `2 0,666667`

Remarques La commande `gawk`, version par défaut sous linux, prenait en compte la variable d'environnement `LC_NUMERIC` jusqu'à la version 3.1.5. Mais ce comportement a causé tant de problèmes que, depuis la version 3.1.6 de `gawk`, ce n'est plus le cas sauf si on utilise l'option `--posix` ou, ce qui est plus spécifique, l'option `--use-lc-numeric`.
De plus, depuis la version 3.15 de `gawk`, les fonctions traitant les chaînes de caractères telles que `length`, `index` ou `substr` travaillent en termes de caractères et non d'octets,
⚠⇒ ce qui se révèle très différent pour les caractères accentués codés sur plusieurs octets en UTF-8 par exemple (*cf.* 4.1.3, p. 31).

9.3 Exemples

9.3.1 Exemples élémentaires de programmes `awk`

— `{print NR, $0}` affiche chaque ligne précédée de son numéro (comme `cat -n`)

— `END {print NR}` affiche le nombre de lignes du fichier (comme `wc -l`)

— `{print $1}` affiche le premier champ de chaque ligne

— `{s+=$1} END{print s}` affiche la somme de la première colonne (champ supposé numérique)

— `{s+=$1} END{print s/NR}` affiche la moyenne de la première colonne (champ supposé numérique)

— `{a=$1; $1=$2; $2=a; print $0}` affiche chaque ligne en échangeant les deux premiers champs

— `{s=0; for (i=1;i<=NF;i++){s+=$i}; print s}` affiche la somme des champs (supposés numériques) de chaque ligne (noter la remise à zéro de `s` à chaque ligne)

— `BEGIN {col = ""} {col = $1 " " col} END {print col}` affiche la première colonne transposée en ligne par concaténation de chaînes

9.3.2 Exemples appliqués à un fichier de données particulier

Soit le fichier de dépenses `donnees` suivant :

donnees

```
lundi       4   01   2002   10H Suzanne   30   restaurant
mardi       23  02   2003   12h Jean      40   carte-orange
mercredi    7   04   2001   16H Anne      10   cafe
jeudi       8   10   2004   9H  Anne      15   librairie
vendredi    14  05   2003   22h Jean      70   TGV
samedi      22  05   2004   21H Anne      20   cafeteria
dimanche    13  10   2002   10H Jean      30   disques
mardi       11  01   2004   13h Suzanne   14   cinema
lundi       27  03   2000   9h  Anne      10   bus
```

1. `awk '{print}' donnees`
   affiche tout le fichier
2. `awk '/2003/' donnees`,
   `awk '/2003/{print}' donnees`, mais aussi
   `awk '/2003/{print $0}' donnees`
   affichent chacune les lignes de l'année 2003

   ```
   mardi       23   02   2003   12h Jean      40   carte-orange
   vendredi    14   05   2003   22h Jean      70   TGV
   ```

3. `awk '{print "année", $4}' donnees`
   affiche

   ```
   année 2002
   année 2003
   année 2001
   année 2004
   année 2003
   année 2004
   année 2002
   année 2004
   année 2000
   ```

4. `awk 'END {print NR, "lignes"}' donnees`
   affiche

   ```
   9 lignes
   ```

5. `awk '$7 > 30' donnees`
   affiche

   ```
   mardi       23   02   2003   12h Jean      40   carte-orange
   vendredi    14   05   2003   22h Jean      70   TGV
   ```

6. `awk '$4 == 2004 {print $7, $6}' donnees | sort -k1,1n`
   (*cf.* chapitre 10 pour la signification du tube `|`) affiche

   ```
   14 Suzanne
   15 Anne
   20 Anne
   ```

♠ 9.4 Les tableaux associatifs sous `awk`

La notion de tableau est extrêmement large et souple dans `awk` ;

— comme les variables, les tableaux n'ont pas besoin d'être déclarés ;

— ils ne sont limités en taille que par l'espace mémoire disponible ;

— ils sont dynamiques : on peut ajouter ou supprimer des éléments ;

— ils peuvent être indicés par des nombres ou par des chaînes de caractères.

`tab[ind]=val` affecte la valeur `val` à l'élément d'indice `ind` du tableau `tab`.
N.B. : la fonction interne `split` permet de construire des tableaux de caractères : `split(s, a, c)` découpe la chaîne de caractères `s` selon le séparateur optionnel `c` en un tableau `a[1]`,...,`a[n]` de `n` éléments et renvoie le nombre `n`.

Exemples d'utilisation des tableaux `awk` sur le fichier `donnees`

indice numérique

```
{ # en commençant à l'indice 0
  jours[i++]=$1 # remplir le tableau avec les jours
}
END{ # après lecture des NR données, imprimer le tableau
  print "affiche les jours dans l'ordre des données"
  for (i=0; i<NR; i++) print i, jours[i]
}
```

affiche :

```
affiche les jours dans l'ordre des données
0 lundi
1 mardi
2 mercredi
3 jeudi
4 vendredi
5 samedi
6 dimanche
7 mardi
8 lundi
```

```
{ # en commençant à l'indice 0
  jours[i++]=$1 # remplir le tableau avec les jours
}
END{ # après lecture des NR données, imprimer le tableau
  print "autre syntaxe de boucle : noter l'ordre"
  for (i in jours ) print i, jours[i]
}
```

affiche :

```
autre syntaxe de boucle : noter l'ordre
4 vendredi
5 samedi
6 dimanche
7 mardi
8 lundi
0 lundi
1 mardi
2 mercredi
3 jeudi
```

```
BEGIN{ # construction d'un tableau des noms des mois de l'année
   mois = "janv fevr mars avr mai juin juil aout sept oct nov dec"
   n = split(mois,tab_mois," ")
   print "tableau des", n,  "mois"
```

```
   for (i = 1; i <= n; i++) print i, tab_mois[i]
   print  "\nfichier où le numéro du mois a été changé en son nom"
}
{ # remplacement du champ 3 (mois numérique) par le nom du mois
   lemois = tab_mois[$3+0] ; $3 = lemois ; print $0
}
```

affiche :

```
tableau des 12 mois
1 janv
2 fevr
3 mars
4 avr
5 mai
6 juin
7 juil
8 aout
9 sept
10 oct
11 nov
12 dec

fichier où le numéro du mois a été changé en son nom
lundi 4 janv 2002 10H Suzanne 30 restaurant
mardi 23 fevr 2003 12h Jean 40 carte-orange
mercredi 7 avr 2001 16H Anne 10 cafe
jeudi 8 oct 2004 9H Anne 15 librairie
vendredi 14 mai 2003 22h Jean 70 TGV
samedi 22 mai 2004 21H Anne 20 cafeteria
dimanche 13 oct 2002 10H Jean 30 disques
mardi 11 janv 2004 13h Suzanne 14 cinema
lundi 27 mars 2000 9h Anne 10 bus
```

indice chaîne de caractères (remarquer l'ordre en sortie)

```
{
    jours[$8]=$1
}
END{
    for (dette in jours) print dette, jours[dette]
}
```

affiche :

```
cafeteria samedi
restaurant lundi
bus lundi
disques dimanche
TGV vendredi
cinema mardi
cafe mercredi
librairie jeudi
carte-orange mardi
```

9.5 Compléments

`awk` est un utilitaire programmable dont ce chapitre ne donne qu'un aperçu très succint. Pour plus d'informations, on se reportera utilement à l'ouvrage Aho *et al.* (1995), écrit par les concepteurs de `awk`, ainsi que Robbins (1999) et la documentation en ligne `http://www.gnu.org/software/gawk/manual/` de la version `gawk` qui présente des fonctionalités étendues.

9.5.1 Transmission d'informations du shell à `awk`

Il est possible de transmettre des informations du shell vers le programme `awk` grâce à l'option `-v` selon la syntaxe :
`awk -f` *`fichier_d_instructions.awk`* `-v` *`var_awk=valeur fichier_de_données`*
par exemple : `awk -f pgm.awk -v qui=${USER} donnees`
permettra d'utiliser dans `pgm.awk` la variable `qui` contenant le nom de l'utilisateur donné par la variable d'environnement `USER` (*cf.* 11.1.4, p. 80).

♠ 9.5.2 La fonction `getline`

Une restriction fondamentale de `awk` est qu'il ne procède a priori qu'à **une seule lecture des données** et que toute opération nécessitant plusieurs passes sera difficile à programmer. Par exemple, un simple calcul de pourcentage en colonnes nécessite a priori deux lectures des données. Une solution souvent lourde peut consister à stocker les données dans un tableau associatif et à les manipuler ensuite sous le sélecteur `END`. Mais on peut aussi utiliser la fonction `getline` qui permet de déclencher la lecture d'un enregistrement dans un fichier... pour relire le fichier déjà lu par `awk` une fois la somme calculée.

Syntaxe de `getline`

`getline` lit une ligne sur l'entrée courante et l'affecte à `$0`
`getline < "-"` lit une ligne sur l'entrée standard et l'affecte à `$0`
`getline var` lit une ligne sur l'entrée courante et l'affecte à `var`
`getline var < "fic"` lit une ligne dans le fichier `fic` et l'affecte à `var`
`getline var < "-"` lit une ligne sur l'entrée standard et l'affecte à `var`
`"cmd_unix" | getline` lit la sortie standard de la commande UNIX `cmd_unix` via un tube

Remarques

- `getline` n'a pas d'argument mais retourne `-1` en cas d'erreur, `0` en cas de fin de fichier, et `1` si la lecture est correcte ;
- sauf s'il s'agit de variables `awk`, les noms de fichiers et les commandes UNIX doivent apparaître délimités par `"..."` comme toutes les chaînes de caractères ;
- si l'affectation se fait dans une variable, il n'y a pas de découpage en champs `$1`, `$2`, ... de la ligne lue.

Exemples

avec deux fichiers
`awk '{if( (getline x < "tab") > 0) print x; print $0}' donnees`
permet d'imprimer alternativement une ligne du fichier `tab` suivie d'une ligne du fichier `donnees`. L'impression s'arrête à l'épuisement du fichier `donnees`, mais si le fichier `tab` comporte moins de lignes, on termine en n'imprimant que celles de `donnees`

avec un fichier et l'entrée standard
`awk '{print $0;print "commenter"; getline x < "-"; print $0,x}'` affiche chaque ligne, demande de saisir un commentaire puis affiche la ligne commentée ; on pourra rediriger le dernier `print` dans un fichier.

avec un seul fichier d'entrée

`getline` force la lecture d'une nouvelle ligne du fichier dans l'instruction où il apparaît :
`awk '{print $0; if(getline == 1) print "paire"}'`
affiche les lignes de numéro impair et `paire` à la place des lignes de numéro pair.

♠ 9.5.3 Instructions de saut

L'instruction `next` permet de sauter à l'enregistrement suivant sans effectuer les actions qui suivent.
L'instruction `nextfile` permet de sauter à la lecture du fichier suivant de la liste, donc de terminer la lecture des données s'il s'agit du dernier. Elle est utile quand on traite plusieurs fichiers de données dans lesquels tous les enregistrements ne sont pas examinés.
Enfin, l'instruction `exit [`*`exit_code`*`]` permet de terminer le traitement sans lire la suite des données ni la suite des instructions, sauf celle sous le sélecteur `END`. L'argument optionnel `exit_code`, rendu au système comme code de retour (*cf.* 11.3, p. 84) de la commande `awk` peut être utilisé pour signaler les comportements exceptionnels. À cet argument près, l'instruction `exit` se comporte comme `nextfile` s'il n'y a qu'un fichier de données.

9.5.4 Format des sorties

L'instruction `printf "format", expression` permet comme en `C` d'afficher des expressions avec le format choisi. Celui-ci contient du texte et des descripteurs de format. Chaque descripteur est composé par le symbole `%` suivi par la taille et le type de la variable à afficher (`%5d` pour un entier de 5 chiffres, `%6.2f` pour un réel avec 2 décimales après la virgule, `%s` pour une chaîne de caractères). Alors que chaque `print` crée une nouvelle ligne, avec `printf`, il est nécessaire de forcer les retours à la ligne par `printf "\n"`. Mais `printf` laisse plus de latitude que `print`, en particulier il permet de concatener sur une même ligne des affichages issus de plusieurs enregistrements d'entrée.

9.5.5 Redirections de sortie

Le résultat des fonctions `print` et `printf` peut être redirigé dans un fichier ou dans un tube (voir chapitre 10 à propos des redirections).
Exemples :

1. `awk '{if ($1 ~ /^#/) print $0 >"comments"; else print $0}' script`
   écrit dans le fichier `comments` les lignes du fichier `script` commençant par `#` et affiche les autres lignes
2. `awk '{print $0 > $6}' donnees`
   éclate le fichier de dettes en un fichier par personne (nommé par le prénom)
3. `awk '{ print $0 | "sort -k"NF","NF"f"}' donnees`
   affiche les dettes classées par ordre alphabétique du dernier champ sans tenir compte de la casse (noter qu'ici `NF` est à l'extérieur de la chaîne de caractères donc interprété par `awk` et représente le numéro du dernier champ)

```
lundi       27  03  2000  9h  Anne    10  bus
mercredi    7   04  2001  16H Anne    10  cafe
samedi      22  05  2004  21H Anne    20  cafeteria
mardi       23  02  2003  12h Jean    40  carte-orange
mardi       11  01  2004  13h Suzanne 14  cinema
dimanche    13  10  2002  10H Jean    30  disques
jeudi       8   10  2004  9H  Anne    15  librairie
lundi       4   01  2002  10H Suzanne 30  restaurant
vendredi    14  05  2003  22h Jean    70  TGV
```

Chapitre 10

Processus, redirections et tubes

10.1 Flux standard

Une commande UNIX manipule trois flux standard de données, auxquels sont associés des entiers appelés descripteurs :

0 : l'entrée standard
(par défaut le clavier)

1 : la sortie standard
(par défaut l'écran)

2 : la sortie d'erreur standard
(par défaut l'écran)

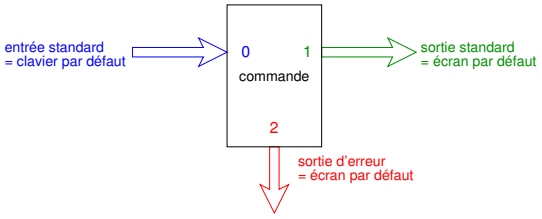


FIGURE 10.1 – Flux standard de données associés à une commande

10.2 Redirections

On peut rediriger ces flux vers des fichiers ou les entrées/sorties d'autres commandes. Ces redirections constituent une des souplesses essentielles du système UNIX car elles permettent de combiner des commandes de base pour effectuer des traitements complexes.

10.2.1 Redirection de sortie vers un fichier (> et >>)

syntaxe

```
commande > fichier
```

Pour ajouter le résultat à la fin du fichier :

syntaxe

```
commande >> fichier
```

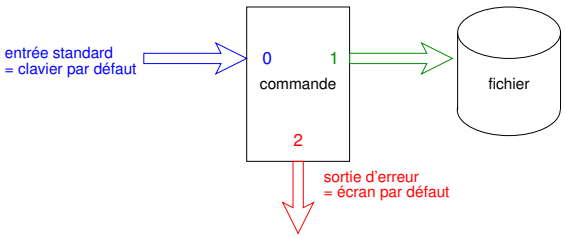


FIGURE 10.2 – Redirection de la sortie

Exemples :

```
ls -l > liste
date > resultats
echo fin >> resultats
```

⚠⇒ **Attention** : une redirection de la sortie standard d'un filtre vers son fichier d'entrée vide le fichier, par exemple : `cat fic1 > fic1` efface le contenu du fichier `fic1`...

10.2.2 Redirection de l'entrée depuis un fichier (<)

syntaxe

commande < *fichier*

Exemple : lecture des données d'entrée d'un exécutable sur un fichier au lieu de la saisie au clavier

```
a.out < entrees
```

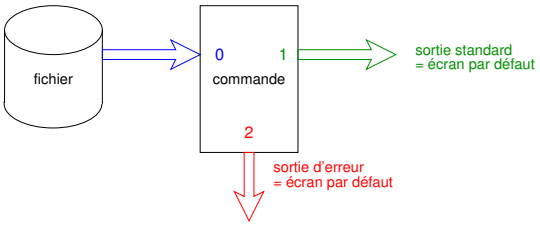


FIGURE 10.3 – Redirection de l'entrée

Remarque : Dans certaines commandes qui admettent en argument une liste de fichiers, l'entrée standard peut être désignée par le symbole « - ». Par exemple,
`sort fic1 | cat pre - post`
affiche le fichier `fic1` trié, précédé du fichier `pre` et suivi du fichier `post`

10.2.3 Redirection de la sortie d'erreurs vers un fichier (2> et 2>>)

syntaxe

commande 2> *fichier*

Pour ajouter les erreurs à la fin du fichier :

syntaxe

commande 2>> *fichier*

Attention : pas d'espace entre 2 et >
Exemple : stockage des diagnostics d'une compilation dans un fichier pour éviter le défilement à l'écran (il faudra corriger d'abord la première erreur)

```
gfortran essai.f90 2> erreurs
```

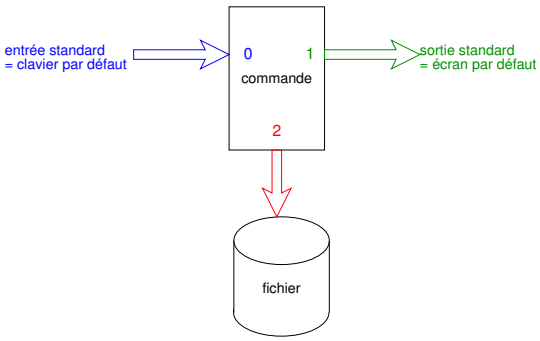


FIGURE 10.4 – Redirection de l'erreur

10.2.4 Redirection de l'erreur standard vers la sortie standard (2>&1)

La sortie d'erreur standard peut être regroupée dans le même flux que la sortie standard grâce à la syntaxe `commande 2>&1`
Exemple (on suppose que `/etc/motd` est accessible en lecture) :

```
cat /etc/motd /fichier_inexistant    affiche le mot du jour et un message d'erreur
cat /etc/motd /fichier_inexistant > resultat        affiche un message d'erreur
cat /etc/motd /fichier_inexistant > resultat 2>&1          n'affiche plus rien
```

Noter que la redirection de la sortie standard dans la dernière commande doit *précéder* la redirection de l'erreur standard vers le flux de la sortie standard. ⇐⚠

10.3 Tubes ou pipes (|)

Pour appliquer deux traitements successifs à un flux de données, au lieu d'utiliser un fichier temporaire intermédiaire, il est plus efficace de connecter directement la sortie de la première commande à l'entrée de la deuxième, constituant ainsi un tube (pipe). Le

traitement est plus rapide car la seconde commande commence à s'exécuter dès que les premières sorties de la première sont disponibles, sans passer par le disque.

La méthode séquentielle avec fichier intermédiaire :

```
commande_1 > fichier
commande_2 < fichier
rm fichier
```

peut être simplifiée en :

syntaxe

```
commande_1 | commande_2
```

Exemple : affichage paginé des fichiers du répertoire courant

```
ls -l | more
```

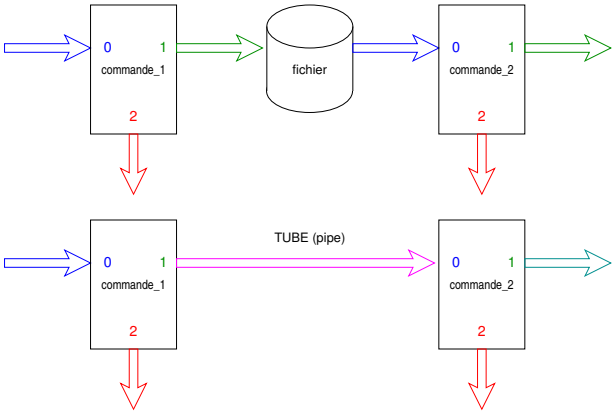


FIGURE 10.5 – Tube ou pipe

10.4 Gestion des processus

10.4.1 Généralités : la commande `ps`

Un processus est une tâche élémentaire identifiée sur le calculateur par un numéro unique ou pid (**p**rocess **id**entifier). On peut afficher la liste des processus avec la commande `ps`. Cette commande est déclinée suivant les systèmes selon trois syntaxes liées à l'historique d'UNIX, syntaxes qu'il est prudent de contrôler avec `man`. Certains systèmes acceptent partiellement les deux syntaxes historiques System V et BSD, mais la syntaxe POSIX, actuellement en cours d'implémentation doit être préférée.
Principales options de sélection des processus et des informations affichées pour chaque processus selon la syntaxe POSIX :

`-e` affiche tous les processus de tous les utilisateurs

`-U` *`liste_d_utilisateurs`* sélectionne les processus appartenant à cette liste d'utilisateurs

`-f` (**f**ull) affiche une liste complète d'informations sur chaque processus

`-o` *`options_de_sortie`* permet de spécifier les informations affichées et leur format

Principaux champs affichées par `ps` :

UID	PID	PPID	TTY	VSZ	CMD
n° d'utilisateur	n° du processus	n° du père	terminal	taille	commande

Exemples (sous `aix`)

la commande `ps` affiche

```
    PID    TTY  TIME CMD
1212592  pts/2  0:00 ps
1294516  pts/2  0:01 -ksh
```

Seuls les processus attachés au terminal courant [1] sont affichés.
N.-B. : la commande `ps` se voit agir.

la commande `ps -U p6m1mni` affiche

```
    UID     PID    TTY  TIME CMD
  40369  307400      -  0:02 sshd
```

1. La commande `tty` affiche le nom du terminal courant, ici `pts/2`.

```
  40369 1212590  pts/2  0:00 ps
  40369 1294516  pts/2  0:01 ksh
```

Quand on précise le nom de l'utilisateur, tous ses processus sont affichés, y compris ceux non attachés à un terminal.

la commande `ps -f` (**f**ull) affiche

```
     UID     PID    PPID   C    STIME    TTY  TIME CMD
 p6m1mni 1294516  307400   0 00:23:53  pts/2  0:01 -ksh
 p6m1mni 2027692 1294516  45 00:59:00  pts/2  0:00 ps -f
```

En précisant les champs et éventuellement leur intitulé :

la commande `ps -o uid=ID,tty=term,pid,ppid,vsz,args=commande` affiche

```
     ID   term     PID    PPID   VSZ commande
  40369  pts/2 1294516  307400  1280 -ksh
  40369  pts/2 2027768 1294516   532 ps -o uid=ID,tty=term,pid,ppid,vsz,args=
```

10.4.2 Caractères de contrôle et signaux

Les caractères de contrôle (obtenus par la frappe simultanée de la touche `Ctrl`, et d'un caractère, par exemple `Ctrl C`, noté `^C`) sont des caractères spéciaux la plupart du temps interprétés par le shell.

`^?` ou `^H`	`erase`	effacement du dernier caractère
`^L`	`erase`	effacement de l'écran
`^S`	`stop`	blocage de l'affichage à l'écran
`^Q`	`start`	déblocage de l'affichage à l'écran
`^D`	`eof`	fermeture du flux d'entrée
`^C`	`intr`	interruption du processus
`^\`	`quit`	interruption du processus avec production d'une image mémoire (fichier binaire `core`)
`^Z`	`susp`	suspension du processus en cours
`^W`	`werase`	effacement du premier mot à gauche du curseur

L'affectation des caractères de contrôle à certaines fonctions dépend des installations et est gérée par la commande `stty` (`stty -a` affiche l'affectation courante).
Un caractère de contrôle ne peut agir que sur le processus en interaction avec le terminal ; pour intervenir sur un autre processus, il faut lui envoyer un message appelé *signal*. Pour envoyer un signal à un processus quelconque, on utilise la commande `kill`, avec comme paramètre le numéro du processus (PID). `kill` envoie par défaut un signal de terminaison, mais si le processus ne s'interrompt pas, on peut être plus sévère avec `kill -KILL` (`kill -9`).

10.4.3 Processus en arrière plan

Comme le système UNIX est multi-tâche, il est possible de lancer une commande longue à exécuter en *arrière-plan* (background) et de « garder la main » pour d'autres commandes pendant que cette tâche de fond se déroule de façon asynchrone. Il suffit d'ajouter une esperluette (`&`) à la fin de la ligne de commande.

syntaxe

```
commande &
```

Gestion des processus :

— `jobs` affiche la liste numérotée des processus en arrière-plan(`jobs -l` affiche aussi les numéros de processus associés (`pid`).

— `fg` (**fore**ground) passe le job courant en premier plan
— `bg` (**b**ack**g**round) passe le job courant en arrière-plan

```
-] jobs -l
[1]- 26001 Running                 xterm &
[2]+ 26005 Running                 xclock &
```

En utilisant la syntaxe `%` suivi du numéro du job (plus simple que le numéro de processus ou `pid`), on peut indiquer le processus concerné aux commandes `fg`, `bg` et `kill` ; par exemple `kill %2` va arrêter `xclock`.

Exemple

— Lancer `xterm`, on « perd la main » dans la fenêtre initiale jusqu'à la fin du processus `xterm` lancé en premier-plan. Terminer normalement le processus `xterm` par exemple en tapant `exit` ou `^D` (la fermeture du flux d'entrée du shell termine le processus) dans la nouvelle fenêtre. On retrouve la main dans la fenêtre initiale.
— Lancer `xterm` dans la fenêtre initiale en premier plan. On peut interrompre le processus `xterm` par `^C` et retrouver la main dans la fenêtre initiale.
— Lancer `xterm &` en arrière plan : on conserve la main dans la fenêtre initiale. Les deux fenêtres peuvent être utilisées de façon asynchrone. Depuis la fenêtre initiale, on peut interrompre le processus `xterm` en arrière plan par `kill` suivi de son `pid`.

Un processus en arrière-plan ne peut plus (en général) recevoir de données saisies au clavier (s'il en attend, le processus se bloque), et il est prudent de ne pas le laisser
♥ ⇒ afficher des résultats volumineux à l'écran pour éviter des interférences avec l'affichage de premier-plan : en général, on redirige les entrées et sorties d'une commande avant de la lancer en arrière-plan [2].
♥ ⇒ N.-B. : si on a lancé par erreur une tâche longue en premier plan, on peut la suspendre avec `^Z`, puis la relancer en arrière-plan par `bg`, afin de reprendre la main pendant qu'elle s'exécute.

10.5 Compléments

10.5.1 Les fichiers spéciaux : exemple `/dev/null`

Le répertoire `/dev` contient des *fichiers spéciaux* gérant des flux de données entre le calculateur et les périphériques (devices) : terminaux, imprimantes, disques, ... La commande `tty` permet de connaître le fichier spécial particulier (ils sont numérotés) attribué à un terminal. On peut ainsi par exemple rediriger vers le pseudo terminal 3 la sortie d'erreur standard d'une commande lancée dans le terminal 2 (*`commande`* `2> /dev/pts/3`).
Mais le fichier spécial `/dev/tty` désigne de façon générique le terminal attaché à la connexion.
Parmi les fichiers spéciaux, `/dev/null` peut être utilisé comme une « poubelle » pour se débarrasser de certaines sorties inutiles.

syntaxe

```
commande 2> /dev/null
```

empêche le flux d'erreur de s'afficher à l'écran.

Exemple

`find` *`rep`* `-name "`*`nom`*`" -print 2> /dev/null`
permet d'éviter l'affichage des messages d'erreur, notamment ceux émis [3] quand on tente d'accéder à des fichiers non autorisés.

2. On préfère souvent garder la sortie d'erreur à l'écran pour surveiller le bon déroulement du processus en arrière-plan.
3. Attention, tous les messages d'erreur sont perdus, y compris ceux qui pourraient signaler une erreur de syntaxe dans l'utilisation de `find`.

10.5.2 Duplication de flux : `tee`

La commande `tee` (T en anglais) duplique le flux de son entrée standard sur le fichier passé en argument et sur sa sortie standard. Par exemple, `tee` permet de conserver une trace du résultat intermédiaire dans un tube :

```
cmd_1 | tee f_intermediaire | cmd_2
```

♠ 10.5.3 Notion de document joint (<<)

Il est possible de rediriger l'entrée d'une commande habituellement interactive pour lui passer des requêtes depuis une procédure. Il suffit de faire suivre la commande par `<<` et un mot délimiteur quelconque (sans espace) ; le texte qui suit, jusqu'à la première ligne (exclue) qui *commence par le délimiteur*, est redirigé [4] vers l'entrée de la commande.

Exemple

```
vi journal > /dev/null <<EOF          édition du fichier journal
:r /etc/motd                          lecture du fichier /etc/motd
8dd                                   destruction des 8 premières lignes
:w                                    sauvegarde
:q                                    sortie de vi
EOF                                   fin de redirection
```

10.5.4 Groupement de commandes

On rappelle que pour écrire plusieurs commandes UNIX sur une même ligne, il suffit de les séparer par un point-virgule `;`

```
cd ; pwd ; ls
```

Il est possible de grouper plusieurs commandes grâce aux parenthèses ; elles s'exécutent alors dans un sous-shell, fils du shell interactif. Au retour dans le shell père, il n'y a aura pas héritage de l'environnement modifié dans le fils. ⇐⚠

```
cd /tmp ; pwd                         affiche /tmp
(cd /bin ; pwd)                       affiche /bin dans le sous-shell
pwd                                   affiche de nouveau /tmp
```

Remarquer la différence avec une exécution séquentielle normale en cas de redirection.

```
date ; hostname > memo1
(date ; hostname) > memo2
```

Dans le premier cas, la date s'affiche à l'entrée et `memo1` contient `ibm1`, alors que rien ne s'affiche dans le deuxième et que `memo2` contient les deux lignes :

```
Wed Oct 12 22:44:05 DST 2005
ibm1
```

De manière plus générale, lorsque plusieurs commandes sont regroupées en blocs par des parenthèses (sous-shell), des accolades (fonction anonyme, *cf.* 15.1.3, p. 101), ou une structure itérative (`do...done`, *cf.* 14.3.5, p. 100), les redirections extérieures au bloc portent sur l'ensemble du bloc de commandes.

4. Le shell effectue les substitutions de variables et de commandes sur le texte redirigé, sauf si le mot délimiteur du début du document joint est protégé par le caractère de protection \, des `'...'` ou des `"..."`. En `bash` et `ksh`, si le mot de début est protégé, celui de fin ne doit pas l'être ; en `csh` ou `tcsh`, les deux mots délimitant le texte du document joint doivent être protégés de façon identique.

10.5.5 Processus détaché

Une tâche très longue, sans interaction avec un terminal, peut être lancée comme processus *détaché* dont l'exécution pourra continuer après la fin de session interactive.

`nohup` *`commande`* `<entrees >sortie 2>erreurs &`

Par défaut la commande `nohup` redirige ses sorties standard et d'erreur vers le fichier `nohup.out` dans le répertoire courant. Le processus lancé avec `nohup` ne sera plus sensible au signal `SIGHUP` produit par la fermeture de la session [5].

Il existe d'autres modes d'exécution des processus qui font appel au service `cron` :

— détaché et *différé* (faire précéder la commande de `at`, suivi de la date souhaitée [6] pour le lancement, ou `batch` et alors c'est le système qui gère les priorités d'exécution),

— processus *périodiques* (voir la commande `crontab`) essentiellement pour la gestion du système (sauvegardes, purges d'espaces temporaires, mise à jour de bases d'indexation pour les manuels et les commandes, ...).

5. Il est possible de « déplacer » le terminal d'interaction d'une session, à condition de gérer la session interactive via la commande `screen`. On peut alors la détacher du terminal sans l'interrompre, puis la reprendre depuis un autre poste via `screen -r`

6. L'instant fourni à la commande `at` peut être une date absolue ou relative : la commande accepte par exemple `now +5hours` ou `tomorrow`. Il est de même pour la commande `date` qui permet d'afficher une date quelconque, grâce à l'option `-d` dont l'argument peut être une date absolue ou un décalage par rapport à la date courante. En particulier, `date -d +2days` affiche la date telle que `date` la donnera dans 2 jours.

Chapitre 11

Variables et métacaractères

11.1 Variables

11.1.1 Affectation et substitution

Le shell manipule des *variables* non typées et qui n'ont pas besoin d'être déclarées. Elles permettent de stocker des chaînes de caractères.
Syntaxe d'affectation [1] (en shell de type Bourne) :

syntaxe

variable=valeur (sans espace autour du signe =)

Référence à la valeur [2] de la variable : ⇐♥

syntaxe

$*variable* ou, plus précisément, ${*variable*}

Ne pas confondre la syntaxe du shell, qui utilise le $ uniquement pour la référence à la valeur, avec celle de `perl` ou `php` qui utilisent le $ pour désigner une variable scalaire. ⇐⚠

```
a=toto
b=25
c=3b
echo variables : a b c,  valeurs : ${a} ${b} ${c}
d=x+${a}
echo a=${a} d=${d}
```

N.-B. : pour effectuer des opérations arithmétiques sur les variables, on utilisera la commande `expr` qui interprète les valeurs des variables comme des entiers (*cf.* chapitre 13), par exemple `expr 12 + 2` fera l'addition attendue.
La commande interne `set` permet d'afficher la liste de toutes les variables du shell, sous la forme *variable=valeur*, à raison d'une variable par ligne.

```
set | more
set | wc -l                    pour compter le nombre de variables
```

11.1.2 La commande `read`

La commande interne (built-in) `read` permet aussi l'affectation de valeurs à des variables.

syntaxe

`read` *liste de variables*
liste de valeurs (saisies au clavier sur une seule ligne)

`read` lit une ligne sur l'entrée standard et affecte chaque mot de cette ligne à une variable, de gauche à droite. Si le nombre de mots est supérieur au nombre de variables, la dernière variable se voit affecter la chaîne constituée des mots restants de la ligne.

1. La convention utilisée par le shell est intermédiaire entre celles du C et du fortran, où le nom de la variable est utilisé sans signe $ (`var2=var1`) et celle de langages tels que `perl` et `php` pour lesquels il est toujours précédé d'un $ (`$var2=$var1`), que ce soit en référence ou en affectation.

2. L'emploi des accolades pour délimiter le nom de la variable est conseillé et parfois nécessaire lorsque la valeur de la variable est suivie d'une chaîne de caractères : `$var2` désigne `${var2}`, le contenu de la variable `var2` et non `${var}2`, la chaîne obtenue en concaténant le contenu de `var` et le caractère 2.

```
read aa bb                                          (commande lancée)
A BB CCC DDDD                                       (saisie de l'utilisateur)
echo ${aa}
A                                                   (réponse du shell)
echo ${bb}
BB CCC DDDD                                         (réponse du shell)
```

11.1.3 Portée des variables

⚠⇒ Par défaut, les variables ne sont pas héritées par les processus fils, comme on peut le constater en testant ce qui suit :

```
a=bonjour                                          affectation dans le processus père
echo a contient +${a}+            ⇒ a contient +bonjour+
bash                                                lancement d'un shell fils
    ps
    echo a contient +${a}+            ⇒ a contient ++          (non affecté)
    a=salut
    echo a contient +${a}+            ⇒ a contient +salut+
    exit                                                retour au shell père
ps
echo a contient +${a}+            ⇒ a contient +bonjour+          inchangé
```

11.1.4 Variables d'environnement

Mais certaines variables vitales, dites *variables d'environnement* et généralement désignées par des noms en majuscules, sont systématiquement héritées par les processus fils (à l'inverse, un processus fils ne peut pas affecter une variable du processus père). La commande interne `env` permet d'afficher la liste des variables d'environnement et leur contenu sous le même format que `set` pour les variables normales.

```
env | more
env | wc -l
echo TERM = ${TERM} , HOME = ${HOME}
echo PATH = ${PATH} , USER = ${USER}
```

Variables d'environnement standard :

`SHELL` : interpréteur de commandes ou shell (`bash`, `ksh`, `tcsh`, ...)

`TERM` : type de terminal utilisé (`vt100`, `xterm`, ...)

`HOME` : répertoire d'accueil

`PATH` : liste des chemins absolus de recherche des commandes séparés par des « `:` »

`USER` : nom de l'utilisateur

Pour forcer l'héritage d'une variable par les processus fils, on doit explicitement le spécifier dans le processus père par :

syntaxe

```
export variable
```

En `bash`, `export -p` affiche la liste de variables exportées et leurs valeurs.

♠ 11.1.5 Complément : valeur par défaut

Le mécanisme de substitution des variables admet de nombreuses constructions plus sophistiquées à l'intérieur des accolades, notamment pour fournir des valeurs par défaut suivant la syntaxe `${variable-valeur}` qui donne :

— la valeur de `${variable}` si cette variable est définie

— *valeur* sinon

Exemple : `${TMPDIR-/tmp}`
La variante `${variable:-valeur}` rend aussi la valeur par défaut lorsque la variable est définie mais vide.

♠ 11.1.6 Édition du contenu des variables

La syntaxe `${variable}` qui permet de référencer le contenu d'une variable admet de nombreuses extensions permettant de manipuler la chaîne de caractère résultante, notamment :

— `${#var}` qui désigne le nombre de caractères du contenu de la variable *var*

— `${var:n}` qui désigne la sous-chaîne commençant *après* le caractère numéro `n`

Exemple : si on affecte `fr_FR.UTF-8` à la variable de langue `LC_ALL`, la commande `echo ${LC_ALL:6}` affiche à partir du septième caractère de la variable, c'est-à-dire la partie codage soit ici `UTF-8`.

— `${var:n:lg}` qui désigne la sous-chaîne de longueur `lg` commençant après le caractère numéro `n`

— `${var//motif1/motif2}` qui remplace toutes les occurrences de *motif1* par *motif2* dans le contenu de la variable *var*

Exemple : `${PATH//:/ }` permet de transformer la liste des chemins d'accès aux commandes en bash ou ksh (donc séparés par des `:`) par une liste conforme à la syntaxe csh où le séparateur est l'espace.
Si `${PATH}` vaut `/usr/bin:/usr/local/bin:/usr/bin/X11`
`${PATH//:/ }` vaudra `/usr/bin /usr/local/bin /usr/bin/X11`

11.2 Caractères spéciaux

Nous avons déjà vu que le shell *interprète* certains caractères spéciaux aussi appelés métacaractères (metacharacters), en particulier les espaces pour découper la ligne de commande en mots, `>`, `>` et `|` pour les redirections, ainsi que `$` pour référencer les variables. Mais nous allons voir que ces caractères spéciaux sont en fait très nombreux.

11.2.1 Caractères générant des noms de fichiers

Caractères « jokers » [3] ou constructeurs de modèles de noms de fichiers :

`*`	une chaîne quelconque dans un nom de fichier
`?`	un caractère quelconque dans un nom de fichier
`[...]`	*un* caractère quelconque pris dans la liste entourée par les crochets (à l'intérieur des `[ ]`, c_1`-`c_2 représente la liste des caractères entre c_1 et c_2 dans l'ordre lexicographique)
`[!...]`	*un* caractère quelconque pris *hors* de la liste entourée par les crochets
`{motif1,motif2}`	motif1 ou motif2 (alternative) NB : sans espace après la virgule

3. Il est possible d'inhiber l'expansion des caractères « jokers » (ou wildcards) pour désigner des fichiers dont le nom comporte un ou plusieurs d'entre-eux avec l'option `noglob` du shell : `set -o noglob`. La commande `set +o noglob` réautorise l'expansion (*cf.* 15.1.7, p. 103).

Exemples :

`*`	tous les fichiers du répertoire courant
`*.f90`	tous les fichiers dont le nom finit par `.f90`
`*.*`	tous les fichiers dont le nom comporte un point
`data??`	tous les fichiers dont le nom est `data` suivi de deux caractères
`f.[abc]`	les fichiers `f.a`, `f.b`, et `f.c` s'ils existent
`f.[0-9]`	les fichiers dont le nom s'écrit `f.` suivi d'un chiffre
`f.[!0-9]`	les fichiers dont le nom s'écrit `f.` suivi d'un caractère qui ne soit pas un chiffre
`f.200[1-5]`	les fichiers dont le nom s'écrit `f.` suivi d'un nombre compris entre `2001` et `2005` (ne pas utiliser `[2001-2005]` qui signifie un seul chiffre parmi `0`, `1`, `2` ou `5` et où l'intervalle réduit à `1-2`)
`*.{for,f90}`	les fichiers d'extension `.for` ou `.f90`

⚠⇒ Par défaut (mais ce comportement est modifiable via une option du shell, *cf.* 15.1.7, p. 103), on ignore les noms des fichiers cachés, c'est à dire ceux dont le nom commence par un point.

⚠⇒ Par défaut, si aucun fichier du répertoire de travail ne correspond au motif générique utilisant ces caractères jokers, le shell les restitue inchangés (*cf.* 14.2, p. 96). Par exemple, dans un répertoire sans fichier source en C, la commande `ls *.c` affiche `*.c`. Pour obtenir dans ce cas une réponse sous forme de liste vide, on utilise la commande interne `shopt` pour positionner l'option `nullglob` du shell : `shopt -s nullglob`.

11.2.2 Liste des métacaractères

␣, TAB	espace et tabulation : séparateur d'entrée (IFS)
`*`, `?`, `[...]`	constructeurs de noms de fichiers
`~`	répertoire d'accueil
`<`, `<<`, `>`, `>>`, `|`	redirections
`$` ou `${...}`	évaluation de variable
`$(...)` ou `` `...` ``	substitution de commande
`;`	séparation de commandes (sur une même ligne)
`:`	commande « nulle »
`(...)`	groupement de commandes dans un sous-shell
`{...}`	groupement de commandes (sans sous-shell)
`&`	lancement en arrière plan
`!`	non logique
`||` et `&&`	association de commandes
`#`	introducteur de commentaire en shell
`\`	protection individuelle de caractère
`'...'`	protection forte
`"..."`	protection faible

11.2.3 Substitution de commande

Le résultat d'une commande (sa sortie standard) peut être affecté à une chaîne de caractères pour être stocké dans une variable ou repris par une autre commande.

syntaxe

```
$(commande)
```

⚠⇒ Noter l'ancienne syntaxe, avec des accents graves (backquotes) `` `commande` ``, moins lisible, en particulier dans le cas d'imbrications. Ne pas confondre `${variable}` et `$(commande)`.

La substitution de commande est utilisée pour paramétrer les shell-scripts, pour effectuer des calculs sur les entiers avec la commande `expr`, et, par exemple, pour manipuler des noms de fichiers et des chemins avec les commandes `basename` et `dirname`.

```
qui=$(whoami) ; echo $qui ; echo ${qui} # affectation de la variable qui
echo $(whoami)                          # utilisation directe
echo la date est $(date)
echo la date est `date`                 # ancienne syntaxe (Bourne shell)
#
s1=$(expr 12 + 2)                       # calcul puis affectation à s1
echo la somme de 12 et 2 est ${s1}
echo la somme de 12, 2 et 1 est $(expr $(expr 12 + 2) + 1) # imbrication
#
sans_suffixe=$(basename source.for .for) # suppression du suffixe .for
source90=$(basename source.for .for).f90 # changement du suffixe en .f90
echo ${sans_suffixe} ${source90}
```

11.2.4 Protections des métacaractères

Pour rendre aux caractères spéciaux leur valeur littérale et les protéger de l'interprétation par le shell, trois méthodes sont disponibles :

— la contre-oblique (backslash) \ interdit l'interprétation du caractère qui suit :

```
echo liste des fichiers : *
echo caractère \*
echo \$TERM = ${TERM}
echo texte; date
echo texte\; date
```

— un couple d'apostrophes (quotes) autour de la chaîne constitue une *protection forte*, interdisant toute interprétation par le shell à l'intérieur de la chaîne protégée :

```
echo liste des fichiers : *
echo 'caractère *'
echo '$TERM = ${TERM}'
echo texte; date
echo 'texte; date'
echo 'la date est $(date)'
```

— parfois, on souhaite laisser le shell effectuer les substitutions de variables et de commandes à l'intérieur de la chaîne et on préfère une *protection faible* à l'aide d'un couple de doubles apostrophes (double quotes) :

```
echo liste des fichiers : *
echo "caractère *"
echo "$TERM = ${TERM}"
echo texte; date
echo "texte; date"
echo "la date est $(date)"
```

N. B. : à l'intérieur des doubles apostrophes, les simples ne sont pas interprétées (et réciproquement). Les contre-obliques sont interprétées à l'intérieur des doubles apostrophes, mais pas à l'intérieur des simples. ⇐⚠

11.2.5 Exemples

Exemple 1

Comme le shell découpe la ligne de commande en « mots », il n'est pas possible de considérer une chaîne de caractères comportant des blancs comme un mot, par exemple pour l'affecter à une variable (`NAME=nom prenom` cherche à lancer la commande `prenom`, après avoir affecté `nom` à `NAME`). Pour protéger le blanc (un ou plusieurs espaces ou tabulations) de l'interprétation par le shell, trois méthodes sont disponibles :

— `NAME=nom\ prenom`, mais il en faut autant que d'espaces à protéger
`NAME=nom\ \ \ prenom`

— `NAME='nom prenom'`

— `NAME="nom prenom"`
qui va permettre de substituer le nom d'utilisateur dans : `NAME="$USER prenom"`

De la même façon, comme l'interprétation par le shell précède toujours celle de la commande, les espaces multiples non protégés entre mots seront fusionnés en un seul espace par le shell. Cette protection est en particulier nécessaire pour que la commande `echo` respecte le nombre des espaces entre les mots du message. Par exemple, la commande
`echo mot1 mot2` affichera simplement : `mot1 mot2`
alors que la commande
`echo "mot1 mot2"` affichera : `mot1 mot2`

Exemple 2

Il est possible d'affecter une valeur à une variable par simple concaténation de chaînes qui utilisent d'autres variables ou le résultat de commandes : `var="${var1}$(cmd)"`. À partir de commandes élémentaires d'identification, on peut construire une variable `adresse` contenant l'adresse de messagerie sur un serveur.

```
adresse=$(whoami)@$(hostname)
```

Exemple 3

En choisissant les options de la commande `date`, on peut afficher seulement l'année, le mois et jour sous le même format que celui obtenu en listant des fichiers avec `ls -l` (*cf.* 2.3.1, p. 9). Cette date peut être utilisée pour sélectionner via `grep` les seuls fichiers du jour dans une liste de fichiers :

```
ls -al | grep "$(date +'%%Y-%m-%d')"
```

En fait, pour rechercher des fichiers selon un critère de date, la commande `find` (*cf.* 3.2.1, p. 21) est plus puissante.

Exemple 4

```
find ~lefrere -name '*.f90' -print
```

⚠⇒ Sans protection, `*.f90` est remplacé par la liste des fichiers de suffixe `.f90` dans le *répertoire courant* avant l'exécution de `find` (*cf.* 3.2.1, p. 22).
S'il y en a plus d'un, cela provoque une erreur de syntaxe sur `find`.

11.3 Code de retour d'une commande ($?)

Toute commande UNIX renvoie en fin d'exécution un code entier, appelé valeur de retour (voir par exemple la valeur de retour entière de la fonction `main` en C) ou statut de fin (`return status`) à l'appelant qui peut le référencer par `$?` pour contrôler les commandes ultérieures. Par convention, le code de sortie `0` indique que la commande s'est bien déroulée, les autres valeurs pouvant renseigner sur le type de problème rencontré.

`cd /bin` `cd /introuvable`
`echo $?` affiche `0` `echo $?` affiche `1`

La commande `test` permet de produire un statut de fin en évaluant des conditions diverses portant sur des fichiers, des chaînes de caractères ou des entiers (*cf.* le chapitre 13).

11.4 Compléments

11.4.1 Inversion du statut de retour (`!`)

Le code de retour d'une commande peut être inversé (0 devient 1 et tout code différent de 0 devient 0) en faisant précéder la commande d'un point d'exclamation, suivi d'une espace [4] :

syntaxe

```
! ␣commande
```

11.4.2 Combinaison de commandes (`&&`)

syntaxe

```
commande_1 && commande_2
```

La première commande est exécutée. Si et seulement si elle réussit (code de retour égal à zéro), la seconde est alors exécutée.
N.-B. : le code de retour de la combinaison est celui de la dernière commande effectivement exécutée, donc différent de `0` si *`commande_1`* échoue et le code de retour de *`commande_2`* si *`commande_1`* réussit.
Par exemple, on lance un exécutable seulement si sa production (compilation et lien) s'est effectuée sans erreur.

```
g95 source.f90 && a.out
```

11.4.3 Combinaison de commandes (`||`)

syntaxe

```
commande_1 || commande_2
```

La première commande est exécutée. Si et seulement si elle échoue (code de retour différent de zéro), la seconde est alors exécutée.
N.-B. : le code de retour de la combinaison est celui de la dernière commande effectivement exécutée, donc `0` si *`commande_1`* réussit et le code de retour de *`commande_2`* si *`commande_1`* échoue.
On peut ainsi gérer de façon minimale les échecs de certaines commandes :

```
cp fichier1 fichier2 || echo la copie a échoué
```

♠ 11.4.4 La commande « nulle » :

Le shell possède aussi une commande « nulle », désignée par le caractère « `:` » qui n'exécute aucune action mais donne lieu à interprétation par le shell ; elle peut être utile dans les structures de contrôle (boucles infinies ou premier bloc d'un test type `if; then ... else ... fi` dans lequel seul le bloc `else` requiert une action). Comme son flux de sortie est vide, elle permet aussi de créer des fichiers vides par redirection (`: >`*`fichier_à_vider`*). Enfin, lors de son exécution, les variables sont évaluées, donc elle peut servir à positionner des variables :
`REP="${TMPDIR-/tmp}" :`
affecte `"${TMPDIR}"` s'il est défini (*cf.* 11.1.5 p. 80) ou sinon `/tmp` à `REP`.

4. Sans l'espace, « `!` » suivi d'une commande, ou du début d'une ligne de commande présente dans l'historique (avec suffisamment de caractères pour ne pas présenter d'ambigüité, comme dans les complétions) serait interprété comme la relance de cette commande.

♠ 11.4.5 Invite pour la commande `read`

La commande `read` peut aussi afficher un message, sans retour à la ligne (prompt) invitant à saisir la réponse. La syntaxe dépend des implémentations :

– sous bash : option `-p` suivie du message, le tout placé avant la première variable ;

```
read -p "entrez deux valeurs : " aa bb                    (en bash)
entrez deux valeurs :  AAA BBB
echo ${aa} ${bb}
AAA BBB                                                   (réponse du shell)
```

– sous ksh : caractère `?` suivi du message, le tout placé après la première variable.

```
read aa?"entrez deux valeurs : " bb                       (en ksh)
entrez deux valeurs :  AAA BBB
echo ${aa} ${bb}
AAA BBB                                                   (réponse du shell)
```

Chapitre 12

Procédures de commande

12.1 Fichiers de commande

Une procédure en shell (shell script) est un fichier texte contenant une suite de commandes UNIX et rendu exécutable (par la commande `chmod u+x nom_du_fichier` par exemple, *cf.* 3.1.4, p. 20).
Si la variable d'environnement `PATH` contient le chemin d'accès au répertoire contenant le fichier, la procédure peut être exécutée telle une commande UNIX, en tapant simplement son nom. Sinon, il faut fournir le chemin complet d'accès au fichier.
En pratique, on placera le répertoire courant (`.`) dans le `PATH` pour tester les procédures facilement, mais on créera un répertoire personnel `${HOME}/bin` que l'on adjoindra à la liste `PATH` pour stocker les commandes personnelles une fois mises au point. ⇐ ♥

12.1.1 Exemple de procédure sans paramètre

Les procédures suivantes affichent la date, le nom de l'utilisateur et le nom du calculateur. Pour le shell, tout ce qui, sur une ligne, suit le caractère « `#` » placé en début de mot est un commentaire [1].

```
# affichage de la date
date
whoami # du nom de l'utilisateur
hostname # du nom du calculateur
```

```
#! /bin/sh
# shell-script interprété en sh quel que soit le shell courant
# utilise des substitutions de commandes selon la syntaxe $(cmde)
echo nous sommes le $(date)
echo mon nom de login est $(whoami)
echo sur le calculateur $(hostname)
# fin du shell-script
```

12.1.2 Les paramètres des procédures

À l'intérieur d'une procédure, plusieurs variables spéciales sont automatiquement positionnées auxquelles on peut se référer :

`$0`	le nom du fichier de commande (tel que spécifié lors de l'appel)
`$1`, `$2`, ... `$9`	*les paramètres positionnels* (arguments) avec lesquels la procédure a été appelée (en fait le nombre de paramètres peut dépasser 9 et on peut accéder par exemple au dixième paramètre via `${10}`)
`$*`	la chaîne formée par l'ensemble des paramètres d'appel `"$1 $2 $3 ..."`
`$#`	le nombre de paramètres positionnels lors de l'appel
`$$`	le numéro du processus lancé (pid = **p**rocess **id**entifier)
`$_`	le dernier paramètre positionnel

1. Cependant, la première ligne `#!/bin/sh` permet de spécifier le chemin absolu du fichier exécutable qui va interpréter la procédure indépendamment du shell interactif de l'utilisateur.

12.1.3 Exemple de procédure avec paramètres

```
#!/bin/sh
echo la procédure $0
echo a été appelée avec $# paramètres
echo le premier paramètre est $1
echo le dernier paramètre est $_
echo la liste des paramètres est $*
echo le numéro du processus lancé est $$
```

12.1.4 Utilisation de `set`

`set` est une commande interne du shell à multiples facettes :
— `set` sans paramètre affiche la liste des variables et leur valeur
— `set` suivie d'une liste de mots affecte chacun des mots à un paramètre positionnel (`$1`, `$2`, ...)

```
set aaa bbb "ccc d"
echo $# affiche 3
echo $1 affiche aaa
echo $2 affiche bbb
echo $3 affiche ccc d
```

♥ ⇒ — `set` suivie d'une option introduite par `-` (ou `+`) permet de positionner des réglages du shell ; les options suivantes sont utiles dans la phase de mise au point des procédures :
  — `set -v` (**v**erbose) affiche les commandes (sans évaluation) avant de les exécuter
  — `set -x` (**x**trace) affiche les commandes après évaluation des substitutions avant de les exécuter

12.1.5 La commande `shift`

La commande interne `shift` permet de décaler les paramètres positionnels de la procédure d'une unité : le $n+1^{\text{e}}$ paramètre devenant le n^{e}, le premier est alors perdu.

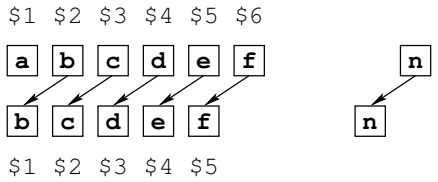


FIGURE 12.1 – Décalage des paramètres positionnels par `shift`

`shift` suivi d'un entier p décale de p les paramètres. La variable `$#` est automatiquement décrémentée de p.

Exemple de procédure utilisant `shift`

Si la procédure `shift1.sh` contient :

```
#!/bin/sh
echo liste des $# paramètres : $*
echo décalage d\'une unité
shift
echo liste des $# paramètres : $*
echo décalage de deux unités
shift 2
echo liste des $# paramètres : $*
exit
```

En lançant `shift1.sh aa b c ddd ee ff`, on obtient :

```
liste des 6 paramètres : aa b c ddd ee ff
décalage d'une unité
liste des 5 paramètres : b c ddd ee ff
décalage de deux unités
liste des 3 paramètres : ddd ee ff
```

La commande `shift` sera notamment utilisée dans une boucle sur les paramètres positionnels d'une procédure pour se débarrasser des paramètres après leur prise en compte.

♠ 12.2 Compléments

♠ 12.2.1 Distinction entre `$*` et `$@`

La variable spéciale `$@` est équivalente à `$*`, et contient la liste des paramètres positionnels. Mais ces variables se comportent différemment si elles sont entourées de guillemets :

- `"$*"` représente la liste des paramètres en *un seul mot* c'est-à-dire `"$1 $2 $3 ..."`
- `"$@"` représente la liste des paramètres avec *un mot par paramètre* c'est-à-dire `"$1" "$2" "$3" ...`. En particulier, si aucun paramètre positionnel n'est fourni, `"$@"` rend un mot vide.

♠ 12.2.2 Compléments sur la commande `set`

Lorsqu'on utilise `set` pour affecter les paramètres positionnels `$1`, `$2` ..., si le premier caractère du premier mot qui suit la commande commence par un signe `-`, `set` va l'interpréter comme l'introduction d'une option. Pour permettre d'affecter à `$1` une valeur commençant par `-`, il faut utiliser la syntaxe `set -- liste_de_mots`, les deux signes `--` indiquant la fin des options. ⇐⚠

```
set -- $(ls -al .profile)
echo le fichier $9 comporte $5 octets
```

Si on utilise `set` suivie d'une liste de mots dans une procédure, on va écraser autant de paramètres positionnels qu'on fournira de mots : il est donc prudent de les sauvegarder avant de lancer `set`.

Chapitre 13

Les commandes `test` et `expr`

13.1 La commande `test`

La commande `test` permet de comparer des nombres entiers ou des chaînes de caractères et de tester les propriétés de fichiers. Elle peut être invoquée sous deux formes :

— `test` *`expression`*

— `[␣`*`expression`*`␣]` en plaçant simplement la condition entre crochets [1]

Les arguments de `test` *doivent être séparés par un espace* de la même façon qu'un espace doit séparer la condition à tester des délimiteurs « `[` » et « `]` ».
La commande `test` fournit un code de retour (status en anglais) `$?` (*cf.* 11.3, p. 84) nul si la condition est réalisée et non nul sinon, status selon lequel des traitements différenciés pourront être conduits à l'aide de structures de contrôle du shell de type `if`, `while` ou `until`.

Remarque : les commandes `true` et `false` permettent de rendre un code de retour fixe, respectivement `0` et `1`.

13.1.1 Comparaisons arithmétiques

`-le`	**l**ower or **e**qual	inférieur ou égal à
`-lt`	**l**ower **t**han	inférieur à
`-eq`	**eq**ual	égal à
`-ge`	**g**reater or **e**qual	supérieur ou égal à
`-gt`	**g**reater **t**han	supérieur à
`-ne`	**n**ot **e**qual	différent de

Exemples :

```
test␣3␣-gt␣2
echo␣$?
--> 0
```

```
test␣3␣-lt␣2
echo␣$?
--> 1
```

```
#!/bin/sh
# sauve.sh : shell-script qui doit être appelé avec au moins 1 paramètre
if [ $# -lt 1 ]
then  # cas où $# < 1
   echo il manque le paramètre '$1' de la commande $0
else  # cas où $# >= 1
   echo appel correct
   cp $1 $1.sav
   echo copie de $1 dans $1.sav
fi
exit
```

1. Certains shells, dont `bash`, `ksh` et `zsh` admettent une syntaxe plus simple utilisant des doubles crochets, `[[ ... ]]`, entre lesquels certains métacaractères (dont `*`, `>`, `<`, `||`, `&&`) ne sont plus interprétés par le shell.

13.1.2 Comparaisons de chaînes de caractères

`-n s`	chaîne `s` de longueur non nulle
`-z s`	chaîne `s` de longueur nulle
`s1 = s2`	chaîne `s1` et chaîne `s2` identiques
`s1 != s2`	chaîne `s1` et chaîne `s2` différentes

Exemples :

```
[ -z "" ]
echo $?
--> 0
```

```
[ a1 = a2 ]
echo $?
--> 1
```

```
var1=aaa ; var2=aaa
[ ${var1} != ${var2} ]
echo $?
--> 1
```

Remarque : Ces tests sont très souvent appliqués à des variables dans des scripts. Il est alors prudent de délimiter la chaîne résultant de l'évaluation de la variable par des « " ». Cela permet d'éviter une erreur de syntaxe pour la commande `test` qui, sans les « " », ne verrait aucun argument dans le cas où la variable évaluée est constituée de blancs, est vide ou non initialisée.

```
var1=aaa ; var2="   "
test ${var1} != ${var2}
--> ksh: test: 0403-004 Specify a parameter with this command.
ou, suivant le shell
--> -bash: test: aaa: unary operator expected
echo $?
--> 1 en ksh, ou  --> 2 en bash
test "${var1}" != "${var2}"
echo $?
--> 0
```

13.1.3 Tests sur les fichiers

`-r toto`	**r**ead	`toto` est un fichier accessible en lecture
`-w toto`	**w**rite	`toto` est un fichier accessible en écriture
`-x toto`	e**x**ecute	`toto` est un fichier accessible en exécution
`-e toto`	**e**xist	le fichier `toto` existe
`-f toto`	**f**ile	le fichier `toto` existe et est un fichier ordinaire
`-d toto`	**d**irectory	le fichier `toto` existe et est un répertoire
`-s toto`	**s**ize	le fichier `toto` existe et est non vide

Exemples :

```
test -r /etc/motd
echo $?
--> 0
test -w /etc/motd
echo $?
--> 1
```

```
test -d /tmp
echo $?
--> 0
test -f /tmp
echo $?
--> 1
```

♠ 13.1.4 Combinaisons de conditions

`!`	négation
`-o`	ou logique (or)
`-a`	et logique (and)

Les parenthèses permettent de grouper des sous-conditions, mais elles ne doivent pas être interprétées par le shell, donc il faut les protéger par « \ ».

Exemple : `test␣\(␣"${a}"␣-ge␣2␣\)␣-a␣\(␣"${a}"␣-le␣5␣\)` est vrai si $2 \leqslant a \leqslant 5$.

13.2 La commande `expr`

La commande `expr` permet d'évaluer des expressions contenant des opérateurs arithmétiques ou logiques ou portant sur des chaînes de caractères. Chaque terme de l'expression à évaluer doit être séparé du suivant par un espace et il faut aussi protéger (par des « \ ») les métacaractères de l'interprétation par le shell. La commande `expr` est principalement utilisée pour effectuer des calculs sur des variables du type `expr ${`*var1*`} + ${`*var2*`}` que l'on peut ensuite affecter à une autre variable via une substitution de commande du type *var*`=$(expr ${`*var1*`} + ${`*var2*`})`.

13.2.1 Opérateurs arithmétiques

`+`	somme
`-`	différence (binaire) ou opposé (unaire)
`*`	produit
`/`	quotient de la division entière
`%`	reste de la division entière

Exemples :

```
expr␣3␣\*␣2
--> 6
expr␣13␣%␣5
--> 3
```

```
a=3
a=$(expr␣${a}␣\*␣2)
echo␣$a
--> 6
```

```
var1=3
var2=2
var3=$(expr␣10␣\*␣\(␣${var1}␣+␣${var2}␣\))
echo␣${var3}
--> 50
```

♠ 13.2.2 Autres opérateurs

La commande `expr` permet aussi des comparaisons grâce aux opérateurs binaires `<`, `<=`, `=`, `!=`, `>`, `>=`, qui utilisent des métacaractères devant évidemment être protégés d'une évaluation par le shell. Le résultat de la comparaison est `1` si la condition est réalisée, et `0` sinon (ne pas confondre ce résultat avec le code de retour `$?` de la commande, qui suit une convention inverse, comme le code de retour de `test`) [2].

Exemples :

```
expr␣2␣\>␣1
--> 1
echo␣$?
--> 0
```

```
expr␣2␣\<␣1
--> 0
echo␣$?
--> 1
```

13.3 Autres outils pour les calculs

13.3.1 Opérateurs intégrés au shell

La plupart des shells modernes (`ksh`, `bash` et `zsh` notamment), disposent de commandes internes qui permettent d'effectuer des opérations arithmétiques sur les entiers beaucoup plus rapidement qu'avec `expr` en utilisant des syntaxes nettement plus concises.

2. La commande `expr` permet aussi certains traitements sur les chaînes de caractères (recherche de motifs, détermination de longueur, recherche de sous-chaînes, de position d'une sous-chaîne).

Par exemple, ils interprètent directement les commandes du type
var=$((${*var1*}+${*var2*})), voire *var*=$((*var1*+*var2*)),
pour calculer la somme de deux variables et affecter le résultat à une troisième. À ce propos, ne pas oublier que, dans ce contexte, les entiers commençant par le chiffre `0` sont, comme en langage C, considérés comme écrits en octal : ainsi `echo $((011+2))` va afficher `11`, alors que `expr 011 + 2` affiche la valeur attendue, soit `13`. ⇐⚠

13.3.2 Les commandes de calcul non entier `dc` et `bc`

Les commandes internes du shell ainsi que la commande `expr` ne peuvent faire de calculs que sur les nombres entiers. Pour manipuler des nombres non entiers, on peut faire appel aux commandes `dc` ou `bc`.
Elles ont en commun notamment de travailler en précision arbitraire et de traiter soit un fichier de script passé en argument, soit par défaut l'entrée standard. Donc ces commandes se terminent[3] par une fin de fichier, soit `^D`. Elles se prêtent à l'écriture de shell-scripts si on récupère dans une variable les résultats, par défaut affichés sur la sortie standard.

La commande de calcul `dc`

La commande `dc` permet d'effectuer des calculs numériques en notation polonaise inversée, avec la notion de pile et d'opérateurs post-fixés.

La commande de calcul `bc`

La commande `bc` permet d'effectuer des calculs numériques dans une syntaxe proche de celle du langage C. Grâce à l'option `-l`, elle peut faire appel à certains fonctions de la bibliothèque mathématique du langage C. Par exemple, la fonction arc-tangente, appelée simplement `a`, peut être invoquée pour calculer π comme suit :

```
pi = 4.*a(1)
```

3. En particulier quand on a lancé `dc` en lieu et place de la commande `cd`.

Chapitre 14

Structures de contrôle du shell

Introduction

Comme les langages de programmation tels que le C, le fortran ou python, le shell possède des *structures de contrôle* qui prennent tout leur intérêt lors de l'écriture de shell-scripts, même si elles restent utilisables en interactif sur la ligne de commande. Elles permettent de décider, au vu de certaines conditions portant en général sur le code de retour de commandes, quelles instructions de la procédure doivent être exécutées et de mettre en œuvre de façon itérative une liste de commandes.
Une commande quelconque peut être utilisée pour conditionner une action [1], mais très souvent, c'est le code de retour de la commande `test` qui est utilisé, et il est donc nécessaire de consulter le chapitre 13 qui lui est consacré.
Ces structures de contrôle sont introduites par des mots-clefs réservés (`if`, `case`, `for`, ...) placés en début de ligne. Ne sont décrites ici que les syntaxes des shells de type Bourne ; pour ceux de type C-shell, voir par exemple DuBois (1995).
Pour éviter de confondre les syntaxes des différents langages, on consultera le tableau comparatif des structures de contrôle en shell, fortran et langage C en annexe page 119.

14.1 Les conditions

14.1.1 La construction `if ... fi`

Dans la structure `if`, la condition porte sur le code de retour de la commande qui suit `if`. En particulier, s'il s'agit d'une commande composée, par exemple avec un tube, c'est le code de retour de la dernière commande de la ligne qui est testé.

syntaxe

```
if commande
then
   commande(s)                          (exécutées si le code de retour est 0)
else
   commande(s)                  (exécutées si le code de retour est différent de 0)
fi
```

La partie `else ...` jusqu'à `fi` exclus est optionnelle.

Exemples

```
#!/bin/sh
# fichier qui contrôle si un paramètre au moins a été fourni
#
if test $# -eq 0 # éviter "=" pour une comparaison numérique
   then
   echo Relancer la commande en lui ajoutant un paramètre
else
   echo Enfin\! Vous avez compris.
fi
```

1. C'est un des aspects les plus déroutants par rapport aux structures de contrôle d'autres langages de programmation.

```
#!/bin/sh
# fichier qui indique si l'utilisateur de nom passé
#         en paramètre est connecté
if who | grep "^$1 " # recherche comme un mot en première colonne
   then
   echo $1 est connecté
fi
```

Les structures `if` ... `fi` peuvent s'emboîter et on peut contracter `else if` en `elif` en ne conservant alors que le `fi` terminal.

syntaxe

```
    if ...                              if ...
       then                                then
         ...                                 ...
    else if ...                         elif ...
            then                           then
               ...                          ...
         else if ...                    elif ...
                 then                       then
                   ...                      ...
              fi
         fi
    fi                                  fi
```

exemple avec elif

```
#!/bin/sh
if test $# -eq 0
then
   echo Relancer la commande en lui ajoutant un paramètre
elif who | grep "^$1 " > /dev/null   # affichage perdu
then
   echo $1 est connecté
fi
```

14.1.2 La construction `case ... esac`

Quand le nombre des tests imbriqués est important, et quand ils portent sur la correspondance entre une chaîne variable et des motifs, il est souvent plus efficace d'utiliser la structure `case` qui permet d'énumérer une liste de motifs possibles.

syntaxe

```
case variable in                          il y a comparaison de variable
    motif1) commande(s)                     avec les motifs de haut en bas.
                        ;;                    A la première coïncidence,
    motif2) commande(s)                                il y a exécution
                        ;;                  des commandes correspondantes
    ...                                       jusqu'au double point-virgule
esac                                                 et sortie de la structure.
```

Les motifs peuvent faire intervenir des caractères spéciaux qui seront interprétés comme :

- `*` = un nombre quelconque de caractères quelconques
- `[xyz]` = l'un quelconque des caractères énumérés entre les crochets
- `[x-z]` = l'un des caractères entre `x` et `z` dans l'ordre lexicographique
- `motif1|motif2|motif3` = un quelconque des motifs séparés par des `|`

Cas du C-shell : noter que la structure `switch` du C-shell et du langage C ne se comporte de manière similaire qu'à condition de terminer chaque cas par `breaksw`/`break;`. Le motif le plus général (équivalent de `*`) est, dans ces langages, désigné par `default`.

Exemple

```
#! /bin/sh
echo ecrivez OUI
read reponse
case ${reponse} in
          OUI)          echo bravo
                        echo merci infiniment       ;;
          [Oo][Uu][Ii]) echo merci beaucoup
                                                    ;;
          o*|O*)        echo un petit effort !      ;;
          n*|N*)        echo vous etes contrariant  ;;
          *)            echo ce n\'est pas malin
                        echo recommencez            ;;
esac
```

Attention : les motifs peuvent se recouvrir , mais **seule** la première coïncidence provoque l'exécution de commandes. L'ordre est donc important ; l'échange des deux premiers cas, par exemple, aura des conséquences.

14.2 Les structures itératives

14.2.1 La structure `for ... do ... done`

syntaxe

```
for variable in liste de mots  (par défaut: "$@")
do
     commande(s)
done
```

Les commandes entre `do` et `done` sont exécutées pour chaque mot de la liste.

⚠⇒ N.-B : ne pas confondre cette structure avec les boucles avec compteur du fortran ou du C ; en shell, la suite de valeurs que prend la variable est soit explicitement fournie soit, à défaut, la liste `"$@"` des paramètres positionnels de la procédure.

Exemple avec liste explicite

```
#! /bin/sh
for mot in 1 5 10 2 "la fin"
do
    echo mot vaut ${mot}
done
```

Exemple avec liste implicite

soit le script `do-echo.sh`

`do-echo.sh`

```
#! /bin/sh
for param
do
    echo +${param}+
done
```

`do-echo.sh 11 2 "3 3" 44`
affiche

```
+11+
+2+
+3 3+
+44+
```

Exemples avec liste produite par le shell

```
#!/bin/sh
for fichier in *.f90
do
    echo fichier ${fichier}
done
```

Par défaut, si aucun fichier du répertoire courant ne correspond au motif générique, celui-ci est restitué avec le ou les caractères jokers inchangés. Ce comportement peut être gênant dans les shell-scripts où on préférerait alors une liste vide. La commande `shopt -s nullglob` permet de forcer le shell à produire une liste vide dans ce cas. ⇐⚠

Procédure à un argument : le motif recherché.

```
#!/bin/sh
motif=$1
for fic in $(grep -l ${motif} *)
do
    echo le fichier $fic contient le motif $motif
done
```

14.2.2 La structure `until ... do ... done` (*jusqu'à ce que*)

syntaxe

```
    until commande
    do
         commande(s)
    done
```

Les commandes entre `do` et `done` sont exécutées *jusqu'à ce que* la commande qui suit `until` rende un code nul.

Exemple

Script qui boucle jusqu'à ce qu'un utilisateur se connecte :

```
#!/bin/sh
utilisateur=$1
until who | grep "^${utilisateur} " > /dev/null
do
   echo ${utilisateur} n\'est pas connecté
   sleep 2
done
echo ${utilisateur} est connecté
exit 0
```

14.2.3 La structure `while ... do ... done` (*tant que*)

syntaxe

```
    while commande
    do
         commande(s)
    done
```

Les commandes entre `do` et `done` sont exécutées *tant que* la commande qui suit `while` rend un code nul.

Exemple

Script qui boucle jusqu'à ce qu'un utilisateur se déconnecte :

```
#!/bin/sh
utilisateur=$1
while who | grep "^${utilisateur} " > /dev/null
do
   echo ${utilisateur} est connecté
   sleep 2
done
echo ${utilisateur} n\'est pas connecté
exit 0
```

14.3 Compléments : branchements

14.3.1 La commande `exit`

La commande `exit` [2], suivie éventuellement d'un code de retour arrête l'exécution de la procédure et rend ce code (0 par défaut) à l'appelant. Elle permet en particulier de ♥ ⇒ terminer prématurément la procédure en cas d'utilisation inadaptée et doit dans ce cas rendre un code différent de zéro.

```
...
if [ $# -lt 1 ]                              # test sur le nb d'arguments
   then
      echo "il manque le(s) argument(s)" >&2 # message sur sortie d'erreur
      exit 1                                 # sortie avec code d'erreur
fi
...
```

14.3.2 La commande `break`

La commande `break` permet de sortir d'une boucle avant la fin ; `break` ***n*** sort des n boucles les plus intérieures.

Exemple

```
#!/bin/sh
# fichier break.sh
while true       # boucle a priori infinie
do
  echo "entrer un chiffre (0 pour finir)"
  read i
  if [ "$i" -eq 0 ]
  then
     echo '**' sortie de boucle par break
     break          # sortie de boucle
  fi
  echo vous avez saisi $i
done
echo fin du script
exit 0
```

2. Elle se comporte de façon similaire à la fonction `exit` du langage C.

14.3.3 La commande `continue`

La commande `continue` saute les commandes qui suivent dans la boucle et reprend l'exécution en début de boucle.
`continue` ***n*** sort des $n-1$ boucles les plus intérieures et reprend au début de la n^{e} boucle.

Exemple

```
#!/bin/sh
# fichier continue.sh
for fic in *.sh
do
  echo "< fichier ${fic} >"
  if [ ! -r "${fic}" ]
    then
    echo "************************"
    echo "fichier ${fic} non lisible"
    continue # sauter la commande head
  fi
  head -4 ${fic}
done
exit 0
```

♠ 14.3.4 La commande `trap`

La commande `trap` permet d'intercepter les signaux envoyés à un processus en cours d'exécution (soit par un caractère de contrôle, soit par la commande `kill`, *cf.* 10.4.2, p. 75) et d'exécuter des commandes au lieu des actions habituellement requises par ces signaux.

syntaxe

```
trap action liste de signaux à intercepter
```

- Si la chaîne de caractères ***action*** entre `trap` et la liste des signaux est la chaîne vide (`''` ou `""`), le signal intercepté est ignoré par le processus.
- Si elle se réduit à `-`, la commande `trap` restitue aux signaux listés leur effet par défaut.
- Noter que l'action indiquée *se substitue* à l'action normalement provoquée par le signal : en particulier, si on intercepte un signal d'interruption, il faut préciser `exit` en fin d'action si on souhaite que le signal provoque finalement l'arrêt du processus. ⇐⚠
- Si l'action comporte plusieurs commandes, elle est entourée de caractères de protection « `'` » ou « `"` » et les commandes sont séparées par des « `;` » [3]

Par exemple, si un processsus en cours d'exécution reçoit un signal d'interruption, on peut souhaiter supprimer les fichiers temporaires de travail qui ont été créés et restituer si possible l'état initial des fichiers utilisateur avant d'interrompre le processus. On insérera alors dans la procédure une commande du type :

```
trap "/bin/rm -f /tmp/fichier_temporaire; exit 1" INT KILL TERM
```

3. L'action est interprétée deux fois par le shell :
   - à l'exécution de la ligne `trap` ;
   - et à l'interception éventuelle d'un signal de la liste.

Si le délimiteur est une protection forte, le shell n'interprétera l'action qu'au moment où le signal est intercepté. Si le délimiteur est une protection faible, le shell interprète les `$` dès de l'exécution de la ligne `trap`, sans attendre l'arrivée d'un signal.

Exemple

```
#! /bin/sh
# fichier trap.sh
for f in *
do
  echo début de boucle $f
  trap "echo fin provoquée par signal; \
        exit 3" INT QUIT
  echo fin de boucle $f
done
echo fin de script
exit 0
```

♠ 14.3.5 Structures itératives et redirections

Pour rediriger la sortie standard d'une structure itérative, on peut rediriger chaque commande sur la fin du fichier de sortie par `>>`, en veillant à vider le fichier en début de boucle. Cela permet entre autres de choisir finement quelles commandes on redirige à l'intérieur de la boucle. Mais il est parfois préférable de placer la redirection juste apès
♥ ⇒ le mot réservé `done` qui termine la boucle.

Exemple : redirection à chaque commande

```
#!/bin/sh
# fichier redir-iterat2.sh
# redirection et structure itérative
# version redirection commande par commande
cat /dev/null > resultat # partir d'un fichier vide
for i in 1 2 3
do
  echo $i >> resultat   # accumuler dans la boucle
done
exit 0
```

Exemple : redirection globale du bloc

```
#!/bin/sh
# fichier redir-iterat.sh
# redirection et structure itérative
# version redirection globale
for i in 1 2 3
do
  echo $i
done > resultat # redirection après done
exit 0
```

Chapitre 15

Retour sur le shell

15.1 Le shell : approfondissement

15.1.1 Notion de commande interne

Certaines commandes très usitées ont été intégrées au shell, ce qui rend leur exécution plus performante car elle ne nécessite pas le lancement d'un nouveau processus. Ces commandes sont appelées *commandes internes* (builtin) ou primitives du shell. De plus, les commandes internes permettent d'affecter le shell courant : par exemple, la commande `cd` doit être une commande interne pour que le changement de répertoire qu'elle provoque perdure après son exécution.
Parmi les commandes internes, citons `bg`, `cd`, `exit`, `export`, `echo`, `false`, `fg`, `kill`, `read`, `set`, `shift`, `trap`, `true`, `type`... Les commandes internes ne sont pas documentées sous une page de manuel à leur nom, mais dans le manuel du shell lui-même ; toutefois, sous `bash`, la commande `help` *`cmd`* affiche une aide succincte sur la commande interne *`cmd`*.

15.1.2 Les alias du shell

Les alias permettent de définir des raccourcis de commandes existantes en leur spécifiant éventuellement des options par défaut.

syntaxe

```
alias nom_court='commande [options]'
```

La commande interne `alias` sans argument affiche la liste des alias définis.

Exemples

`alias ls='ls -F'` force l'option `-F` (**F**lag)
`alias rm='rm -i'` force l'option de confirmation
`alias la='ls -a'` pour voir les fichiers cachés
Par défaut, si un nom d'alias coïncide avec une commande, l'alias est prioritaire. Mais il suffit de préfixer le nom de la commande par une contre-oblique « \ » pour retrouver la commande native.
La commande interne `unalias` suivie du nom d'un alias permet de le supprimer.

15.1.3 Les fonctions du shell

Afin de rendre plus modulaire l'écriture des shell-scripts, il est possible de définir des fonctions constituées d'une suite de commandes mémorisées afin d'être exécutées lors de l'appel à la fonction. Deux syntaxes sont disponibles pour définir une fonction :

syntaxe

```
function nom
{
  liste_de_commandes
}
```

syntaxe

```
nom ()
{
  liste_de_commandes
}
```

Exemple de fonction

La fonction `gfortran2003mni` permet de lancer le compilateur fortran `gfortran` avec des options de compilation très restrictives pour la mise au point des programmes sans avoir à répéter ces options explicitement à chaque appel.

```
gfortran2003mni ()
{
    gfortran -Wextra -Wall -fimplicit-none -Wunused -ffloat-store \
        -fbounds-check -Wimplicit-interface -ftrapv -fexceptions  \
        -pedantic -fautomatic -std=f2003  $* ;
}
```

Les fonctions admettent des paramètres qui sont référencés comme les paramètres positionnels des shell-scripts. Mais les autres variables manipulées dans une fonction sont globales par défaut, sauf si on les déclare locales suivant la syntaxe `local` *`variable`*.

15.1.4 Interprétation d'un nom de commande avec `type`

La commande interne `type` suivie d'un mot permet d'identifier comment ce mot sera interprété par le shell. Dans l'ordre, le shell scrute successivement si le mot passé peut être :

- un alias,
- un mot-clef (`if` par exemple),
- une fonction,
- une commande interne,
- un shell-script ou un programme exécutable (dont la recherche suit l'ordre des répertoires indiqué par la variable d'environnement `PATH`)

Avec `type -a` *`mot`*, il est possible d'afficher toutes les interprétations possibles de l'identificateur *`mot`*.

```
type -a echo
echo est une primitive du shell
echo est /bin/echo
```

Signalons enfin que, pour accélérer l'accès aux commandes les plus utilisées, `bash` entretient une table de leurs chemins. La commande interne `hash` permet d'afficher la liste de ces correspondances. Après une modification de la variable `PATH`, la liste des répertoires ⚠⇒ scrutés pour rechercher une commande change et il est parfois nécessaire de supprimer l'entrée associée dans cette table pour éviter qu'elle ne court-circuite cette recherche : `hash -d` *`cmd`* supprime l'entrée associée à la commande *`cmd`* et `hash -r` vide cette table.

15.1.5 Algorithme d'interprétation de la ligne de commande

Enfin, l'interprétation de la ligne de commande par le shell suit un algorithme complexe dans lequel interviennent successivement :

- l'interprétation des redirections [1] ;
- le découpage en mots ;
- l'identification des mots-clefs du shell (en particulier ceux introduisant les structures de contrôle) ;
- l'interprétation des alias ;

1. Les redirections sont interprétées très tôt dans ce processus, ce qui autorise à les placer de façon très libre sur la ligne de commande, y compris, même si cela reste vivement déconseillé pour la lisibilité, entre le nom de la commande et ses paramètres, voire entre les paramètres.

— le développement des accolades ;

— la substitution des « `~` » ;

— la substitution des variables ;

— la substitution des commandes dans les expressions `$(...)$` ;

— la substitution des caractères jokers par des noms de fichiers...

Certaines de ces étapes pouvant s'avérer récursives, on se reportera pour plus de précision, au schéma de la figure 7.1 du chapitre 7 de Cameron et Rosenblatt (1997).

15.1.6 Fichiers d'initialisation du `shell`

Lors de l'ouverture d'une session, un shell de type Bourne (`sh`, `ksh`, `pdksh` ou `bash`) exécute successivement plusieurs fichiers d'initialisation [2], qui permettent de configurer l'environnement de travail :

1. `/etc/profile` qui détermine l'environnement commun à tous les utilisateurs
2. `~/.bash_profile` en `bash` ou (sinon) `~/.profile` qui configure l'environnement personnel au niveau de chaque utilisateur particulier
3. puis, suivant le shell utilisé, éventuellement un fichier dont le nom peut être défini par la variable d'environnement `ENV` et qui est par défaut :
   — `~/.kshrc` en `ksh`
   — `~/.bashrc` en `bash`

Dans ces fichiers sont définis ou modifiés notamment les variables `PATH` (liste des chemins de recherche des commandes), `MANPATH` (liste des chemins de recherche des manuels des commandes), `PS1` (**P**rompt **S**tring, invite primaire) [3] du shell).

15.1.7 Options du `shell`

La commande interne `set -o` affiche les options en cours pour le shell. Elle permet aussi de positionner certaines options, par `set -o` *`option`* ou de les supprimer par `set +o` *`option`*. Par exemple `set -o noclobber` (ou aussi `set -C`) configure le shell qu'il empêche d'écraser un fichier existant par redirection de sortie. Par ailleurs `set -o vi` ou `set -o emacs` permet de choisir le mode d'édition de la ligne de commande.
D'autres options plus nombreuses (une quarantaine) sont accessibles via la commande interne `shopt`. `shopt` permet d'afficher la liste de ces options. `shopt -s` permet d'activer une option alors que `shopt -d` permet de la désactiver. Par exemple `shopt -s dotglob` permettrait de considérer par défaut les fichiers dont le nom commence par un point, qui sont habituellement cachés.

15.2 Exécutions spéciales de commandes

15.2.1 Exécution dans le shell courant

Il est parfois nécessaire d'exécuter certaines commandes dans le shell courant, notamment pour positionner des variables (personnalisation de l'environnement).

syntaxe

`.␣`*`shell-script`*

2. Pour les shells de type `csh` (`csh` et `tcsh`), les fichiers d'initialisation de site exécutés sont `/etc/csh.cshrc`, puis `/etc/csh.login` et les fichiers d'initialisation personnels `~/.cshrc` (en `tcsh` `~/.tcshrc` s'il existe, sinon `~/.cshrc`) puis `~/.login`.

3. Lorsque l'utilisateur saisit une chaîne de caractères délimitée par des « ' » ou des « " » comportant un retour-ligne, ou une structure de contrôle sur plusieurs lignes, l'invite du shell est modifiée pour signaler que le shell attend la fermeture de cette chaîne ou de cette structure. Plus généralement, dans tous les cas où le shell attend une saisie pour terminer la lecture d'une commande, c'est l'invite secondaire `PS2` (`>` par défaut) qui est affichée.

En particulier le fichier d'initialisation[4] de la session, `~/.profile` est exécuté de cette façon pour pouvoir affecter le shell courant.

♠ 15.2.2 Exécution en remplacement du shell courant

La commande interne `exec` *`commande`* permet de susbtituer une commande au shell courant, de telle sorte que lorsque la commande se termine, le processus à partir duquel `exec` a été lancé se termine aussi.
C'est une méthode pour changer de shell interactif sans « empiler » les processus : `exec bash` vient substituer `bash` à l'interpréteur courant. Si on saisit ensuite `exit`, on ferme la session interactive.
La commande `exec` *`redirection`* (sans commande) en début de script permet d'effectuer des redirections qui porteront sur toute la procédure.
`exec > fic` en début de script
redirige la sortie sortie standard vers `fic` pendant tout le script.

♠ 15.2.3 Double évaluation par le shell : `eval`

Quand une double évaluation de la ligne de commande par le shell est nécessaire, on préfixe la commande par `eval`. Une première substitution est alors effectuée avant l'évaluation normale par le shell.

⚠⇒ Le cas le plus classique est l'accès au contenu *du contenu* d'une variable : on cherche à évaluer `${${`*`variable`*`}}`, où *`variable`* contient le *nom* de la variable à évaluer. On écrira `eval valeur=\${${`*`variable`*`}}` en protégeant le premier `$` de la première substitution par le shell.

Exemple : affichage du dernier argument positionnel d'un script

Si le script `test-eval.sh` contient

```
i=$#
echo variable \${${i}}
eval echo valeur \${${i}}
```

L'appel `test-eval.sh un deux trois` affichera le nom du dernier paramètre puis sa valeur, par exemple :
```
variable ${3}
         valeur trois
```

15.2.4 Priorité d'exécution avec `nice`

Une commande grosse consommatrice de ressources peut être lancée avec une priorité plus faible grâce à la commande `nice`.

syntaxe

```
nice commande
```

Si le processus est déjà lancé, la commande `renice` permet de baisser sa priorité. Noter que seul l'administrateur peut lancer une commande avec une priorité plus élevée ou modifier la priorité d'un processus qui ne lui appartient pas.

15.2.5 Mesure du temps d'exécution avec `time`

La commande `time`, précédant une commande UNIX permet d'afficher la durée d'exécution de cette commande *sur la sortie d'erreur standard*. L'option `-p` est conseillée pour obtenir une mise en forme portable. Trois durées sont indiquées : le temps réel écoulé entre lancement et fin de la commande, le temps utilisateur et le temps système.

Exemple `time -p ./calcul_det_recursif.x >/dev/null`

```
real 2.16
user 2.15
sys 0.00
```

4. Ceci vaut pour les shells de type `sh`, comme `bash` et `ksh`. Mais, en `csh` ou `tcsh`, l'exécution dans le shell courant du script `.cshrc` est lancée par : `source␣.cshrc`

15.2.6 Récursivité

Un shell-script peut en appeler un autre, pourvu que ce dernier soit accessible soit par son chemin explicite, soit (en l'absence de `/` dans le nom du script) en explorant les répertoires de la variable d'environnement `PATH`. Mais un shell-script peut aussi s'appeler lui-même, de façon récursive. La récursivité, si elle peut apparaître comme une méthode de programmation concise et élégante, se heurte cependant parfois à des problèmes de performances ou de ressources en termes de nombre de processus lancés.

Exemple

Le script `scripts/recursive-ls0.sh` dresse la liste du répertoire qui lui est passé en argument de façon récursive [5], en se relançant dès qu'il rencontre un répertoire. Attention, pour ne pas alourdir cet exemple, il est présenté dans sa version la plus simple (absolument pas robuste), en particulier sans contrôle de validité du paramètre passé.

recursive-ls0.sh

```
#!/bin/sh
# fichier  recursive-ls0.sh
cd $1
pwd
for i in $(ls)
do
   if [ -d "${i}" ]
   then
       recursive-ls0.sh "${i}"
   else
       ls -l "${i}"
   fi
done
```

15.2.7 Exécution de commande avec les droits administrateur avec `sudo`

`sudo` permet à un utilisateur d'exécuter des commandes avec temporairement les droits de l'administrateur ou (option `-u user`) ceux d'un autre utilisateur. Il faut, pour cela, que l'administrateur ait auparavant autorisé cet utilisateur à lancer avec `sudo` certaines commandes particulières. Cela lui permet d'effecteur certaines tâches ponctuelles d'administration (par exemple monter des systèmes de fichiers) sans avoir à connaître le mot de passe de l'administrateur. Cette commande permet une gestion individualisée des droits plus fine que via les droits d'endossement (*cf* 3.1.4, p. 19).

15.3 Autres commandes internes

15.3.1 Affichage de texte messages avec `echo`

La commande `echo` a été largement utilisée dans les exemples dans sa version par défaut. Elle est disponible comme primitive du shell, mais aussi comme commande `/bin/echo` avec les mêmes fonctionnalités. Elle se contente d'afficher à l'écran le texte fourni en argument, une fois ce texte soumis, sauf protection explicite (*cf.* 11.2.4, p. 83), à interprétation par le shell.
Si on lui passe l'option `-e`, elle peut interpréter certaines séquences d'échappement introduites par une contre-oblique, comme `\n` pour le saut de ligne, `\t` pour la tabulation horizontale (*cf.* 5.5.3, p. 49). Ne pas oublier que la contre-oblique ne doit pas être exposée au shell, et doit donc être protégée. Par exemple, pour afficher les composantes

5. Ne pas oublier que la commande `ls -lR` réalise cet affichage de façon plus efficace.

du chemin de recherche des commandes (`PATH`) à raison d'une par ligne, on peut utiliser (*cf.* 11.1.6, p. 81) : `echo -e "${PATH//:/\n}"` ou encore `echo -e ${PATH//:/\\n}`
Enfin, l'option `-n` supprime le changement de ligne en fin de message. Elle permet par exemple de concaténer des messages sur une seule ligne :
`echo -n message ; echo suite` affichera `messagesuite`

♠ 15.3.2 Analyse syntaxique avec `getopts`

`getopts` simplifie le découpage (parsing) des paramètres optionnels passés à un script. Par exemple, `getopts x:yz` *`nom_option`*

- recherche si une des options `x`, `y` ou `z` a été passée,
- place cette option dans la variable *`nom_option`*
- si *`nom_option`* vaut `x`, la valeur fournie pour l'option `x` est placée dans la variable `OPTARG`

On doit fournir à `getopts` toutes les lettres-clefs (ici `x`, `y` et `z`) spécifiant les options possibles en faisant suivre celle(s) qui admettent un argument par « `:` ».
`getopts` rend un code nul tant qu'il reste des options dans la liste des paramètres passés au script : on peut donc l'utiliser dans une structure `while` et exploiter les valeurs *`nom_option`* et `OPTARG` via une structure `case`.

```
while getopts x:yz nom_option
do
  case ${nom_option} in
     x)
      x_value="${OPTARG}"
      ;;
     y)
      exploitation de l'option -y
      ...
      ;;
     z)
      exploitation de l'option -z
      ...
      ;;
  esac
done
```

15.4 Internationalisation

Si l'anglais reste la langue incontournable des systèmes informatiques, de grands progrès ont été accomplis récemment dans l'internationalisation des programmes. Tenir compte des spécificités linguistiques et culturelles suppose non seulement de représenter les caractères des différentes langues via des codes plus ou moins universels, mais aussi de respecter les conventions d'écriture de certaines informations comme la date, la monnaie ou les nombres décimaux.
De nombreuses commandes sont affectées par le choix de la langue, notamment `man`, `wc`, `grep`, `awk`, `sort` mais aussi `ls` pour l'ordre d'affichage.

15.4.1 Les variables de contrôle local

L'ensemble de ces variantes relève du « contrôle local » qui recouvre plusieurs catégories de particularités, spécifiées par des variables d'environnement dont le nom commence

par `LC_`[6], parmi lesquelles :

- `LC_CTYPE` qui définit la classification des caractères en majuscules, minuscules, chiffres, ponctuation,... par exemple les classes `[:upper:]`, `[:lower:]`, `[:digit:]`, ... (*cf.* 6.3.3, p. 53) ;
- `LC_COLLATE` qui définit les regroupements de caractères éloignés dans les tables ou équivalences (*cf.* p. 53) pour spécifier l'ordre lexicographique, comme `[=a=]` qui, en français, contient notamment `à`, `â`, `ä` ;
- `LC_TIME` qui définit le format d'affichage de la date et de l'heure ;
- `LC_NUMERIC` qui définit le format des nombres et en particulier le séparateur (« `,` » en français, mais « `.` » en anglais) entre partie entière et partie fractionnaire des nombres décimaux ;
- `LC_MONETARY` qui définit l'unité monétaire ;
- `LC_MESSAGES` qui définit la langue dans laquelle sont affichés les messages du système ;
- `LC_PAPER` qui définit le format de la page par défaut (A4 en Europe ou Letter aux États-Unis) ;
- `LC_ALL` qui permet de définir globalement toutes les autres.

Exemples : affichage du jour de la semaine avec `date`

```
LC_ALL=C  date "+%A"                                  Friday
LC_ALL=de_DE.ISO-8859-1  date "+%A"                   Freitag
LC_ALL=es_ES.ISO-8859-1  date "+%A"                   viernes
LC_ALL=fr_FR.ISO-8859-1  date "+%A"                   vendredi
LC_ALL=en_EN.ISO-8859-1  date "+%A"                   Friday
```

15.4.2 Les valeurs des variables de contrôle local

Les variables de contrôle local permettent d'indiquer :

- la langue avec deux lettres minuscules : `en` pour english, `fr` pour le français, ...
- éventuellement les variantes locales de la langue grâce à deux lettres majuscules : `en_US` pour l'anglais des États-Unis, `fr_FR` pour le français de France, `fr_CA` pour le français du Canada francophone...
- le codage (*cf.* 4.1.2, p. 30) des caractères utilisés pour la langue : `ISO-8859-1` pour le `latin1` des langues européennes, ou plus souvent maintenant `UTF-8` pour de l'unicode codé en `utf-8`.

Le format général utilise les séparateurs « `_` » entre la langue et sa variante locale et « `.` » entre langue et codage : `fr_FR.UTF-8` par exemple.
La norme par défaut est spécifiée par la valeur `C` ou `POSIX` des variables de langue : elle correspond aux conventions nord-américaines des systèmes qui restent toujours disponibles. En particulier, avec la norme `POSIX`, le classement des caractères se fait selon l'ordre de la table ascii, avec notamment toutes les majuscules avant les minuscules. On peut se douter que de nombreuses commandes UNIX notamment les filtres comme `awk` (*cf.* 9.2.5, p. 66), `sort`, `tr`, `wc`, mais aussi `ls` (qui classe ses résultats), vont voir leur comportement affecté par le positionnement de ces variables.
La commande `locale` permet d'afficher les valeurs de toutes les variables de contrôle local. Rappelons qu'il est possible de modifier temporairement la valeur d'une variable le temps du déroulement d'une commande, grâce à la syntaxe :
`variable=valeur` `commande`
On pourra forcer temporairement le séparateur décimal à sa notation anglo-saxone (point) en préfixant par exemple une commande `awk` par `LC_NUMERIC=C` pour éviter de « perdre » la partie fractionnaire des nombres.

6. La variable d'environnement `LANG` peut aussi être utilisée au niveau global. Enfin la variable d'environnement `LANGUAGE` détermine la langue utilisée pour les messages à l'utilisateur :

```
LANGUAGE=es_ES.UTF-8 man toto                  No hay ninguna página sobre toto
```

15.4.3 Outils associés au codage UTF-8

Pour visualiser des caractères codés en UTF-8, il faut disposer :

— de polices adaptées qui comportent les glyphes utilisés au codage UTF-8 (`iso10646`) ;

— de terminaux gérant les caractères multi-octets.

Terminaux et codage

Le terminal graphique `xterm` possède des options permettant de prendre en compte les variables de contrôle local (`-lc`) ou simplement l'encodage (`-en`) ou de spécifier l'encodage UTF-8 (`-u8`) ainsi qu'une version spécialisée pour utf-8 : `uxterm`. Une fois `xterm` lancé, il est possible d'activer l'UTF-8 via un menu accessible avec la combinaison touche de Contrôle et bouton droit de la souris.
Le terminal `konsole` possède un menu de configuration qui offre un très large choix de codages dont `iso-8859-1` et `utf-8`.
Un outil de conversion permettant le dialogue entre un terminal UTF-8 et une application dans un autre codage permet d'assurer la transition pour les applications pas encore multilingues. Il s'agit du filtre `luit`, souvent appelé de façon automatique par certaines applications dont `xterm`.

Éditeurs de texte et codage

Par ailleurs, les éditeurs de texte comme `vim` et `emacs` possèdent, dans leurs versions les plus récentes, les outils permettant de prendre en charge le multiliguisme et les caractères multi-octets, avec un éventuel transcodage au vol (*cf.* 4.3.3 p. 36 pour `vim` et 4.4.7 p. 40).

Autres outils

D'autre part, c'est dans un environnement UTF-8, que la commande `wc -m -c`[7] va permettre de détecter le nombre de caractères sur 2 octets (en supposant qu'il n'y ait aucun caractère sur plus de 2 octets).
`LC_ALL=fr_FR.UTF-8 wc -m -c` *`fichier-utf.txt`*
Mais elle pourra afficher un message d'erreur[8] comme par exemple en français :
`Chaîne multi-octets ou étendue de caractères invalide ou incomplète`
si le fichier comporte par exemple des caractères avec des signes diacritiques codés en ISO-8859-1.
Noter enfin que l'option d'encodage UTF-8 est proposée dans certains outils de production de fichiers texte, par exemple à partir de `pdf` : `pdftotxt -enc UTF-8` est capable d'extraire des ligatures telles que « ffi » alors stockées sur 3 octets.

7. L'ordre des options de `wc` est sans importance sur l'ordre d'affichage des nombres demandés : le nombre de caractères précède toujours le nombre d'octets, de même que le nombre de lignes précède le nombre de mots, qui précède le nombre de caractères.

8. C'était le cas dans les versions 5 de `wc`, mais en version 7, aucun message d'erreur n'est émis : les caractères invalides en UTF-8 ne sont tout simplement pas comptabilisés et le nombre de caractères apparaît alors inférieur d'autant.

Chapitre 16

Conclusion

16.1 Du bon usage d'unix

Connaître parfaitement la syntaxe du shell et des commandes UNIX n'est ni absolument nécessaire (il est toujours possible et même conseillé de consulter la documentation en ligne), ni certainement suffisant pour utiliser ce système efficacement. L'objectif de ce court chapitre est de donner quelques conseils de bon usage des outils qui viennent d'être présentés.

16.1.1 Analyser avant de coder...

Avant de se lancer dans la programmation d'un shell script, il est toujours prudent d'analyser le problème posé pour rechercher l'algorithme permettant de le traiter.

16.1.2 Choisir les outils dans la panoplie unix

Les commandes disponibles sous UNIX sont si diversifiées qu'il existe bien souvent de très nombreuses combinaisons (en particulier des combinaisons de filtres) pour mettre en œuvre l'algorithme choisi. L'arbitrage reste très marqué par l'expérience personnelle acquise sur les différents outils, mais une culture minimale est nécessaire pour éviter de passer du temps à reconstruire laborieusement des fonctions déjà existantes : en particulier, il serait maladroit de réinventer un programme de tri alors que la commande `sort` est déjà optimisée dans ce but. Par ailleurs, il faut garder présent à l'esprit la puissance de `sed` dans les substitutions de motifs, celle de `awk` dans les opérations de type tableur. Mais `wc -l` sera plus rapide que `awk 'END{print NR}'` pour simplement compter le nombre de lignes d'un fichier. Enfin, pour assurer une meilleure *portabilité* des scripts, on privilégiera les commandes et les options conformes à la norme POSIX.

16.1.3 Mettre au point et corriger

Il est souvent préférable de partir d'une version élémentaire de la procédure, de la tester dans les cas les plus simples avant de l'enrichir progressivement pour lui permettre de traiter les cas plus délicats.
Mais il vaut mieux adopter dès le début les écritures les plus *robustes*, par exemple `${var}` ou mieux `"${var}"` (*cf.* 13.1.2, p. 91) au lieu de `$var` pour la valeur de la variable `var`. L'écriture systématique de messages via la commande `echo`, et l'usage des options `set -v` qui affiche la commande avant de l'exécuter et surtout `set -x` (*cf.* 12.1.4, p. 88) qui affiche chaque commande après évaluation des susbtitutions de variables (`${...}`) et de commandes (`$(...)`) peuvent aider à localiser les éventuelles erreurs.

16.1.4 Documenter

Lors de la réutilisation d'un script (par son auteur ou par d'autres), la présence de documentation constitue une aide essentielle. Il est donc conseillé de *commenter* les shell-scripts dès leur conception : auteur, date, nom du fichier, système et machine de développement, objectif et méthode, puis version et date de mise à jour doivent être indiqués dans les premières lignes. Une pratique très saine consiste à définir une chaîne de caractères indiquant brièvement la syntaxe d'appel (parfois dans une fonction) en tête

de script et à l'afficher en cas d'usage non conforme (nombre de paramètres et options par exemple). Les différentes parties doivent être identifiées et les points délicats ainsi que les faiblesses connues précisés dans les commentaires. À ce sujet, les formulations les plus concises, parfois très spécifiques d'UNIX, ne sont pas forcément les plus lisibles et les plus à même d'évoluer [1]. Enfin, la mise en évidence des blocs de commandes par une indentation systématique est fortement conseillée [2].

16.2 Conclusion

Cette présentation de l'environnement UNIX, reste loin d'être exhaustive. Le shell, bien connu comme interpréteur de commande, y est aussi décrit comme un langage de programmation avec ses variables, ses propres structures de contrôle et ses procédures paramétrées. Il est donc capable de prendre en charge efficacement de nombreuses tâches de transfert, de mise en forme de données, de gestion de fichiers... autant de travaux difficiles à confier à des programmes en langage compilé, tels que le fortran, plus adaptés au calcul scientifique lui-même.
Une des limitations majeures du shell est qu'il ne manipule que des chaînes de caractères : il faut avoir recours à des utilitaires comme `awk` qui permet d'accéder aux opérateurs du langage C, pour effectuer du calcul en flottant. D'autres langages, actuellement en plein développement tels que `perl` (*cf.* http://www.perl.org/) allient sous une syntaxe unique les capacités de manipulation de fichiers, de chaînes de caractères, mais aussi de calcul en flottant [3].
Mais, pour le calcul scientifique, il est précieux de disposer d'un système d'exploitation du domaine public comme linux que l'on peut installer sur la plupart des ordinateurs, et qui permet de bénéficier de tous les outils développés dans la communauté UNIX, qui s'étend aujourd'hui bien au-delà du secteur universitaire. En particulier, il est possible sous linux de disposer de logiciels libres de calcul scientifique comme par exemple `scilab` (*cf.* http://www.scilab.org/) qui intègre une très riche bibliothèque mathématique et le fortran 2003 grâce aux versions récentes du compilateur `gfortran` (*cf.* http://gcc.gnu.org/fortran/).
On ne saurait conclure sans mentionner le langage `python` (*cf.* http://www.python.org/) en plein essor dans différents domaines, dont le calcul scientifique et la visualisation graphique ; `python` peut aussi jouer un rôle d'intégration entre des applications écrites dans différents langages. Mais il reste plus efficace de confier les tâches de manipulation de fichiers directement au système d'exploitation UNIX, dont une bonne connaissance constitue un atout.

1. À titre d'exemple, pour changer tous les noms de fichiers du répertoire courant de suffixe `.f77` en suffixe `.f90`, doit-on préférer la commande (inspirée de Powers *et al.* (2003), 10.9, p. 212 et 213)
`ls -d *.f77 | sed "s/\(.*\)\.f77$/mv '&' '\1.f90'/" | sh`
à un script avec une boucle explicite ?

2. Le nombre d'espaces choisi doit rester faible pour permettre d'écrire chaque commande sur une seule ligne, sous peine de devoir protéger les retour-ligne par des contre-obliques.

3. `perl` présente aussi l'avantage de pouvoir être utilisé sous des systèmes d'exploitation autres qu'UNIX.

Annexe A : Bref guide d'installation et d'utilisation de MobaXterm

Chapitre rédigé par Christophe Boitel (Christophe.Boitel@lmd.polytechnique.fr).

1 MobaXterm

Ce logiciel regroupe de très nombreuses fonctionnalités au sein d'une seule application fonctionnant sous Windows. Dans un terminal, elle donne accès à un jeu de commandes Unix classiques relatives à l'arborescence et à la manipulation de fichiers textes. Elle propose également un ensemble d'outils graphiques de connexion et d'accès à des environnements distants.
Ce document décrit succinctement la manière

- d'établir une connexion sécurisée (protocole ssh) sur un compte du serveur de l'université `sappli1.datacenter.dsi.upmc.fr`
- d'échanger des fichiers entre le serveur sappli et son ordinateur personnel
- d'afficher des applications du serveur sappli réclamant un environnement fenêtré de type Unix,

mais également hors connexion (localement)

- d'avoir accès au travers d'un terminal à un ensemble de commandes Unix
- de compiler des programmes écrits en langage C ou FORTRAN à l'aide des compilateurs gcc et gfortran.

2 Installation

1. Se rendre sur le site http://mobaxterm.mobatek.net
2. Aller dans le menu **Download**
3. Choisir la version **Home Edition** en téléchargement libre
4. Dans la colonne **Home Edition** cliquer sur **Download now** puis **MobaXterm Home Edition v7.7 (installer edition)**
5. Lancer le programme **MobaXterm_Setup_7.7.msi** et suivre les instructions de l'assistant d'installation. Noter en passant le dossier où le logiciel sera installé (Ex : C:\Program Files (x86)\Mobatek\MobaXterm Personal Edition\)

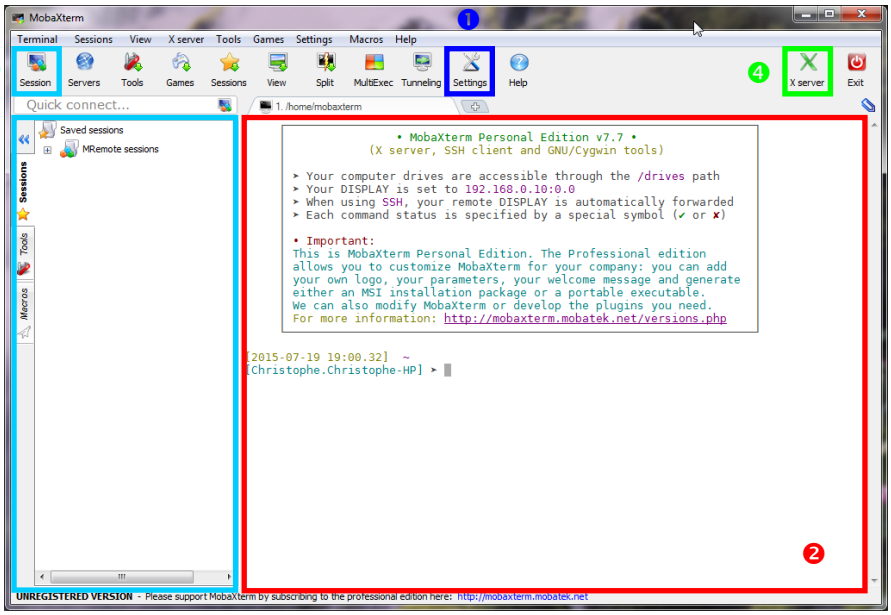
Installation des plugins
Les fonctionnalités de MobaXterm peuvent être étendues grâce à des compléments, ou plugins, à placer dans le dossier d'installation de MobaXterm. Typiquement les compilateurs ne sont disponibles que sous forme de plugins.

6. Sur le site de MobaXterm choisir le menu **Plugins**
7. Télécharger **Gcc** puis **GFortran**
8. Déplacer les fichiers téléchargés, **Development.mxt3** et **GFortran.mxt3**, dans le dossier d'installation de MobaXterm
9. Vérifier la bonne prise en compte des modules gcc et gfortran. Taper dans le terminal les commandes gcc ou gfortran
   `gcc : no input files` doit s'afficher en retour. Si ce n'est pas le cas, vérifier que le module **Development.mxt3** est bien présent dans le dossier de l'application. Même démarche pour la commande gfortran avec le plugin **GFortran.mxt3**.
10. fin de l'installation.

3 Utilisation de MobaXterm

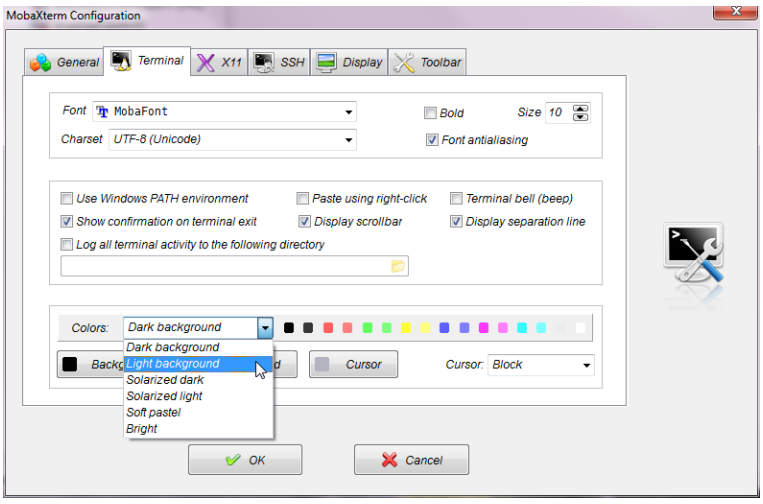
Lancer l'application à l'aide du raccourci qui a dû être créé sur le bureau ou depuis le dossier **MobaXterm Personal Edition** du menu Démarrer (dans le sous-menu Programmes ou Tous les programmes suivant votre version de Windows).
Note à propos du pare-feu : l'application disposant de fonctionnalités réseau, Windows réclamera probablement votre accord pour l'autoriser à accéder au réseau. Donner votre autorisation.

L'interface du logiciel est organisée en 4 zones
- Un menu
- Une barre de boutons donnant un accès rapide à certaines fonctions
- Une zone de création de sessions
- Une zone d'affichage des terminaux



❶ Configuration du logiciel

Cliquer sur le bouton **Settings** pour accéder aux configurations de l'application.



L'onglet **Terminal** permet d'ajuster l'aspect des terminaux. Les boutons Foreground color et Background color permettent de modifier respectivement la couleur des caractères et du fond du terminal. Des jeux de couleurs préconfigurés sont également proposés dans le menu **Colors**. Choisir par exemple Light Background pour une meilleure lisibilité.

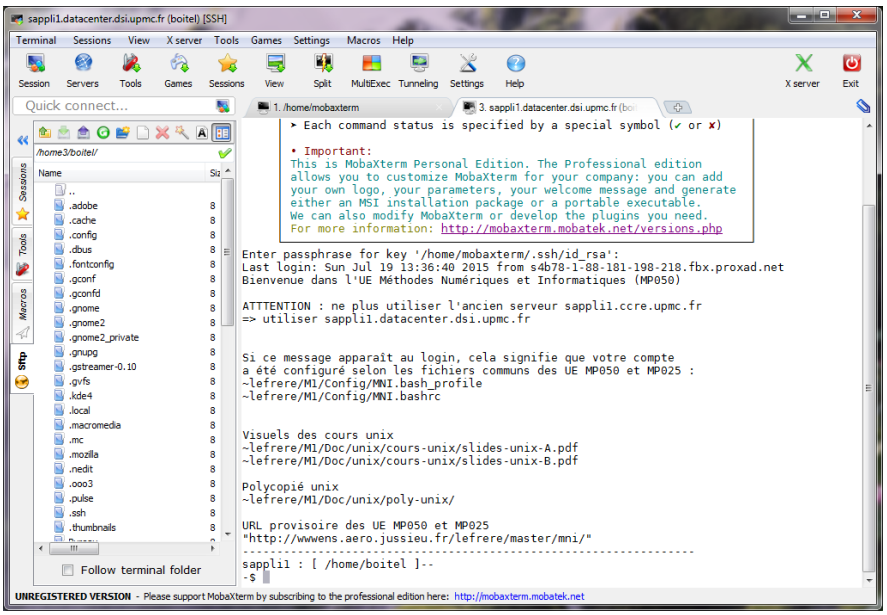
Dans le menu **Charset** choisir UTF-8 (UNICODE) qui est la dernière norme de codage de caractères.
L'onglet **X11** donne accès aux réglages du gestionnaire de fenêtres qui permet l'affichage d'applications graphiques interactives en provenance d'un serveur Unix. Le gestionnaire est activé par défaut mais peut être contrôlé par l'option **Automatically start X server at MobaXterm start up**. Se reporter à la partie ❹ pour avoir des détails sur l'intérêt du gestionnaire de fenêtres.

❷ Les terminaux

C'est dans cette zone que les commandes Unix sont tapées que ce soit localement c'est-à-dire sans connexion préalable ou après connexion à un serveur. Dans le premier cas les dossiers et fichiers visibles sont ceux de votre dossier personnel sur votre pc et dans le second ce sont ceux de votre répertoire d'accueil sur le serveur.
Le mode plein écran permet au terminal en cours d'utilisation d'occuper tout l'écran de votre ordinateur. Pour accéder à ce mode cliquer dans le terminal avec le bouton droit de la souris et choisir **Fullscreen** dans le menu qui apparaît. Répéter l'opération pour revenir au mode normal.
Astuce : pour que le terminal local s'ouvre sur un dossier précis, faire un clic droit avec la souris sur le dossier et choisir **Open MobaXterm terminal here**.

❸ Ouverture d'une session sur le serveur sappli et transfert de fichiers

Cliquer sur le bouton **Session**
Choisir le protocole SSH en cliquant sur le bouton **SSH** (en haut à gauche) puis indiquer dans le champ **Remote host** le serveur à contacter ; ici `sappli1.datacenter.dsi.upmc.fr`. Enfin indiquer votre identifiant dans le champ **Specify username**. Valider la fenêtre en cliquant sur **OK** pour ouvrir la connexion.

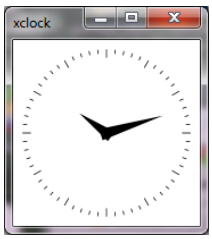


Une fois la connexion établie un nouvel onglet est créé dans la zone des terminaux. En parallèle un onglet **Sftp** doit apparaître dans la partie gauche de l'application. Cette nouvelle zone donne un accès graphique aux dossiers et aux fichiers se trouvant sur le serveur. Un glisser/déposer de dossiers ou de fichiers depuis ou vers cette zone permet respectivement de télécharger des fichiers depuis le serveur ou de déposer des fichiers sur le serveur. Pour clore une session, cliquer sur la croix présente dans son onglet ou taper la commande exit.

Note : Une fois la connexion réussie, les paramètres d'une session sont mémorisés dans l'onglet **Session** sous le nom du serveur. Un double-clic sur ce nom permet de rouvrir automatiquement la session. Il est également possible de modifier les paramètres de la session en cliquant avec le bouton droit de la souris sur le nom de la session et en choisissant **Edit Session** dans le menu qui apparaît.

❹ Le protocole graphique X11

Le protocole X11 prend en charge l'affichage d'applications fenêtrées, le plus souvent interactives, fonctionnant sur un serveur de type Unix. Ce protocole n'est donc utile qu'une fois connecté à un serveur comme sappli. Dans la session ouverte au point ❸ il est possible de tester le fonctionnement du protocole X11 en tapant simplement la commande **`xclock`** (valider avec la touche Entrée).



La fenêtre ci-contre doit apparaître. Si le message d'erreur **`X connection to localhost:11.0 broken (explicit kill or server shutdown)`** s'affiche dans le terminal, vérifier que le protocole X11 est activé (logo X en vert en haut à droite de la fenêtre). Si ce n'est pas le cas cliquer dessus, attendre quelques secondes son activation et relancer la commande xclock.
Le principal intérêt est de pouvoir accéder aux applications de l'environnement graphique des systèmes Unix utilisés pendant les séances de TP, comme les éditeurs de texte.

Note : Le protocole X11 est gourmand en débit réseau et peut être inutilisable sur un réseau à faible débit.

Logiciels complémentaires

Le logiciel **Notepad++** (http://notepad-plus-plus.org/fr/) est un éditeur très pratique pour l'édition de codes en langage C ou FORTRAN. Il intègre entre autre chose la coloration syntaxique des mots-clés de ces langages.

Gedit est un autre éditeur disponible en standard dans l'environnement Gnome de Unix. Il dispose des mêmes fonctionnalités que Notepad++ tout en proposant une interface plus simple et intuitive. Contrairement à Notepad++ il est disponible dans l'environnement Unix utilisé pour les TPs. Gedit est téléchargeable pour Windows à l'adresse https://wiki.gnome.org/Apps/Gedit/ (choisir le menu Download puis Windows binaries. Télécharger l'exécutable **gedit-setup….exe** correspondant à la version la plus récente).

Annexe B : Bref guide d'installation de Cygwin

Ce chapitre a été rédigé par Frédéric Meynadier (Frederic.Meynadier@obspm.fr).

B.1 Introduction

Cygwin est un ensemble de programmes permettant d'émuler, dans une certaine mesure, un environnement linux sous windows. Il ne nécessite aucun partitionnement ou modification du système windows, c'est une couche supplémentaire qui tourne par dessus. Les performances sont donc moins bonnes, et toutes les fonctionnalités d'un système Unix ne sont pas reproduites.

Il s'agit néanmoins d'un choix très judicieux si on souhaite disposer d'un environnement linux de base sans modification lourde de son PC windows, et sans se connecter à distance en permanence. Il faut cependant disposer d'une connexion internet haut débit pour télécharger les fichiers assez volumineux qui composent ces programmes.

Le guide d'installation pas à pas qui suit a été rédigé pour la version 1.5.18-1 et mis à jour pour la version 1.7.30. À titre indicatif, l'espace disque nécessaire est de l'ordre de 500 Mo, et l'installation prend environ 30 mn sur un système à 2.6 GHz équipé d'une connexion haut débit ($>$ 2 Mo/s).

B.2 Installation des composants de base

Allez sur la page web d'accueil de Cygwin :
`www.cygwin.com`
Pour une première installation, suivre le lien `Install` ou pour une mise à jour ou des compléments, suivre le lien `Update`. Ils aboutissent tous les deux à la page :
`www.cygwin.com/install.html`
Avant de continuer, il faut savoir si on dispose d'un système Windows 32 bits ou 64 bits : selon le cas, on doit télécharger le fichier `setup-x86.exe` ou `setup-x86_64.exe` (cas des systèmes les plus récents). On récupère ainsi un fichier exécutable windows qui permettra d'installer les différents composants de Cygwin, que ce soit pour la première installation ou, ultérieurement, pour l'ajout de composants. Une fois téléchargé, exécuter ce fichier.

Trois possibilités sont offertes : a priori `Install from Internet` est celle qu'il vous faut. Ce choix demande à Cygwin de télécharger puis d'installer les fichiers que vous demanderez.

À l'écran suivant, le "Root Directory" est le point de votre disque dur qui sera, plus tard, la racine (`/`) de votre système de fichiers cygwin. Le choix par défaut, `C:\cygwin`, est recommandé. Il est conseillé de laisser les autres options telles que recommandées, sauf si on sait ce qu'on fait...

L'écran suivant demande le "Local Package Directory", c'est là qu'il stocke les fichiers compressés des composants qui seront installés. Par défaut c'est le répertoire où a été téléchargé `setup-x86_64.exe`, aussi modifiez le tel que vous le souhaitez, par exemple `C:\cygwin_packages`

A priori, si vous n'utilisez pas de proxy, choisissez ensuite "Direct Connection", puis sélectionnez l'un des serveurs `ftp` abritant une copie des fichiers.

Une liste de composants s'affiche, classée par thème. Développez l'arborescence pour connaître le contenu des thèmes. Seront installés ceux qui ont un numéro de version, tandis que les autres sont ignorés ("`skip`"). Cliquez sur "`skip`" pour sélectionner d'autres paquetages à installer. Si un paquetage en nécessite d'autres, ils seront sélectionnés automatiquement (par exemple, `gcc-g++` nécessite `gcc-core`). Parmi les composants qui seront le plus susceptibles de vous intéresser et qui ne sont pas installés par défaut :

— catégorie `devel : gcc-core, gcc-g++, gcc-gfortran, make`

— catégorie `editors : emacs, xemacs, vim, nedit`

À noter qu'il est également possible d'installer LaTeX, aujourd'hui disponible dans la distribution `texlive`.
Une fois le choix fait, appuyez sur "`suivant`". Les éventuelles « dépendances » supplémentaires sont signalées, acceptez leur installation et continuez.
Une fois l'installation effectuée vous disposez d'un terminal texte sous `bash`, comparable à celui que vous ouvrez sur les stations linux.

B.3 Installation d'un serveur X

Si vous souhaitez profiter des capacités graphiques de certains programmes ainsi que de la transmission de fenêtres à partir de machines distantes, vous devez installer le paquetage `xorg-x11-base`, dans la catégorie `X11`. Il est possible que ce paquetage ait été sélectionné par défaut au moment de l'installation. Si nécessaire, relancez `setup-x86_64.exe` et sélectionnez `xorg-x11-base`. S'il est déjà installé, son numéro de version et `Keep` sont affichés.

Le démarrage du serveur X se fait par l'intermédiaire d'un fichier appelé `startxwin.bat` (qui fait partie du paquetage `X-startup-scripts`, normalement installé avec le reste). Il se trouve dans `/usr/X11R6/bin/startxwin.bat` où `/` est probablement `C:\cygwin` si vous avez gardé le nom par défaut tout à l'heure). Vous pouvez créer un raccourci vers ce fichier pour l'atteindre plus rapidement, par exemple à partir du bureau. Selon la version du programme d'installation, il est possible que ce raccourci ait été automatiquement créé.

À la première utilisation (double clic sur le raccourci ou sur le programme `startxwin.bat`), votre pare-feu risque de vous demander si ce programme a le droit de communiquer avec l'extérieur - répondez "oui" en vous arrangeant pour qu'il retienne la réponse (en général, une case à cocher).

Le terminal qui s'ouvre alors est géré par le serveur X (à la différence du précédent, qui était géré par Windows). Vous pouvez maintenant utiliser des programmes nécessitant des fenêtres graphiques (`gv`, `xpdf`, `gnuplot`, etc... à condition qu'ils soient installés !)

À noter : même si tous les terminaux sont fermés, le serveur X continue de tourner jusqu'au reboot suivant (un "X" apparaît dans la liste des programmes en train de tourner, en bas à droite). Un clic droit sur l'icône offre la possibilité de fermer le serveur.

B.4 Connexion à une machine distante

La connexion aux machines distantes se fera préférentiellement par shell sécurisé (`ssh`), qui permet d'éviter de faire passer le mot de passe en clair sur le réseau. Pour cela, installer le paquetage `openssh` se trouvant dans la catégorie `Net`.

Une fois installé, l'accès se fait par la commande :
`ssh login@sappli1.datacenter.dsi.upmc.fr`
où `login` doit être remplacé par votre login.

B.5 Transfert de fichiers

Utiliser la commande `scp` (fournie avec `ssh`) dont la syntaxe est la suivante :
`scp mon_fichier login@sappli1.datacenter.dsi.upmc.fr:rep_destination`
où `rep_destination` (optionnel) est un sous-répertoire de votre répertoire personnel d'accueil. Vous pouvez l'omettre, et dans ce cas le fichier se trouvera sur la machine distante dans le répertoire d'accueil, mais n'oubliez pas « : » à la fin, sinon le fichier sera simplement copié en local dans un fichier de même nom que la machine distante... Évidemment ça marche aussi dans l'autre sens :
`scp login@sappli1.datacenter.dsi.upmc.fr:repertoire/mon_fichier .`
Il existe également un outil purement windows proposant une interface souris pour ce protocole (`http://winscp.net`)

B.6 Installation de programmes ne figurant pas dans la liste

En dehors de la liste de programmes installables à partir de `setup.exe`, il existe des programmes ne faisant pas partie de la distribution standard, mais dont les sources ont été compilées sous cygwin, et qui peuvent donc être installés.

C'est, par exemple, le cas du compilateur Fortran `g95`, dont les exécutables pré-compilés pour cygwin peuvent être trouvés à l'adresse : http://www.g95.org/. Ils se présentent sous la forme d'un tar.gz à décompresser à partir de la racine (`C:\cygwin`, normalement). Pour ce faire, il suffit de télécharger le fichier tar.gz dans ce répertoire, puis ouvrir un terminal cygwin et procéder comme sous unix :
`cd /; tar -xzvf [fichier].tar.gz`
`g95` est alors immédiatement disponible.

Annexe C : Types de fichiers, suffixes et outils associés

Cette page présente quelques exemples de types de fichiers et les suffixes (extensions) habituellement utilisés pour ces fichiers. On distingue :

— les fichiers texte, que l'on peut examiner et modifier avec les éditeurs de texte (`vi`, `emacs`, `gedit`, ...), dont certains peuvent être utilisés par des outils dédiés ;

— les fichiers binaires pour lesquels des outils spécifiques sont nécessaires.

Fichiers texte

`.txt`	fichier texte	
`.c`	fichier source en langage C	`gcc` (compilation)
`.h`	fichier texte (entête pour préprocesseur)	`cpp` (préprocesseur)
`.f`	fichier source en fortran	`gfortran`, `g95` (compilation)
`.f90`	fichier source en fortran 90	`gfortran`, `g95` (compilation)
`.pl`	fichier source perl	`perl`, interpréteur
`.py`	fichier source python	`python`, interpréteur
`.tex`	fichier texte (source LaTeX)	`latex`, `pdflatex` (compilation)
`.ps`	Adobe Postscript document (text/graphics)	`gv, ghostview` (visualisation)
`.eps`	fichier postscript encapsulé	`gv, ghostview` (visualisation)
`.fig`	fichier de dessin au format texte `xfig`	`xfig` (création), `fig2dev` (conversion)
`.html`	HyperText Markup Language (texte)	`firefox, opera, ...` (affichage)
`.xml`	Extensible Markup Language (texte)	

Fichiers binaires

`.dvi`	Device Independent File Format (TeX)	`xdvi`
`.gif`	Bitmap graphics (Graphics Interchange Format)	`xv`
`.png`	Portable Network Graphics	`xv`
`.pbm`	Portable Bit Map P4 (image noir et blanc)	`xv`, `convert`
`.pgm`	Portable Gray Map P5 (image en niveaux de gris)	`xv`, `convert`
`.ppm`	Portable Pixel Map P6 (image en couleur)	`xv`, `convert`
`.jpg`	Bitmap graphics (JPEG Joint Photography Experts Group)	`xv`
`.mod`	fichier de module fortran	`gfortran`, `g95, ...`
`.o`	fichier objet (binaire)	`ld`
`.a`	fichier bibliothèque (binaire)	`ld`, `ar`
`.so`	fichier bibliothèque dynamique partagée (binaire)	`ld`
`.tar`	tar file archive (binaire)	`tar`
`.gz`	fichier compressé (Lempel-Ziv)	`gzip, gunzip`
`.bz2`	fichier compressé (Burrows-Wheeler)	`bzip2, bunzip2`
`.lzma`	fichier compressé (LZMA : Lempel-Ziv-Markov chain-Algorithm)	`unlzma, xz`
`.pdf`	Portable Document Format	`xpdf`, `evince`, `acroread`, ...

Annexe D : Structures de contrôle dans différents langages

Correspondance des structures de contrôle dans différents langages

shell (sh)	**fortran 90**	**c**	**python** (indentation requise)

`if` commande
`then`
  bloc de cmdes
`else`
  bloc de cmdes
`fi`
	`if (`expr. log.`) then`
	  bloc
	`else`
	  bloc
	`endif`
		`if (`expr. log.`) {`
		  bloc
		`} else {`
		  bloc
		`}`
			`if` expr. log. `:`
			  bloc de cmdes
			`else`
			  bloc de cmdes

`case $`variable `in`
  motif)
  bloc de cmdes
  `;;`
  `*)`
  bloc de cmdes
  `;;`
`esac`
	`select case (`expr.`)`
	  `case ('`sélecteur`')`
	  bloc
	  `case (default)`
	  bloc
	`end select`
		`switch (` expr. entière`) {`
		  `case` sélecteur `:`
		  bloc
		  `break;`
		  `default :`
		  bloc
		`}`
			pas de structure `case`
			
			utiliser `if`, `elif`, `else`

`for` variable `in` liste de mots
`do`
  bloc de cmdes
`done`
	`do` entier `=` début`,` fin[`,` pas]	`for (`expr_1`;` expr_2`;` expr_3`) {`
		  bloc
		`}`
			`for` variable `in` liste `:`
			  bloc de cmdes

`while` commande
`do`
  bloc de cmdes
`done`
	`do while (`expr. log.`)`
	  bloc
	`end do`
		`while (`expr. log.`) {`
		  bloc
		`}`
			`while` expr. log. `:`
			  bloc de cmdes

`continue`	`cycle`	`continue;`	`continue`
`break`	`exit`	`break;`	`break`
	`go to` étiquette-numér.	`goto` étiquette`;`	
`exit`	`return`	`return` expr. `;`	`return` expr.
	`stop`	`exit;`	`exit()` ou `quit()`

Aide mémoire **vi**

Modes

Vi a deux modes : le mode insertion et le mode commande. L'édition débute en mode commande, où s'effectuent les mouvements du curseur, la destruction et le déplacement de texte. Les commandes d'insertion ou de changement permettent d'entrer dans le mode insertion. [ESC] ramène l'éditeur dans le mode commande (d'où l'on peut quitter, par exemple en frappant `:q!`). La plupart des commandes s'exécutent dès qu'on les frappe, sauf les commandes précédées du « : » qui attendent le retour chariot pour s'exécuter.

Sortie

sortie avec sauvegarde des modifications	`:x`
sortie (sauf si modifications)	`:q`
sortie (forcée, sans sauvegarde des modifications)	`:q!`

Insertion de texte

insère avant le curseur, avant la ligne	`i , I`
ajoute après le curseur, après la ligne	`a , A`
ouvre une nouvelle ligne après, avant	`o , O`
remplace un caractère, plusieurs caractères	`r , R`

Déplacements

à gauche, vers le bas, vers le haut, à droite	`h , j , k , l`
mot suivant, mot délimité par un espace	`w , W`
début de mot, de mot délimité par un espace	`b , B`
fin de mot, de mot délimité par un espace	`e , E`
phrase précédente, suivante	`( , )`
paragraphe précédent, suivant	`{ , }`
début, fin de ligne	`0 , $`
début, fin de fichier	`1G , G`
ligne *n*	*n*`G` ou `:`*n*
caractère `c` suivant, précédent	`fc , Fc`
haut, milieu, bas de l'écran	`H , M , L`

Destruction de texte

Presque toutes les commandes de suppression sont accomplies en frappant `d` suivi d'un *déplacement*. Par exemple, `dw` supprime un mot. Voici quelques autres commandes de suppression :

caractère à droite, à gauche	`x , X`
jusqu'à la fin de la ligne	`D`
la ligne	`dd`
la ligne	`:d`

Sélection de texte

Comme celles de suppression, les commandes de sélection sont obtenues en frappant `y` suivi d'une commande de *déplacement*. Par exemple, `y$` sélectionne jusqu'à la fin de la ligne. Voici deux autres commandes de sélection :

ligne	`yy`
ligne	`:y`

Modification de texte

Les commandes de modification sont des suppressions qui placent l'éditeur dans le mode insertion. Elles sont obtenues en frappant `c` suivi d'une commande de *déplacement*. Par exemple `cw` modifie un mot. Voici d'autres commandes de modification :

jusqu'à la fin de la ligne	`C`
la ligne courante	`cc`

Replacer un texte sélectionné

placer après le pointeur ou après la ligne	`p`
placer avant le pointeur ou avant la ligne	`P`

Mémoires nommées

Des mémoires tampons nommées peuvent être utilisées pour toutes les commandes de suppression, modification, sélection ou collage. Il suffit de faire précéder ces commandes par `"`*c* où *c* désigne une lettre minuscule quelconque. Par exemple, `"adw` supprime un mot et le mémorise dans le tampon `a`. Ce mot peut par la suite être rappelé avec une commande de collage appropriée, par exemple `"ap`.

Pointeurs

On peut placer des pointeurs nommés sur n'importe quelle ligne. Toute lettre minuscule peut servir de nom pour un pointeur. Les pointeurs peuvent aussi être utilisés pour délimiter une zone.

poser le pointeur *c* à cette ligne	`m`*c*
aller au pointeur *c*	`'`*c*
aller au 1er caractère non blanc de la ligne du pointeur *c*	`'`*c*

Recherche de chaînes

recherche après	*/string*
recherche avant	*?string*
recherche répétée dans le même sens, en sens inverse	`n , N`

Remplacement

La fonction de recherche et substitution est obtenue grâce à la commande `:`. On l'utilise souvent combinée avec des intervalles ou la commande `:g` (cf. plus bas).

remplace le motif par la chaîne	`:s/`*motif*`/`*chaîne*`/`*options*
options : toutes sur chaque ligne, confirmer chaque fois	`g , c`
recommencer le dernier remplacement `:s`	`&`

Expressions Rationnelles

un caractère quelconque, sauf le retour ligne	`.` (point)
zéro, un ou plus d'une occurrence de	`*`
un caractère quelconque de l'ensemble	`[... ]`
un caractère quelconque hors de l'ensemble	`[^ ... ]`
début, fin de ligne	`^ , $`
début, fin de mot	`\< , \>`
groupement	`\(...\)`
contenu du *n*ième groupe	`\`*n*

Multiplicateurs

Presque toutes les commandes peuvent êtres précédées par un nombre qui indique le nombre de fois qu'elle doit être appliquée. Par exemple, `5dw` détruira 5 mots et `3fe` avancera le curseur jusqu'à la 3e occurrence de la lettre `e`. Même les insertions peuvent être répétées selon cette méthode, par exemple pour insérer la même ligne 100 fois.

Intervalles de lignes

On peut faire précéder la plupart des commandes suivant le deux-points par un intervalle qui indique la ou les lignes où elles s'exécutent. Par exemple, `:3,7d` détruit les lignes 3 à 7. On combine souvent les intervalles d'adressage avec la commande `:s` pour effectuer une substitution sur plusieurs lignes, comme `:.,$s/pattern/string/g` pour substituer de la ligne courante à la dernière ligne.

lignes *n*-*m*	`:`*n*`,`*m*
ligne courante	`:.`
dernière ligne	`:$`
pointeur *c*	`:'`*c*
toutes les lignes	`:%`
toutes les lignes contenant le motif	`:g/`*pattern*`/`

Fichiers

écrit fichier (le fichier courant si aucun nom n'est donné	`:w` *fichier*
lit fichier après la ligne	`:r` *fichier*
fichier suivant	`:n`
fichier précédent	`:p`
édite fichier	`:e` *fichier*
remplace la ligne par la sortie de programme	`!!`*programme*

Divers

échange majuscule et minuscule	`~`
colle deux lignes	`J`
répète la dernière commande de modification de texte	`.`
annule la dernière modification	`u`
annule toutes les modifications sur la ligne	`U`

Vi Reference Card

Modes

Vi has two modes : insertion mode, and command mode. The editor begins in command mode, where cursor movement and text deletion and pasting occur. Insertion mode begins upon entering an insertion or change command. [ESC] returns the editor to command mode (where you can quit, for example by typing `:q!`). Most commands execute as soon as you type them except for "colon" commands which execute when you press the return key.

Quitting

exit, saving changes	`:x`
quit (unless changes)	`:q`
quit (force, even if unsaved)	`:q!`

Inserting text

insert before cursor, before line	`i , I`
append after cursor, after line	`a , A`
open new line after, line before	`o , O`
replace one char, many chars	`r , R`

Motion

left, down, up, right	`h , j , k , l`
next word, blank delimited word	`w , W`
beginning of word, of blank delimited word	`b , B`
end of word, of blank delimited word	`e , E`
sentence back, forward	`( , )`
paragraph back, forward	`{ , }`
beginning, end of line	`0 , $`
beginning, end of file	`1G , G`
line *n*	*n*`G` or `:`*n*
forward, back to char *c*	`f`*c* `, F`*c*
top, middle, bottom of screen	`H , M , L`

Deleting text

Almost all deletion commands are performed by typing `d` followed by a *motion*. For example `dw` deletes a word. A few other deletions are :

character to right, left	`x , X`
to end of line	`D`
line	`dd`
line	`:d`

Yanking text

Like deletion, almost all yank commands are performed by typing `y` followed by a *motion*. For example `y$` yanks to the end of line. Two other yank commands are :

line	`yy`
line	`:y`

Changing text

The change command is a deletion command that leaves the editor in insert mode. It is performed by typing `c` followed by a *motion*. For example `cw` changes a word. A few other change commands are :

to end of line	`C`
line	`cc`

Putting text

put after position or after line	`p`
put before position or before line	`P`

Buffers

Named buffers may be specified before any deletion, change, yank, or put command. The general prefix has the form `"`*c* where *c* may be any lower case letter. For example, `"adw` deletes a word into buffer `a`. It may thereafter be put back into the text with an appropriate put command, for example `"ap`.

Markers

Named markers may be set on any line of a file. Any lower case letter may be a marker name. Markers may also be used as the limits for ranges.

set marker *c* on this line	`m`*c*
goto marker *c*	`‘`*c*
goto marker *c* first non-blank	`’`*c*

Search for Strings

search forward	*/string*
search backward	*?string*
repeat search in same, reverse direction	`n , N`

Replace

The search and replace function is accomplished with the `:s` command. It is commonly used in combination with ranges or the `:g` command (below).

replace pattern with string	`:s/`*pattern*`/`*string*`/`*flags*
flags : all on each line, confirm each	`g , c`
repeat last `:s` command	`&`

Regular Expressions

any single character except newline	`.` (dot)
zero or more repeats	`*`
any character in set	`[... ]`
any character not in set	`[^ ... ]`
beginning, end of line	`^ , $`
beginning, end of word	`\< , \>`
grouping	`\(...\)`
contents of *n*th grouping	`\`*n*

Counts

Nearly every command may be preceded by a number that specifies how many times it is to be performed. For example `5dw` will delete 5 words and `3fe` will move the cursor forward to the 3rd occurance of the letter `e`. Even insertions may be repeated conveniently with this method, say to insert the same line 100 times.

Ranges

Ranges may precede most "colon" commands and cause them to be executed on a line or lines. For example `:3,7d` would delete lines 3–7. Ranges are commonly combined with the `:s` command to perform a replacement on several lines, as with `:.,$s/pattern/string/g` to make a replacement from the current line to the end of the file.

lines *n*–*m*	`:`*n*`,`*m*
current line	`:.`
last line	`:$`
marker *c*	`:’`*c*
all lines	`:%`
all matching lines	`:g/`*pattern*`/`

Files

write file (current file if no name given)	`:w` *file*
read file after line	`:r` *file*
next file	`:n`
previous file	`:p`
edit file	`:e` *file*
replace line with program output	`!!`*program*

Other

toggle upper/lower case	`~`
join lines	`J`
repeat last text-changing command	`.`
undo last change, all changes on line	`u , U`

Bref aide mémoire **emacs**

Consulter l'aide mémoire en anglais pour plus d'informations

IMPORTANT

`C-` veut dire (appuyer sur) la clé [Control]

`M-` veut dire la clé [Méta] (clé [Alt] sur la plupart des claviers)

`C-g` annule l'opération ou la commande emacs en cours

Le mini-buffer

Le mini-buffer est le mini-cadre en bas d'emacs. Toutes les fonctions emacs sont exécutées à partir de ce cadre et s'y déroulent.

Exécuter	`RET`
Auto-complétion	`TAB`
Montrer les combinaisons possibles	`?`
Navigation dans l'historique des commandes	`UP/DOWN`

Les modes

Les modes sous emacs désignent la façon d'effectuer certaines opérations comme l'indentation. Par exemple, dès qu'un fichier ouvert a pour extension `.f90`, le mode Fortran est activé. Il est donc important de nommer correctement les fichiers de travail afin d'activer le bon mode à leur ouverture.

Gestion des buffers

Emacs permet l'ouverture de plusieurs fichiers à la fois. Chaque fichier est alors contenu dans un « buffer », qui est désigné par le nom du fichier. Les buffers ne sont pas toujours rattachés à un fichier (peut être un shell, une fenêtre d'exploration, ...). La ligne de mode, juste au-dessus du mini-buffer, apporte divers renseignements sur le buffer (état, nom, mode, position du curseur)

Choisir un autre buffer	`C-x b`
Lister tous les buffers ouverts	`C-x C-b`
Fermer un buffer	`C-x k`

Navigation et gestion des fichiers

Ouvrir un fichier (existant ou non)	`C-x C-f`
Ouvrir un répertoire existant (dans un buffer d'exploration)	`C-x d`
Enregistrer un fichier	`C-x C-s`
Enregistrer un fichier sous un autre nom	`C-x C-w`
Enregistrer tous les fichiers	`C-x s`

Gestion des cadres

Une fenêtre emacs peut contenir un ou plusieurs cadres, chacun d'eux permettant d'afficher un buffer différent.

Fermer le cadre courant	`C-x 0`
Étendre le cadre courant	`C-x 1`
Diviser le cadre en 2 horizontalement	`C-x 2`
Diviser le cadre en 2 verticalement	`C-x 3`

Déplacement

Vers le prochain mot (en arrière / en avant)	`M-b / M-f`
Vers la prochaine ligne (en arrière / en avant)	`C-p / C-n`
Vers le début / la fin de la ligne courante	`C-a / C-e`
Vers le début / la fin de la fonction courante	`C-M-a / C-M-e`
Vers le début / la fin du buffer	`M-< / M->`
Vers la page de gauche / page de droite	`C-x < / C-x >`

Édition

Annuler la dernière action	`C-_`
Couper le prochain mot	`M-d`
Couper la fin de la ligne courante	`C-k`
Échanger deux caractères successifs	`C-t`
Échanger deux lignes successives	`C-x C-t`
Marquer le début de sélection	`C-ESPACE`
Couper la sélection	`C-w`
Copier la sélection	`M-w`
Coller la sélection	`C-y`
Couper la sélection rectangulairement	`C-x r k`
Coller la sélection rectangulairement	`C-x r y`
Remplir un rectangle avec une chaîne	`C-x r t`

Recherche et remplacement

Recherche en avant / en arrière	`C-s / C-r`
Recherche selon expression régulière en avant	`C-M-s`
Recherche selon expression régulière en arrière	`C-M-r`
Remplacement interactif	`M-%`

Indentation

L'indentation est automatique sous emacs dans la quasi-totalité des langages, à condition de se trouver dans le bon mode.

Indente automatiquement la ligne courante	`TAB`
Indente la sélection	`C-M-\`
Indente le buffer	`C-M-q`
Tabulation sur la sélection	`C-x TAB`

Changement de casse

Mettre une sélection en majuscule	`C-x C-u`
Mettre une sélection en minuscule	`C-x C-l`
Mettre une sélection en capitale	`M-x capitalize-region`

Shell

Il est possible d'exécuter des commandes shell sous emacs, voire d'ouvrir un buffer se comportant comme un terminal shell habituel. Pour ouvrir plusieurs shells sous emacs, il est nécessaire de les renommer (fonction `rename-buffer`).

Exécuter une commande shell	`M-!`
Exécuter une commande shell sur un fichier/dossier (dans un buffer d'exploration)	`!`
Ouvrir un buffer shell	`M-x shell`
Commande précédente de l'historique dans le buffer shell	`M-p`
Commande suivante de l'historique dans le buffer shell	`M-n`

Divers

- Une multitude de fonctions internes propres à emacs est disponible à partir du mini-buffer.
  Appelle une fonction emacs `M-x nom_de_fonction`
  Décrit la fonction `C-h f nom_de_fonction`
- Le manuel peut s'ouvrir directement dans emacs par la fonction `man`.
- Le fichier `~/.emacs` permet de configurer l'emacs de chaque utilisateur, comme par exemple l'activation de la colorisation automatique, la surbrillance des parenthèses, des raccourcis... Certaines distributions comme Mandriva fournissent un fichier `.emacs` déjà très complet.
- Il est possible de sauver l'état d'emacs automatiquement, pour l'obtenir à l'identique à sa prochaine ouverture. Il suffit pour cela d'effectuer la fonction `desktop-save` (disponible selon le `.emacs` de l'utilisateur).
- Pour utiliser emacs directement dans le terminal (au lieu d'ouvrir une nouvelle fenêtre), lancer emacs par la commande `emacs -nw`

Plus de documentations sur emacs

`http://www.emacswiki.org` (voir la version en français)

`http://www.gnu.org/software/emacs/manual/emacs.html`

GNU Emacs Reference Card (1/2)

(for version 24)

Starting Emacs

To enter GNU Emacs 24, just type its name: `emacs`

Leaving Emacs

suspend Emacs (or iconify it under X)	`C-z`
exit Emacs permanently	`C-x C-c`

Files

read a file into Emacs	`C-x C-f`
save a file back to disk	`C-x C-s`
save **all** files	`C-x s`
insert contents of another file into this buffer	`C-x i`
replace this file with the file you really want	`C-x C-v`
write buffer to a specified file	`C-x C-w`
toggle read-only status of buffer	`C-x C-q`

Getting Help

The help system is simple. Type `C-h` (or `F1`) and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**.

remove help window	`C-x 1`
scroll help window	`C-M-v`
apropos: show commands matching a string	`C-h a`
describe the function a key runs	`C-h k`
describe a function	`C-h f`
get mode-specific information	`C-h m`

Error Recovery

abort partially typed or executing command	`C-g`
recover files lost by a system crash	`M-x recover-session`
undo an unwanted change	`C-x u`, `C-_` or `C-/`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`
reverse regular expression search	`C-M-r`
select previous search string	`M-p`
select next later search string	`M-n`
exit incremental search	`RET`
undo effect of last character	`DEL`
abort current search	`C-g`

Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not matched.

Motion

entity to move over	**backward**	**forward**
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-{`	`M-}`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`
scroll current line to center, top, bottom	`C-l`
goto line	`M-g g`
goto char	`M-g c`
back to indentation	`M-m`

Killing and Deleting

entity to kill	**backward**	**forward**
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill **region**	`C-w`
copy region to kill ring	`M-w`
kill through next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* **words** away	`M-@`
mark **paragraph**	`M-h`
mark **page**	`C-x C-p`
mark **sexp**	`C-M-@`
mark **function**	`C-M-h`
mark entire **buffer**	`C-x h`

Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	`SPC` or `y`
replace this one, don't move	`,`
skip to next without replacing	`DEL` or `n`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	`RET`
enter recursive edit (`C-M-c` to exit)	`C-r`

Multiple Windows

When two commands are shown, the second is a similar command for a frame instead of a window.

delete all other windows	`C-x 1`	`C-x 5 1`
split window, above and below	`C-x 2`	`C-x 5 2`
delete this window	`C-x 0`	`C-x 5 0`
split window, side by side	`C-x 3`	
scroll other window	`C-M-v`	
switch cursor to another window	`C-x o`	`C-x 5 o`
select buffer in other window	`C-x 4 b`	`C-x 5 b`
display buffer in other window	`C-x 4 C-o`	`C-x 5 C-o`
find file in other window	`C-x 4 f`	`C-x 5 f`
find file read-only in other window	`C-x 4 r`	`C-x 5 r`
run Dired in other window	`C-x 4 d`	`C-x 5 d`
find tag in other window	`C-x 4 .`	`C-x 5 .`
grow window taller	`C-x ^`	
shrink window narrower	`C-x {`	
grow window wider	`C-x }`	

Formatting

indent current **line** (mode-dependent)	`TAB`
indent **region** (mode-dependent)	`C-M-\`
indent **sexp** (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
indent for comment	`M-;`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
join line with previous (with arg, next)	`M-^`
delete all white space around point	`M-\`
put exactly one space at point	`M-SPC`
fill paragraph	`M-q`
set fill column to *arg*	`C-x f`
set prefix each line starts with	`C-x .`
set face	`M-o`

Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
fetch previous minibuffer input	`M-p`
fetch later minibuffer input or default	`M-n`
regexp search backward through history	`M-r`
regexp search forward through history	`M-s`
abort command	`C-g`

Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. Type `F10` to activate menu bar items on text terminals.

GNU Emacs Reference Card (2/2)

Buffers

select another buffer	`C-x b`
list all buffers	`C-x C-b`
kill a buffer	`C-x k`

Transposing

transpose **characters**	`C-t`
transpose **words**	`M-t`
transpose **lines**	`C-x C-t`
transpose **sexps**	`C-M-t`

Spelling Check

check spelling of current word	`M-$`
check spelling of all words in region	`M-x ispell-region`
check spelling of entire buffer	`M-x ispell-buffer`
toggle on-the-fly spell checking	`M-x flyspell-mode`

Tags

find a tag (a definition)	`M-.`
find next occurrence of tag	`C-u M-.`
specify a new tags file	`M-x visit-tags-table`
regexp search on all files in tags table	`M-x tags-search`
run query-replace on all the files	`M-x tags-query-replace`
continue last tags search or query-replace	`M-,`

Shells

execute a shell command	`M-!`
execute a shell command asynchronously	`M-&`
run a shell command on the region	`M-|`
filter region through a shell command	`C-u M-|`
start a shell in window `*shell*`	`M-x shell`

Rectangles

copy rectangle to register	`C-x r r`
kill rectangle	`C-x r k`
yank rectangle	`C-x r y`
open rectangle, shifting text right	`C-x r o`
blank out rectangle	`C-x r c`
prefix each line with a string	`C-x r t`

Abbrevs

add global abbrev	`C-x a g`
add mode-local abbrev	`C-x a l`
add global expansion for this abbrev	`C-x a i g`
add mode-local expansion for this abbrev	`C-x a i l`
explicitly expand abbrev	`C-x a e`
expand previous word dynamically	`M-/`

Miscellaneous

numeric argument	`C-u` *num*
negative argument	`M--`
quoted insert	`C-q` *char*

Regular Expressions

any single character except a newline	`.` (dot)
zero or more repeats	`*`
one or more repeats	`+`
zero or one repeat	`?`
quote special characters	`\`
quote regular expression special character *c*	`\`*c*
alternative (“or”)	`|`
grouping	`\( ... \)`
shy grouping	`\(:? ... \)`
explicit numbered grouping	`\(:NUM ... \)`
same text as *n*th group	`\`*n*
at word break	`\b`
not at word break	`\B`

entity	**match start**	**match end**
line	`^`	`$`
word	`\<`	`\>`
symbol	`\_<`	`\_>`
buffer	`\‘`	`\’`
class of characters	**match these**	**match others**
explicit set	`[ ... ]`	`[^ ... ]`
word-syntax character	`\w`	`\W`
character with syntax *c*	`\s`*c*	`\S`*c*
character with category *c*	`\c`*c*	`\C`*c*

International Character Sets

specify principal language	`C-x RET l`
show all input methods	`M-x list-input-methods`
enable or disable input method	`C-\`
set coding system for next command	`C-x RET c`
show all coding systems	`M-x list-coding-systems`
choose preferred coding system	`M-x prefer-coding-system`

Info

enter the Info documentation reader	`C-h i`
find specified function or variable in Info	`C-h S`

Moving within a node:

scroll forward	`SPC`
scroll reverse	`DEL`
beginning of node	`b`

Moving between nodes:

next node	`n`
previous node	`p`
move **up**	`u`
select menu item by name	`m`
select *n*th menu item by number (1–9)	*n*
follow cross reference (return with `l`)	`f`
return to last node you saw	`l`
return to directory node	`d`
go to top node of Info file	`t`
go to any node by name	`g`

Other:

run Info **tutorial**	`h`
look up a subject in the indices	`i`
search nodes for regexp	`s`
quit Info	`q`

Registers

save region in register	`C-x r s`
insert register contents into buffer	`C-x r i`
save value of point in register	`C-x r SPC`
jump to point saved in register	`C-x r j`

Keyboard Macros

start defining a keyboard macro	`C-x (`
end keyboard macro definition	`C-x )`
execute last-defined keyboard macro	`C-x e`
append to last keyboard macro	`C-u C-x (`
name last keyboard macro	`M-x name-last-kbd-macro`
insert Lisp definition in buffer	`M-x insert-kbd-macro`

Commands Dealing with Emacs Lisp

eval **sexp** before point	`C-x C-e`
eval current **defun**	`C-M-x`
eval **region**	`M-x eval-region`
read and eval minibuffer	`M-:`
load a Lisp library from **load-path**	`M-x load-library`

Simple Customization

customize variables and faces	`M-x customize`

Making global key bindings in Emacs Lisp (example):

```
(global-set-key (kbd "C-c g") 'search-forward)
(global-set-key (kbd "M-#") 'query-replace-regexp)
```

Writing Commands

```
(defun command-name (args)
  "documentation" (interactive "template")
  body)
```

An example:

```
(defun this-line-to-top-of-window (line)
  "Reposition current line to top of window.
With prefix argument LINE, put point on LINE."
  (interactive "P")
  (recenter (if (null line)
                0
              (prefix-numeric-value line))))
```

The `interactive` spec says how to read arguments interactively. Type `C-h f interactive RET` for more details.

For copies of the GNU Emacs manual, see:

`http://www.gnu.org/software/emacs/#Manuals`

Bibliographie

Indication approximative du niveau des ouvrages référencés :

- **(A)** accessible aux débutants
- **(B)** niveau intermédiaire
- **(C)** niveau ou sujet spécialisé

Abrahams, Paul W. et Bruce R. Larson, *Unix pour l'impatient*, 462 pages (O'Reilly, 1999), ISBN 2-84177-015-X.
(B) Un document très complet détaillant les variantes des commandes suivant les systèmes unix et précisant en particulier celles qui respectent la norme POSIX.

Aho, Alfred, Brian Kernighan et Peter Weinberger, *awk, langage de programmation*, 256 pages (Addison-Wesley, 1995), ISBN 2-87908-110-6.
(B) Écrit par les concepteurs du langage `awk` eux-mêmes, cet ouvrage constitue une mine d'exemples astucieux pour approfondir l'utilisation du filtre `awk`.

Albing, Carl, JP Vossen et Cameron Newham, *bash, Le livre de recettes*, 636 pages (O'Reilly Media Inc., 2007), première édition, ISBN 2-84177-447-3.

Berlat, Abdelmadjid, Jean-François Bouchaudy et Gilles Goubet, *Unix Shell*, 350 pages (Eyrolles, 2001), ISBN 2-212-09094-3.
(A) Un ouvrage concis sur unix orienté utilisateur, illustré d'exercices et doté de précieuses annexes.

Besançon, Thierry, *Administration de systèmes Unix*, Formation Permanente, Université Pierre et Marie Curie, 14e édition, 2012, URL `http://www.formation.jussieu.fr/ars/2011-2012/UNIX/cours/`.
(B) Ces visuels du cours d'Administration Systèmes Réseaux (ARS) de la Formation Permanente de l'UPMC présentent en détail le système unix. L'objectif du cours est bien sûr l'administration, mais les bases d'unix sont présentées de façon très pédagogique, avec de nombreuses comparaisons avec le système Windows.

Bouillet, Dominique, *Unix par la pratique*, 224 pages (Ellipses, 1997), ISBN 2-7298-9711-9.
(A) Une introduction concise à unix du point de vue des utilisateurs. La démarche de cet ouvrage est assez proche de celle de l'UE même s'il traite essentiellement du shell `ksh`.

Bourne, Steve, *Le système unix*, 398 pages (Inter Editions, Addison-Wesley, 1991), ISBN 2-7296-0014-0.
(A-B) Un ouvrage classique écrit par un des concepteurs du shell (le Bourne-shell). Très progressif, il présente d'abord le shell pour l'utilisateur, puis la partie programmation du shell ; il introduit ensuite le langage C et la programmation système et présente enfin les filtres unix.

Cameron, Debra, James Elliott, Marc Loy, Eric S. Raymond et Bill Rosenblatt, *Learning GNU Emacs*, 544 pages (O'Reilly, 2004), troisième édition, ISBN 0-596-00648-9.
(C) Cette troisième édition, revue et augmentée de Cameron et Rosenblatt (1997) décrit l'éditeur de texte `emacs` dans sa version 21.3. Elle intègre un chapitre sur l'installation d'`emacs` sur divers systèmes d'exploitation et met à jour la présentation de cet outil à la fois éditeur de texte, gestionnaire des fichiers, interface du système et véritable environnement de développement.

Cameron, Debra et Bill Rosenblatt, *GNU Emacs*, 462 pages (O'Reilly, 1997), deuxième édition, ISBN 2-84177-015-X.
(C) L'ouvrage classique sur `emacs`, éditeur de texte, mais aussi outil très puissant pour la gestion des fichiers, de la messagerie, ... ou comment `emacs` peut devenir un véritable environnement de développement.

CHAMPARNAUD, JEAN-MARC et GEORGES HANSEL, *Passeport pour UNIX et C*, 502 pages (Vuibert, 2000), ISBN 2-7117-8663-3.
(B) Une approche assez classique consistant à présenter conjointement UNIX et le langage C ainsi que la bibliothèque standard du C.

DESGRAUPES, BERNARD, *Introduction aux expressions régulières*, 248 pages (Vuibert, 2001), ISBN 2-7117-8680-3.
(C) Un ouvrage en français intégralement consacré aux expressions rationnelles qui décrit en détail les subtilités de leur syntaxe. Loin de se restreindre aux filtres UNIX, il aborde les applications dans plusieurs langages dont `perl` et `python`.

DESGRAUPES, BERNARD, *Passeport pour Unicode*, 288 pages (Vuibert, 2005), ISBN 2-7117-4827-8.

DOUGHERTY, DALE et ARNOLD ROBBINS, *sed & awk*, 407 pages (O'Reilly International Thomson, 1999), deuxième édition, ISBN 1-56592-225-5.
(C) Ouvrage spécialisé destiné à ceux qui souhaient approfondir les filtres `sed` et `awk`, notamment grâce aux exemples d'applications avancées présentés.

DUBOIS, PAUL, *Using csh & tcsh*, 239 pages (O'Reilly International Thomson, 1995), ISBN 1-56592-132-1.
(B) L'équivalent de la référence NEWHAM et ROSENBLATT (2006) sur `bash`, pour ceux qui choisissent de travailler dans les shells de type C (`csh` et surtout `tcsh`). Les shells-scripts ne sont cependant pas abordés, car l'auteur lui-même préfère les écrire en `sh` ou en `perl`.

GRANNEMAN, SCOTT, *Linux, l'essentiel du code et des commandes*, 372 pages (Pearson Education, 2011), ISBN 978-2-7440-2523-5.
(A) Dans la collection *Guide de survie*, cet ouvrage, présenté comme un aide-mémoire dans un format de poche, est une description concise des commandes de linux, illustrée de nombreux exemples très pertinents. Il décrit surtout les commandes utilisateur, avant d'aborder les aspects installation et configuration. L'édition originale, *Linux Phraseboook* de 2006 en anglais, a été traduite par DAMIEN MARTIN DE LA SALLE et l'édition française a été revue et corrigée par ÉRIC JACOBONI en 2011.

KERNIGHAN, BRIAN et ROB PIKE, *The Unix programming environment*, 240 pages (Prentice Hall, 1984), ISBN 0-13-93768.
(B) Écrit par les concepteurs du système UNIX, il s'agit avant tout d'un ouvrage d'intérêt historique. Les premiers chapitres présentent le système UNIX, puis les filtres avant d'aborder la programmtion shell. La deuxième partie de l'ouvrage introduit le langage C et est consacrée à la programmation système. D'un niveau avancé, il n'est pas conseillé comme premier contact avec UNIX.

KIDDLE, OLIVER, JERRY PEEK et PETER STEPHENSON, *From Bash to Z Shell : Conquering the Command Line*, 472 pages (Apress, 2004), ISBN 978-1-59059-376-9.
(B) Un premier ouvrage traitant du shell `zsh`, mais qui constitue plus généralement une présentation moderne des shells et aborde aussi le shell `bash`.

LAMB, LINDA et ARNOLD ROBBINS, *Learning the vi editor*, 173 pages (O'Reilly International Thomson, 1998), sixième édition, ISBN 1-56592-426-6.
(A-B) Un document essentiel et très concis qui présente l'éditeur `vi` ainsi que ses diverses extensions. Pour une étude plus complète de `vim`, on pourra consulter la septième édition ROBBINS *et al.* (2008), qui constitue une refonte importante mais plus volumineuse.

LÉRY, JEAN-MICHEL et FRÉDÉRIC JACQUENOD, *UNIX & Linux*, 800 pages (Pearson Education, 2011), troisième édition, ISBN 978-2-7440-7542-1.
(A-B) Écrit par l'ancien directeur du CICRP et son collègue, cet ouvrage réédité très récemment, et illustré de nombreux exemples et exercices couvre un spectre très large allant de l'utilisation de linux à l'administration des systèmes UNIX.

NEBRA, MATHIEU, *Reprenez le contrôle à l'aide de LINUX*, 506 pages (Simple IT, 2010), ISBN 978-2-9535278-2-7.
(A) Écrit par le fondateur du site du zéro (`http://www.siteduzero.com/`), cet ouvrage très récent regroupe et complète les célèbres cours linux pour débutants du site.

Newham, Cameron et Bill Rosenblatt, *Le shell bash*, 342 pages (O'Reilly Editions, 2006), troisième édition, ISBN 2-84177-403-1.
(B) L'ouvrage classique permettant d'approfondir le shell `bash` à la fois interpréteur de commande et langage de programmation (« exploitez et programmez votre shell ») dans sa troisième édition traduite en français.

Powers, Shelley, Jerry Peek, Tim O'Reilly et Mike Loukides, *Unix Power Tools*, 113 pages (O'Reilly, 2003), troisième édition, ISBN 0-596-00330-7.
(C) Ouvrage très avancé sur l'environnement unix, le shell et les scripts. Il s'adresse à des utilisateurs avertis qui vont le parcourir pour s'inspirer des nombreux exemples commentés qui y sont compilés.

Pélissier, Christian, *Unix : utilisation, administration réseau internet*, 895 pages (Hermès, 1998), troisième édition, ISBN 2866017072.
(B) La première partie de l'ouvrage présente unix du point de vue utilisateur, alors que la deuxième aborde le système lui-même. Sont notamment présentés le shell et le C-shell ainsi que la génération d'applications avec `make`. Cette troisième édition, considérablement augmentée fait une plus large place aux interfaces graphiques et à l'environnement réseau.

Ramey, Chet et Brian Fox, *GNU Bash Reference Manual*, 204 pages (Network Theory Ltd., 2006), 3e édition, ISBN 0-9541617-7-7, URL http://www.network-theory.co.uk/bash/manual/.
(B) Un ouvrage de référence très complet sur le shell `bash`

Rifflet, Jean-Marie et Jean-Baptiste Yunès, *Unix : programmation et communication*, 800 pages (Dunod, 2003), quatrième édition, ISBN 2100079662.
(C) Ouvrage classique plutôt destiné à des informaticiens désireux d'approfondir l'étude de la programmation du système unix.

Robbins, Arnold, *Effective awk programming*, 421 pages (O'Reilly International Thomson, 1999), troisième édition, ISBN 0-596-00070-7.
(C) Pour ceux qui souhaitent approfondir le langage `awk`, et aussi les extensions proposées par `gawk`, la version gnu de `awk`.

Robbins, Arnold et Nelson Beebe, *Introduction aux scripts shell - Automatiser les tâches Unix*, 580 pages (O'Reilly Media Inc., 2006), première édition, ISBN 2-84177-375-2.

Robbins, Arnold, Elbert Hannah et Linda Lamb, *Learning the vi and Vim editors*, 492 pages (O'Reilly Media, 2008), septième édition, ISBN 0-596-52983-X.
(A-B) Au prix d'une augmentation importante de la pagination, la septième édition de Lamb et Robbins (1998) comporte une présentation beaucoup plus complète de la version « étendue » `vim` de l'éditeur `vi`.

Rosenblatt, Bill et Arnold Robbins, *Learning the Korn Shell*, 432 pages (O'Reilly International Thomson, 2002), deuxième édition, ISBN 0-596-00195-9.
(B) L'équivalent de la référence Newham et Rosenblatt (2006) sur `bash`, pour le shell `ksh`.

Schwartz, Randal L. et Tom Phoenix, *Introduction à Perl*, 291 pages (O'Reilly, 2002), troisième édition, ISBN 2-84177-201-2.
(A-B) Un document très pédagogique à conseiller pour s'initier à `perl` et comprendre l'état d'esprit du langage au delà des aspects syntaxiques.

Waldmann, Uwe, *A Guide to Unix Shell Quoting*, Max Planck Institut Informatik, 2008, URL http://www.mpi-inf.mpg.de/~uwe/lehre/unixffb/quoting-guide.html.
(C) Issue d'un cours avancé d'unix, une revue plutôt pointue des problèmes de protection des métacaractères sous `bash`, mais aussi sous d'autres shells et quelques applications.

Welsh, Matt, Matthias Kalle Dalheimer, Terry Dawson et Lar Kaufman, *Le système Linux*, 700 pages (O'Reilly International Thomson, 2003), quatrième édition, ISBN 2-84177-241-1.
(B) Une référence pour débuter dans l'installation et l'administration d'un système linux, couvrant notamment les aspects matériels et réseau.

Index

– A –

– B –

– C –

– U –

– V –

– W –

– X –

– Z –