UPMC

Master P&A/SDUEE

UE MNI (4P009)

Méthodes Numériques et Informatiques – B

Introduction à l'environnement Unix

Jacques.Lefrere@upmc.fr

Sofian.Teber@lpthe.jussieu.fr

Albert Hertzog

2014–2015

Table des matières

1 Introduction au système UNIX

1.1 Système d'exploitation

— ensemble de programmes d'un ordinateur servant d'**interface** entre le matériel et les logiciels applicatifs

— abrégé S.E. (en anglais *operating system* O.S.)

— exemples : MS-DOS, Windows (XP, ..., 7, 8), famille Unix (linux, Mac-OS, ...)

Linux aujourd'hui dominant dans le calcul intensif :

plus de 97% des calculateurs du TOP 500

`http://www.top500.org/statistics/`

N.-B. : machine virtuelle = application qui émule un système d'exploitation physique

1.2 Historique d'unix

— depuis les années 1970,
rôle essentiel des milieux universitaires dans la diffusion d'unix

— grande diffusion assurée grâce à la portabilité du langage C
(moins de 10 % du noyau écrit en assembleur)

— plusieurs branches de développement (BSD et System V)
mais normalisation POSIX (*Portable Operating System Interface*)

— système ouvert : implémentations sur diverses architectures
du téléphone portable au super-calculateur
  – propriétaires (aix, hp-ux, solaris, os-X, ...)
  – libres (linux depuis 1991, net-bsd, free-bsd, ...) : linux est (presque) un unix !
  plusieurs distributions linux : debian, ubuntu, mint,
  Red-Hat, mandriva puis mageia, scientific-linux, CentOS, Fedora, ...

$\Rightarrow$ quelques différences dans les commandes (ex. : `ps`, impression `lpr`/`lp`, ...)
mais surtout au niveau administration (gestion des packages par ex.)

1.3 Principales caractéristiques du système UNIX

— interactif (mais traitement *batch* possible)

— multi-tâches (concurrentes et indépendantes)

— multi-utilisateurs (dont l'administrateur ou *root*)

⇒ système d'**identification** et **droits** d'accès aux fichiers

— documentation en ligne (`man`, `info`, ...)

— intégration dans le **réseau**

– partage de ressources (fichiers, authentification, ...)

– applications réparties

— chaînage des processus par les **tubes** (pipes)

⇒ assemblage d'outils élémentaires pour accomplir des tâches complexes

— l'interpréteur de commandes (**shell**) intègre un **langage de programmation**

⇒ programmes interprétés en shell = shell-scripts

1.4 L'interpréteur de commandes ou shell

Le **shell** est l'interface utilisateur du système d'exploitation.

Deux familles (liées aux 2 branches d'unix) avec deux syntaxes différentes
(en particulier dans la programmation) et des fichiers de configuration différents :

	versions libres	fichiers d'initialisation
`sh`, `ksh` (Korn)	`pdksh`	`.profile` `.kshrc`
	`bash` (linux)	**`.[bash_]profile`** **`.bashrc`**
`csh`	**`tcsh`** (ancien mac-os-X)	`.login` `.[t]cshrc`
compatible avec les deux syntaxes	**`zsh`**	`.zprofile` `.zlogin` `.zshrc`

`sh` : shell historique de Bourne

`bash` : ***B**ourne **A**gain **SH**ell*

1.5 Compte utilisateur

- identifiant (ou *login*) (unique)
- mot de passe (ou *password*) confidentiel
- un groupe parmi ceux définis sur la machine
- un répertoire d'accueil personnel (ou *home directory*) où stocker ses fichiers
- un « interpréteur de commandes » (ou *shell*) :
  sh, ksh, **bash**, **csh**, **tcsh** ou **zsh**.

L'ensemble de ces informations est stocké dans un fichier système

mot de passe crypté (souvent dans `/etc/passwd`)

$\Rightarrow$ l'administrateur ne peut pas retrouver un mot de passe oublié

Ressources limitées, par exemple par quota sur le disque

$\Rightarrow$ problème de connexion en mode graphique si quota atteint.

1.6 Sessions unix

— deux types de **sessions** de travail :

  – mode texte (console, accès distant (`slogin`), ...) : ligne de commande

  – mode graphique (multi-fenêtres) : icônes et menus pour lancer les applications
  (dont les consoles **`konsole`** et `xterm` par exemple)
  gestionnaires de fenêtres : `fvwm`, **`kde`**, `gnome`, **`icewm`**, **`lxde`**...

— point commun

  – identification (*login*)

  – authentification (*password*)

Sous linux, en cas de problème en mode graphique,
passage en mode texte par frappe simultanée de
Ctrl | Alt | F1 (6 consoles de `F1` à `F6`).
Retour en mode graphique par Ctrl | Alt | F7 ou Ctrl | Alt | F8 (Mandriva 2010)

2 Le shell : introduction

Le shell est un programme qui interprète les commandes saisies dans un terminal.

2.1 Syntaxe de la ligne de commandes

Le shell découpe la ligne de commande en mots séparés par des blancs

plus généralement par l'`IFS` (***I**nput **F**ield **S**eperator*)

(1) premier mot = **la commande** action

(2) mots suivants = **les paramètres** les objets

(rôle déterminé par leur position dans la ligne de commande)

(3) paramètres optionnels introduits par « **–** » les modalités

cp	**-p**	**fich1**	**fich2**
(1) commande	(3) option	(2) paramètre 1	(2) paramètre 2
copie	en gardant la date	source	cible

2.2 Exemples de commandes élémentaires d'affichage

commande	affichage
date	de la date
whoami	du login
hostname	du nom de la machine
who	de la liste des utilisateurs connectés
echo "chaine de caracteres"	de la chaîne saisie
id	du numéro d'utilisateur
uname	du nom du système d'exploitation

Le shell

— distingue les **majuscules** (rares) des **minuscules**

— interprète certains **caractères** dits **spéciaux**

par exemple les blancs (les éviter dans les noms de fichiers)

2.3 Caractères spéciaux pour le shell (I)

— Caractères de contrôle du terminal (affichage par `stty -a`)

^C	interruption du processus en cours
^Z	suspension du processus en cours (reprise possible)
^? ou **^H**	effacement du dernier caractère (choix par `stty erase ^?`)
^D	fermeture du flux d'entrée (fin de session en shell)

— Aides à l'interactivité

↑ et ↓	permettent de parcourir l'historique des commandes
← et →	déplacements pour éditer la ligne de commande
^E ou **^A**	déplacement en fin (***End***) ou début de ligne (**^B** est pris *Back*)
TAB	demande au système de compléter le nom de commande ou de fichier
	⇒ évite les fautes de saisie et valide les chemins
TAB **TAB**	affiche les différentes possibilités de complétion

— plus beaucoup d'autres (voir chapitres suivants)

— **Générateurs de noms de fichiers** (motifs génériques avec caractères jokers)

`*`	une chaîne de caractères quelconque dans le nom d'un fichier (y compris la chaîne vide)
`?`	**un** caractère quelconque et un seul dans un nom de fichier
`[...]`	**un** caractère quelconque pris dans la liste exhaustive entre crochets
`[`c_1`-`c_2`]`	**un** caractère quelconque entre c_1 et c_2 dans l'**ordre lexicographique**
`[!...]`	**un** caractère quelconque pris **hors** de la liste

— Mise en facteur par des accolades de chaînes séparées par des virgules

`pre{`*`chaine,str,ch3`*`}post` (sans espace) se développe en

`pre`**`chaine`**`post` `pre`**`str`**`post` `pre`**`ch3`**`post`

Exemples de motifs de noms de fichiers

`*`	tous les fichiers du répertoire courant (sauf ceux commençant par `.`)
`*.f90`	tous les fichiers dont le nom finit par `.f90`
`*.*`	tous les fichiers dont le nom comporte un point (au moins)
`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
	NB. : `f.[25-70]` (maladroit, mais) les fichiers `f.0`, `f.2`, `f.5`, `f.6` et `f.7`
`f.[!0-9]`	les fichiers dont le nom s'écrit `f.` suivi d'un caractère qui n'est pas un chiffre
`*.[ch]`	les fichiers source en C et les fichiers d'entête (*header*)

N.-B. : si aucun fichier ne correspond, le générateur est restitué inchangé avec ses caractères jokers

(sauf avec l'option `nullglob` fixée par : `shopt -s nullglob`).

Exemple d'expansion d'accolades

`echo bon{jour,soir}` affiche la chaîne `bonjour bonsoir`

2.4 Documentation en ligne

Différents moyens d'accéder à la documentation en ligne d'une commande :

— **man cmd** : affiche du manuel de la commande `cmd`

page par page grâce au filtre `more` ou **less**

– se déplacer dans le manuel : ↑ ↓, page suivante/précédente

– rechercher un motif : **/motif**

– sortir du manuel : touche q *quit*

Préciser parfois la section du manuel (1 = commandes, 3 = bibliothèques)

man 3 printf (⇒ la fonction C)

au lieu de `man printf` (⇒ section 1 commande)

— **cmd --help** : affiche un bref aide-mémoire de la commande

— **info cmd** : système de documentation hypertexte de certaines commandes

Rechercher quelle commande utiliser pour une opération : **man -k *motclef***

Autre source d'information : usage **averti** d'un moteur de recherche sur le web

3 Hiérarchie des fichiers unix

3.1 Arborescence

L'ensemble des fichiers est structuré sous la forme d'une hiérarchie

— de **répertoires** (*directories*, dossiers (*folders*) sous windows)

— et de **fichiers** (*files*)

constituant un arbre unique.

— / est la **racine** (*root*) : le répertoire qui contient tous les autres fichiers ;

— ses **nœuds** sont des sous-répertoires...

— ses **feuilles** sont les fichiers ordinaires (en général).

— le séparateur de niveaux est la barre oblique / (*slash*)

NB1 : l'arbre unique d'unix est purement logique ; plusieurs périphériques peuvent y être « montés », par exemple `/media/cdrom`, `/media/removable`

NB2 : sous windows, le séparateur est la contre-oblique \ (*antislash*)
sous windows, les périphériques sont désignés par une lettre préfixe suivie de « **:** », par exemple `C:\` ou `D:\`

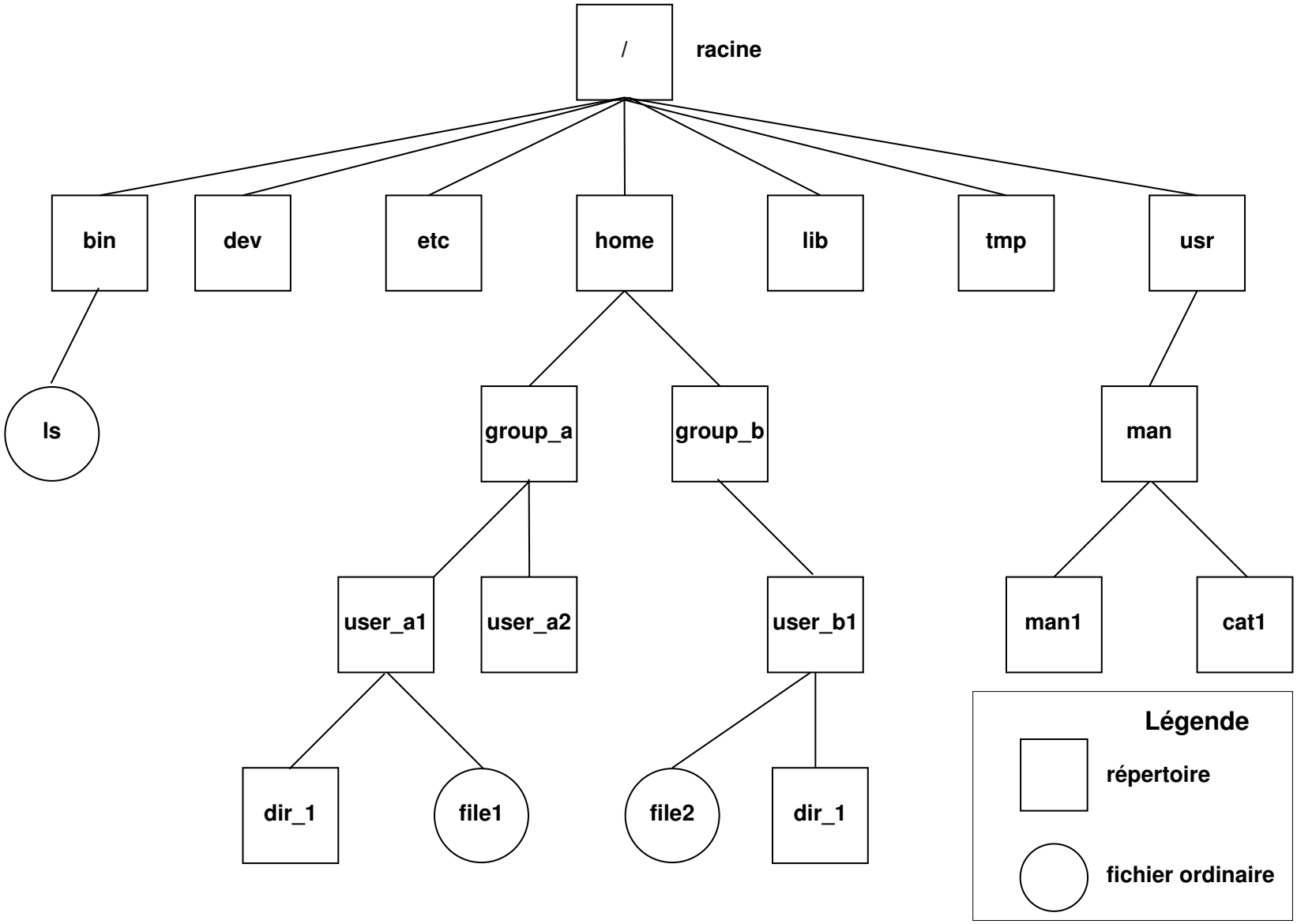


FIGURE 1 – Arborescence des fichiers UNIX

3.2 Chemins d'accès (*path*) d'un fichier

— **le chemin absolu** : commence toujours par / et comporte la liste complète des répertoires traversés depuis la racine,

Exemples : `/usr/man/man1/ls.1`, `/home/group_a/user_a1`

— **un chemin relatif** : comporte la liste des répertoires à parcourir depuis le répertoire courant jusqu'au fichier ou répertoire choisi.

Il ne commence jamais par / et doit passer par un nœud commun à la branche de départ (répertoire courant) et la branche d'arrivée.

- `.` **répertoire courant** ou de travail (*working directory*)
- `..` **répertoire père** (*parent directory*)

Exemples, partant de `/home/group_a/user_a1` :

`dir_1`, `../`, `../user_a2`, `../../group_b/user_b1`

Des fichiers de même nom ne peuvent exister que dans des répertoires différents

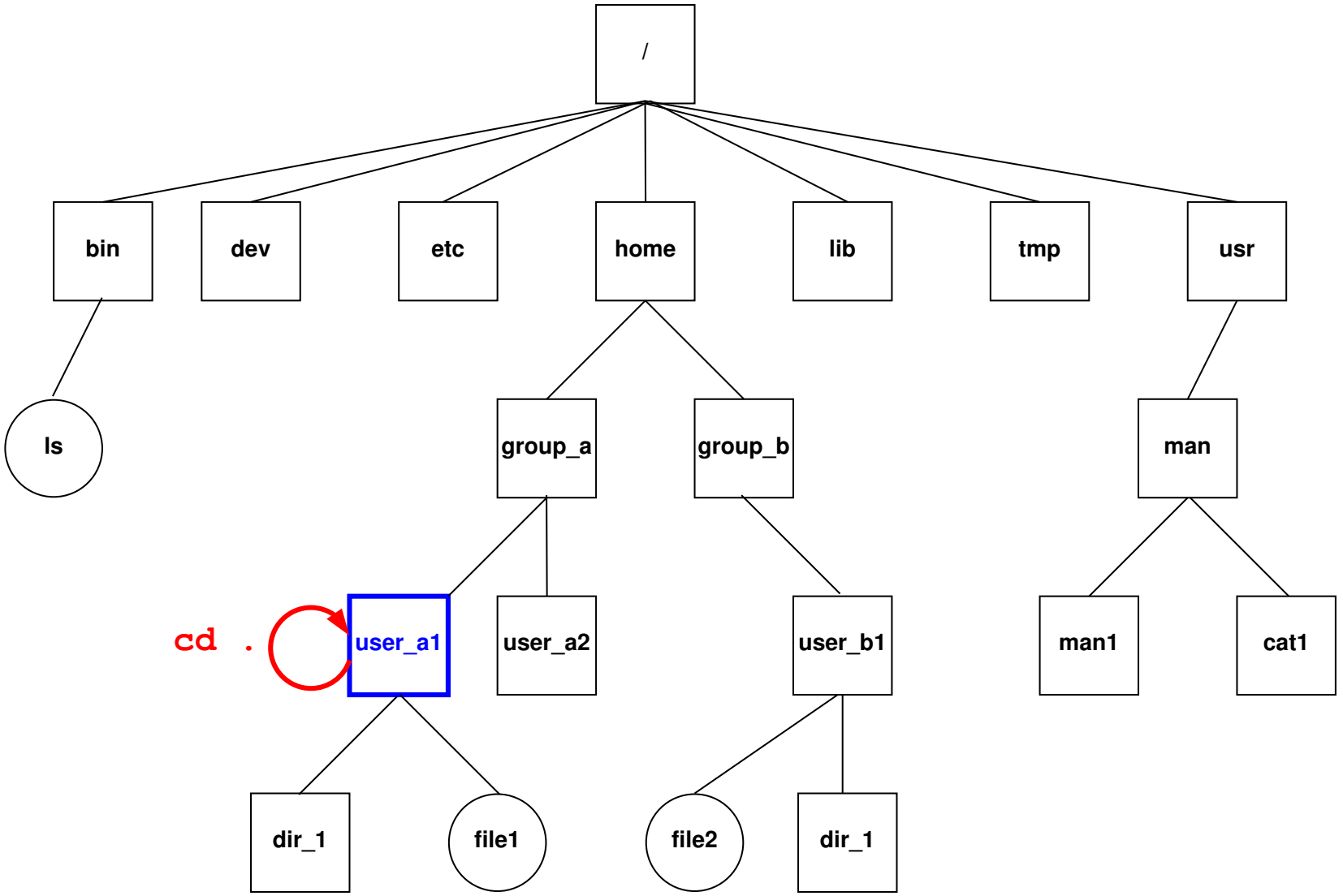


FIGURE 2 – La commande **cd .** laisse dans le répertoire courant `/home/group_a/user_a1`.

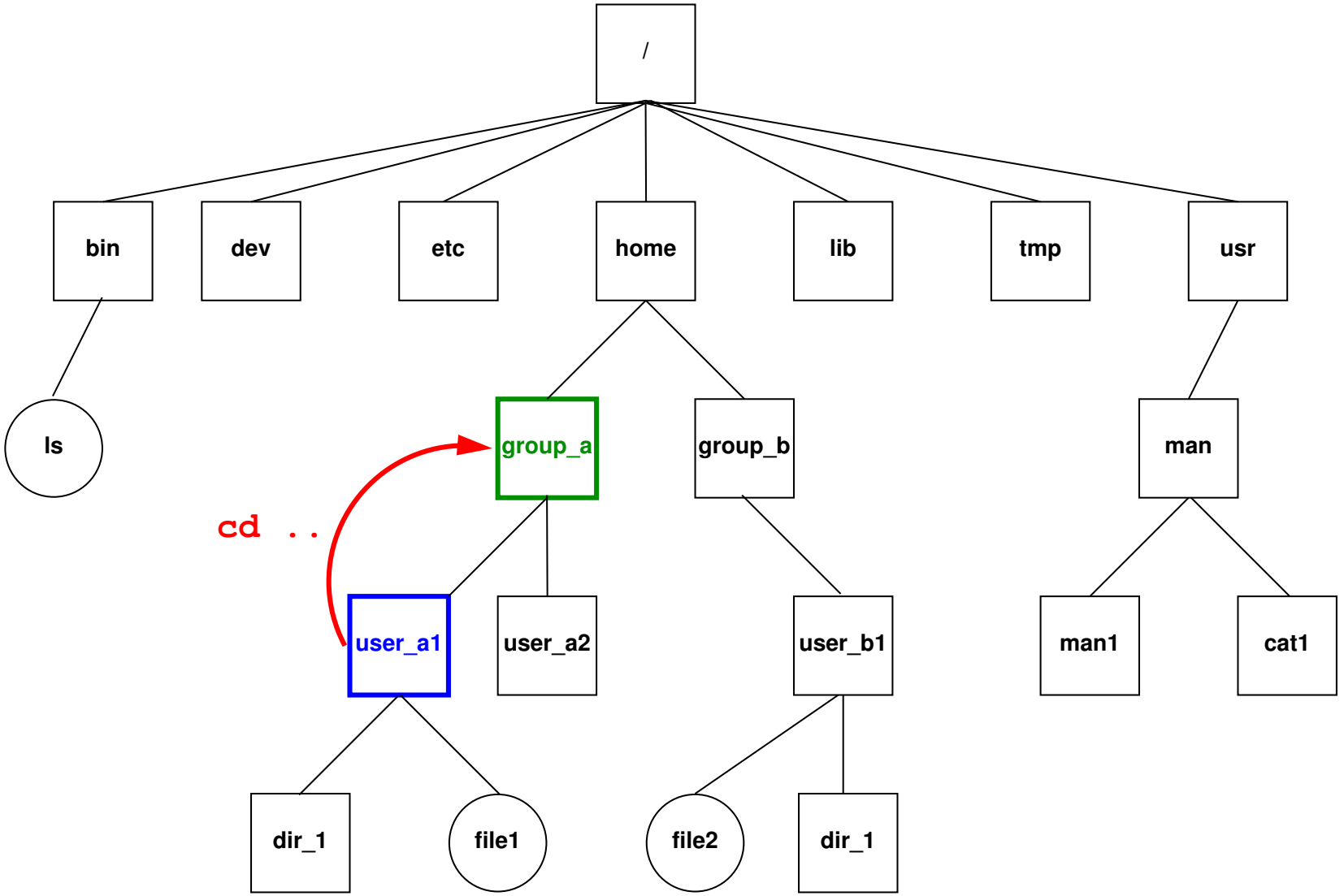


FIGURE 3 – La commande **cd ..** déplace dans le répertoire père `group_a`.

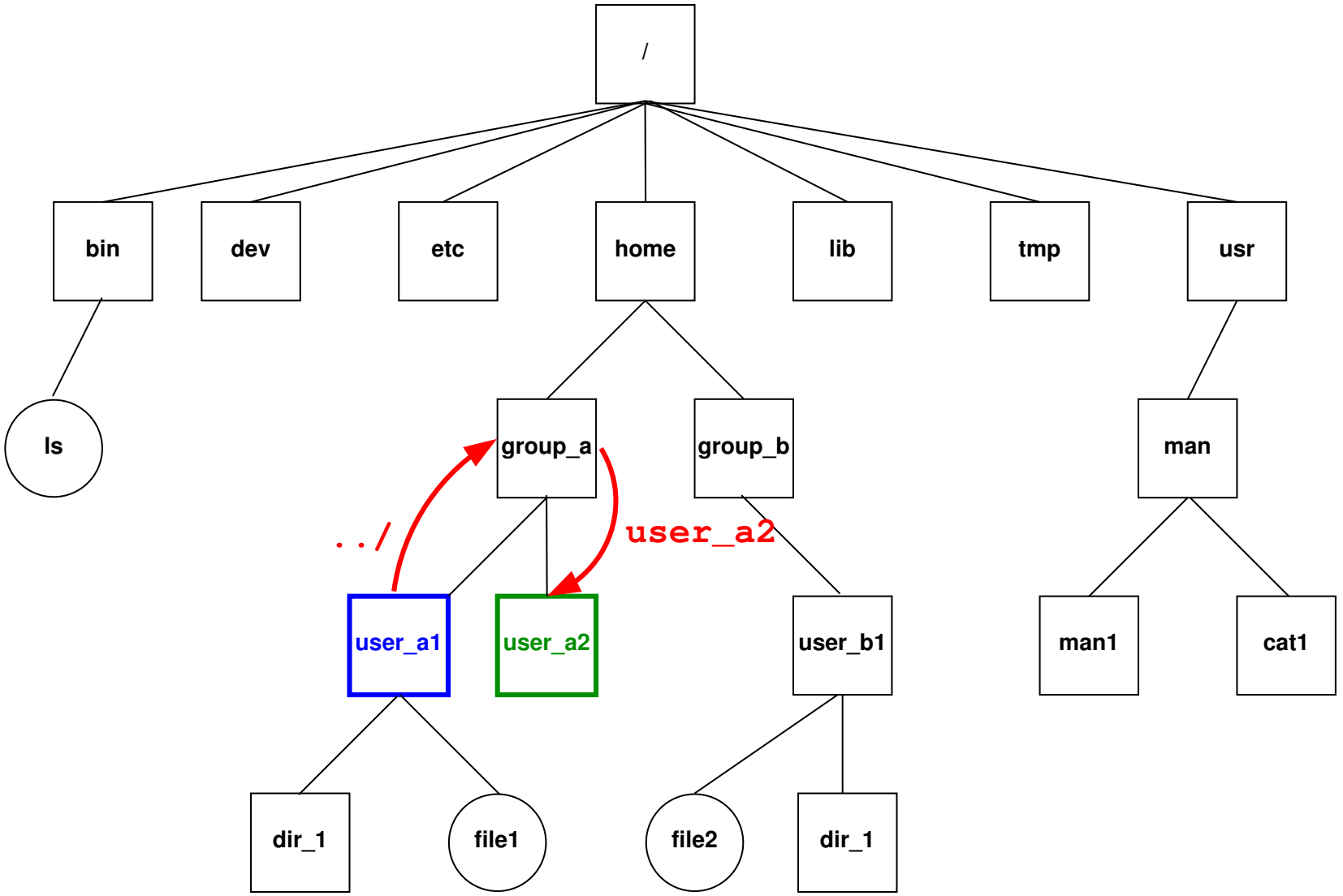


FIGURE 4 – La commande **`cd ../user_a2`** déplace dans le répertoire `user_a2`

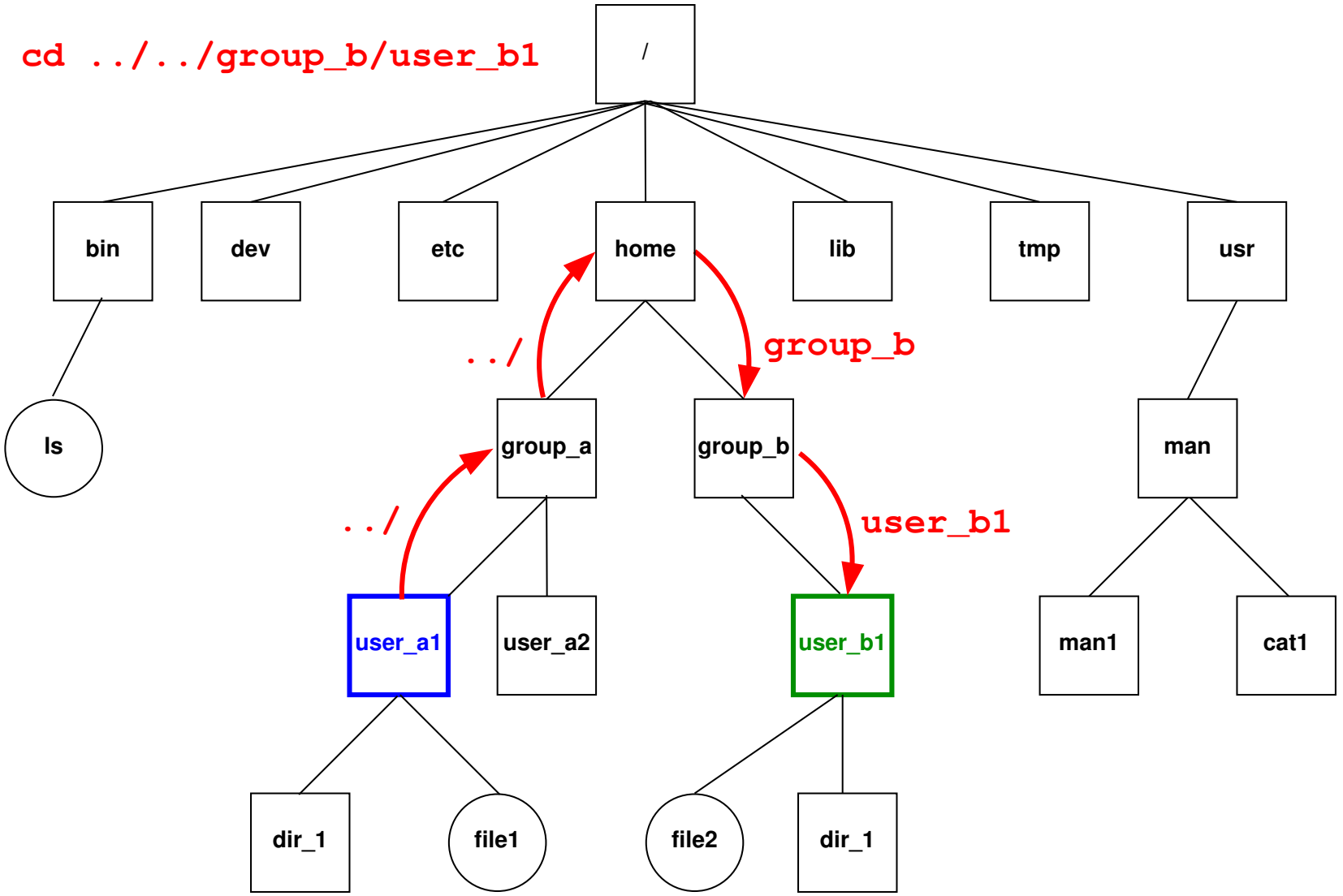


FIGURE 5 – La commande **`cd ../../group_b/user_b1`** déplace dans le répertoire `user_b1`.

3.3 Raccourcis pour les répertoires d'accueil

Chemins en fait absolus :

- **~*user*** répertoire d'accueil d'un utilisateur quelconque
- **~** son propre répertoire d'accueil

Exemples :

~/.bash_profile

est le chemin absolu de votre fichier d'initialisation personnel.

~lefrere/M1/Doc/unix/poly-unix/

est le chemin absolu du répertoire du polycopié UNIX,

situé sous le compte de l'utilisateur lefrere.

3.4 Visualisation d'une branche avec `tree`

La commande **`tree`** permet de représenter une branche de la hiérarchie de fichiers.

```
-] tree .
.
|-- fic
|-- rep1
|   |-- fic1
|   |-- rep1-1
|   |   `-- fic1-1
|   `-- rep1-2
|-- rep2
|   `-- fic2
`-- rep3
    |-- rep3-1
    |   `-- fic3-1
    `-- rep3-2

7 directories, 5 files
```

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

```
-] tree -f .
.
|-- ./fic
|-- ./rep1
|   |-- ./rep1/fic1
|   |-- ./rep1/rep1-1
|   |   `-- ./rep1/rep1-1/fic1-1
|   `-- ./rep1/rep1-2
|-- ./rep2
|   `-- ./rep2/fic2
`-- ./rep3
    |-- ./rep3/rep3-1
    |   `-- ./rep3/rep3-1/fic3-1
    `-- ./rep3/rep3-2

7 directories, 5 files
```

4 Commandes de base

4.1 Commandes de gestion de fichiers

4.1.1 Affichage de liste de noms de fichiers avec `ls`

`ls` [*-options*] [*liste_de_fichiers*]

`-a` (***a****ll*) liste aussi les fichiers cachés (de nom commençant par **`.`**)

`-l` (***l****ong*) affiche les attributs (droits, taille, date, ...) des fichiers

`-R` (***R****ecursive*) affiche la liste des fichiers contenus dans tous les sous répertoires éventuels

`-F` (***F****lag*) marque les fichiers répertoires (**`/`**), exécutables (**`*`**) ou les liens (**`@`**)

`-t` (***t****ime*) classe la liste par ordre de date des fichiers

`-d` (***d****irectory*) affiche le nom des répertoires mais pas leur contenu

`-h` (***h****uman readable*) affiche la taille en utilisant les multiples `k` (kilo) `M` (méga) `G` (giga)

`-r` (***r****everse order*) affiche dans l'ordre inverse

Exemples d'usage de `ls`

sans objet : par défaut le répertoire **courant**

`ls`	liste (courte) des fichiers du répertoire courant
`ls -l`	liste des fichiers du répertoire courant avec attributs
`ls -al`	idem avec aussi les fichiers cachés (commençant par `.`)

cas des répertoires : par défaut le **contenu**

`ls -l rep`	liste détaillée des fichiers du répertoire `rep` (contenu)
`ls -dl rep`	nom et attributs du **répertoire** `rep` (contenant)
`ls -l fic1 fic2 rep`	liste détaillée des fichiers ordinaires `fic1`, `fic2` et du contenu du répertoire `rep`
`ls -t /tmp`	liste des fichiers de `/tmp` par ordre chronologique

4.1.2 Copie de fichiers avec `cp`

en anglais ***copy***

— copie avec changement de nom éventuel (deux arguments seulement)

`cp` `[-options]` *`fichier_`**`origine`*** *`fichier_`**`cible`***

— copie d'un ou plusieurs fichiers sans changement de nom

vers un même répertoire

`cp` `[-options]` ***`liste`**`_de_fichiers`* ***`répertoire`**`_cible`*

`cp *.c bck/` copie les fichiers source C dans le répertoire `bck`

Principales options :

`-i` (***i**nteractive*) demande de confirmation si *`fichier_cible`* existe déjà

`-r` (***r**ecursive*) copie d'une branche (si le premier objet est un répertoire)

`-p` (***p**ermissions*) sans changer les droits ni la date

Confirmation en cas d'écrasement : répondre **`y`** (**`o`** si francisé)

4.1.3 Déplacement et renommage de fichiers avec `mv`

en anglais ***move*** 3 syntaxes

1. **`mv`** `[-options]` *`fichier_`***`origine`** *`fichier_`***`cible`**

   2 objets seulement renommage sauf si chemins d'accès différents

2. **`mv`** `[-options]` ***`liste`***`_de_fichiers` ***`répertoire`***`_cible`

   si le répertoire cible existe il y a déplacement, sinon il est créé

3. **`mv`** `[-options] répertoire_source répertoire_cible`

   renommage ou déplacement de branche

Principale option :

`-i` (***i****nteractive*) demande de confirmation interactive si écrasement de fichier

4.1.4 Suppression de fichiers avec `rm`

en anglais ***rem**ove*

`rm` `[-`*`options`*`]` *`liste_de_fichiers`*

Principales options :

`-i` (***i**nteractive*) demande de confirmation interactive

`-r` (***r**ecursive*) destruction d'une branche (puissant mais... dangereux)

`-f` (***f**orce*) sans demande de confirmation ni erreur si fichier inexistant

Attention : pas toujours d'alias en **`rm -i`** !

4.1.5 Compression de fichiers avec `gzip` ou `bzip2`

Compression et décompression sans perte d'information

Compression $\rightarrow$ fichier de suffixe **`.gz`**

`gzip` [*-options*] *liste_de_fichiers*

Décompression d'un fichier de suffixe `.gz`

`gunzip` [*-options*] *liste_de_fichiers*

Autre outil, plus efficace : **`bzip2`**/**`bunzip2`** (suffixe **`.bz2`**)

Format plus récent : **`lzma`** (suffixe **`.lzma`**)

(commandes **`unlzma`** puis **`xz`** avec options `-z`/`-d`)

4.2 Commandes de gestion de répertoires

4.2.1 Affichage du répertoire courant avec `pwd`

`pwd` (***p**rint **w**orking **d**irectory*) affiche le chemin absolu du répertoire courant

commande interne (*builtin*) du shell

4.2.2 Changement de répertoire courant avec `cd`

`cd` [*répertoire*] (***c**hange **d**irectory*)

commande interne (*builtin*) du shell

`cd` (sans paramètre) retour au répertoire d'accueil `~/.`

`cd -` revient au précédent répertoire (dans le temps)

`cd ..` revient au répertoire père (dans la hiérarchie)

4.2.3 Création de répertoire avec `mkdir`

`mkdir` *`répertoire`* (***make directory***)

option `-p` (***parent***) : crée les répertoires parents si nécessaire

exemple : `mkdir -p dir/subdir`

4.2.4 Suppression de répertoire (vide) avec `rmdir`

`rmdir` *`répertoire`* (***remove directory***)

refus de suppression si le répertoire contient des fichiers

$\Rightarrow$ utiliser `rm -R` *`répertoire`*, mais dangereux !

5 Commandes traitant le contenu des fichiers texte

5.1 Fichiers binaires et fichiers texte, codage

Un fichier (ordinaire) = lot d'informations, conservé dans une mémoire permanente (disque, CD, clef USB, ...) et auquel on donne un nom.

Deux aspects du fichier :

bas niveau : suite de bits groupés en octets

haut niveau : représentation de texte, d'image, de code machine,...

selon un certain codage qui permet d'interpréter la suite de bits.

Préférer des suffixes rappelant le type de codage utilisé :

— fichiers texte

`.c` source C, `.f90` source fortran, `.txt` texte, `.html` hypertexte, ...

— fichiers binaires

`.pdf` pour du PDF, `.jpg` pour une image JPEG

`.o` pour un objet binaire, `.a` pour une bibliothèque, ...

5.2 Codage des fichiers textes

Plusieurs codages pour les caractères :

- **ASCII** sur 7 bits (128 caractères) => non accentués
- codages sur 1 octet = 8 bits (256 caractères) avec caractères accentués :
  - propriétaires : CP852, CP1252 sous windows, MacRoman sous MacOS
  - **ISO-8859** avec les variantes locales
    ISO-8859-1 ou **latin1** pour le français par exemple
- évolution en cours vers standard unicode pour représenter toutes les langues :
  nécessiterait jusqu'à 4 octets par caractère : UTF-32 !
  implémentation **UTF-8** : taille variable des caractères : de 1 à 4 octets
  - sur-ensemble de l'ASCII (donc sur 1 octet pour les non-accentués)
  - les caractères non-ascii de latin1 sur 2 octets
  - les codes binaires (sur 1 octet) des caractères accentués de l'ISO-8859-1 sont invalides en UTF-8 !

5.2.1 Transcodage de fichiers textes avec `recode` ou `iconv`

— **`iconv -f`** *`code_initial`* **`-t`** *`code_final fichier`*

attention : la conversion s'arrête à la première combinaison invalide

— **`recode`** *`code_initial..code_final fichier`*

Attention : par défaut `recode` travaille « en place » (modifie le fichier initial).

`recode` permet d'enlever les signes diacritiques : convertir vers code `flat`,

mais transcodage irréversible (option `-f`)

Exemples de transcodage de l'iso vers utf-8 :

```
iconv -f ISO-8859-1 -t UTF-8 < fic-iso.txt > fic-utf8.txt
recode 'ISO-8859-1..UTF-8' < fic-iso.txt > fic-utf8.txt
```

De nombreux éditeurs (`vim`, `emacs`...) peuvent faire de la conversion au vol

pour la phase d'édition, puis sauvegarder dans le codage initial.

Mais `kedit` ouvre les fichiers iso-latin en mode lecture seule !

Attention : ne pas mélanger deux codages dans un fichier (via par ex. copier/coller)

5.3 Accès au contenu des fichiers

5.3.1 Identification des fichiers avec `file`

`file` *`liste_de fichiers`*

affiche une indication sur la nature du fichier (texte, binaire, ...)

⇒ l'utiliser pour savoir avec quelles commandes manipuler un fichier

```
a.out:                  ELF 64-bit LSB executable, x86-64
carre.f90:              ASCII text
carre+invite.c:         symbolic link to `carre+invite-utf.c'
carre+invite-iso.c:     ISO-8859 C program text
carre+invite-utf.c:     UTF-8 Unicode C program text
ligne.txt:              ISO-8859 text
ligne.utf:              UTF-8 Unicode text
poly-unix.tex:          LaTeX 2e document text
te1-arbre.pdf:          PDF document, version 1.5
```

5.3.2 Comptage des mots d'un fichier texte avec `wc`

`wc` `[-cmwl] [liste_de fichiers]` (***w*ords*c*ount**)

Affiche par défaut le nombre de lignes, de mots et d'octets,

sauf si options cumulables pour sélectionner :

`-l` compte les lignes (***l**ines*)

`-w` compte les mots (***w**ords*)

`-m` compte les caractères (***m**ultibytes*, **`c`** est pris !) : utile en UTF-8 seulement

`-c` compte les octets (***c**haracters* au sens historique !) comme `ls -l`

NB : ordre d'affichage **fixe** **`lwmc`**, c'est-à-dire du plus gros au plus petit

N.-B. : `wc -m` exact seulement si codage du fichier et codage déterminé par les variables d'environnement locales (**LC_ALL**) sont cohérents, cas sur fond jaune .

Exemple :

le texte **aceàçéÀÇÉ**, soit 9 caractères + fin de ligne, soit 10 caractères dont 6 accentués, codés en ISO-8859-1 (`texte.iso`) ou en UTF-8 (`texte.utf`) :

`wc -mc`	locale	
	ISO-8859-1	UTF-8
`texte.iso`	**10 10**	**4⋆ 10**
`texte.utf`	**16 16**	**10 16**

caractères en bleu et octets en rouge

⋆ : décodage incomplet ⇒ nb. de caractères < nb. d'octets

car 6 caractères latin1 accentués non décodés en UTF-8.

5.3.3 Affichage du contenu de fichiers texte avec `cat`

`cat` [*liste_de fichiers*]

affiche (con**cat**ène) le contenu des fichiers de la liste

N.B. : pas de contrôle du défilement (voir `more` ou `less`)

ex : `cat fic1 fic2 fic3` concatène et affiche le contenu des trois fichiers

`cat` = filtre identité : recopie l'entrée standard (clavier) sur la sortie standard (écran)

`cat` **`-n`** affiche les lignes avec leur numéro en tête, suivi d'une tabulation

Ne pas confondre `cat` *fichier* avec `echo` *chaine*

5.3.4 Affichage paginé du contenu d'un fichier texte avec `more`/`less`

`more` *`liste_de fichiers`*

affiche le contenu des fichiers de la liste (avec contrôle du défilement)

`less` *`liste_de fichiers`*

préférable sous linux (défilement arrière possible)

Requêtes sous le pagineur

Entrée	avance d'une ligne
Espace	avance d'un écran
`/`*`motif`*	recherche la prochaine occurence de *`motif`* en avançant
`?`*`motif`*	recherche la prochaine occurence de *`motif`* en reculant
`q`	quitte l'affichage (nécessaire avec `less`)

Nombreuses variables d'environnement associées à `less` :

— Attention aux effets de la variable **`LESSOPEN`** : post-traitement du flux !

— Selon le codage des caractères, **`LESSCHARSET`**`=utf-8` ou `iso8859`

Rappel : `less` = pagineur utilisé par la commande **`man`**

5.3.5 Début et fin d'un fichier texte avec `head` et `tail`

`head/tail` *[options] [liste_de fichiers]*

`head -n nb fichier` affiche les `nb` premières lignes de `fichier`

`tail -n nb fichier` affiche les `nb` dernières lignes de `fichier`

`tail -n +11 fichier` affiche à partir de la ligne 11

5.3.6 Repliement des lignes d'un fichier texte avec `fold`

`fold` *[options] [liste_de fichiers]*

`-w width` : longueur (80 par défaut)

`-s` ne coupe pas les mots (repliement sur les espaces)

5.3.7 Affichage de texte avec `echo`

`echo` *chaîne de caractères*

commande interne du shell ou primitive (*built-in*)

utilisée dans les fichiers de commande pour afficher des messages

5.3.8 Affichage des différences entre deux fichiers texte avec `diff`

`diff` *`fichier_1 fichier_2`*

option **`-b`** ignore les différences portant sur les blancs

option **`-y`** affiche en deux colonnes

`vimdiff` pour éditer 2 fichiers en parallèle (changer de fenêtre : **`^W w`**)

5.3.9 Affichage de la partie texte d'un fichier binaire avec `strings`

`strings` [*`options`*] *`fichier`*

5.3.10 Affichage d'un fichier binaire avec `od`

`od` [*`options`*] [*`liste_de fichiers`*]

***o**ctal **d**ump*

formats d'affichage introduits par **`-t`**

`-t d4` pour des entiers sur 4 octets **`-t f4`** pour des flottants sur 4 octets

6 Environnement réseau

6.1 Courrier électronique

Commandes de gestion du courrier :

— en mode texte : `mail`, `elm`, `alpine`, `mutt`
autres outils gérant le courrier : l'éditeur `emacs`

— en mode graphique : les navigateurs (`mozilla-thunderbird`, ...).

— à distance : accès à sa boîte aux lettres personnelle via un navigateur (après authentification) grâce à un service de `webmail`

Exemple d'adresse électronique :

***Prenom.Nom*@etu.upmc.fr**

6.2 Connexion à distance via `slogin`

Connexion sur une machine distante grâce à la commande sécurisée **`slogin`**.

Authentification sur la machine distante par mot de passe ou échange de clefs.

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

`slogin` **`etu1@`**`sappli1.datacenter.dsi.upmc.fr`

ne pas oublier le login, sauf si identique sur la machine locale

Option **`-X`** pour autoriser les applications graphiques (fenêtres X11) via `ssh`

Lancement de commandes sur la machine distante :

`ssh` ***`user@dist_host.domain dist_cmd`***

`ssh etu1@sappli1.datacenter.dsi.upmc.fr ls ~lefrere/M1/Doc`

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

Copie de fichiers personnels entre deux machines, sans ouvrir de session sur la machine distante, via **`scp`** (fournir le mot de passe à chaque commande)

Syntaxe de `cp` mais préfixer le chemin d'accès des fichiers distants par ***`user@dist_host.domain:`***

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

Session `sftp` (***s**ecure **f**ile **t**ranfert **p**rotocol*) pour plusieurs transferts

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

- Après authentification sur le serveur distant,
- navigation distante : **`cd`**
- navigation locale : **`lcd`**
- importation de fichiers distants : **`get`** *`dist_file`*,
- exportation de fichiers vers la machine distante : **`put`** *`local_file`*
- **`exit`** ou **`quit`** pour terminer la session `sftp`.

6.4 Explorateurs et téléchargement

Navigateurs Web (`lynx`, `firefox`, `opera`, `konqueror`, `amaya`, ...)

Protocoles : **`ftp`** (***F**ile **T**ransfer **P**rotocol*), **`http`** (***H**ypertext **T**ransport **P**rotocol*), ou **`https`** (sécurisé par cryptage).

Ressources localisées grâce à une *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`

en ligne de commande : **`wget`** ou **`curl`** pour télécharger des fichiers

par exemple :

`wget` `"http://ftp.g95.org/v0.92/g95-x86-linux.tgz"`

7 Commandes avancées de gestion des fichiers

7.1 Découpage de fichiers avec `split` et `csplit`

`split [-l nb_de_lignes] fichier`

Découpage d'un fichier en sous-ensembles de **nombre de lignes** fixé (1000 par défaut)

`split -l 100 fic.txt` ⇒ `xaa`, `xab`, `xac`, ...`xba`, ...

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

Découpage *offset* lignes après la rencontre du **motif** *exp*, limité à $n + 1$ coupures

⇒ `xx00`, `xx01`, `xx02`, ..., `xx10`, ... par défaut

`csplit -f fic -b '%d.f90' complet.f90 '/end  *module/+1' '{*}'`

Découpage du fichier source fortran `complet.f90` par modules

1 ligne après **`end module`**, nombre illimité de coupures grâce à `'{*}'`

préfixe `-f` **`fic`** et suffixe avec format (voir `printf`) `-b` **`'%d.f90'`**

⇒ `fic1.f90`, `fic2.f90`, ...

7.2 Recherche de fichiers dans une arborescence avec `find`

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

Recherche **récursive** dans toute la branche sous *répertoire*

Commande très puissante : **critères** de sélection nombreux

pouvant être combinés pour former une expression évaluée à vrai ou faux.

`-name` *motif*	nom selon un motif (à protéger du shell)
`-iname` *motif*	version de `-name` ignorant la casse
`-size` *entier*`[ckM]`	taille en octets (`c`), kiloctets (`k`), mégaoctets (`M`)
`-newer` *fichier*	plus récent qu'un fichier
`-type` *T*	de type donné (`f`=ordinaire, `d`=répertoire)

Les **actions** les plus usitées sont :

`-print`	affiche la liste des fichiers (un par ligne)
`-ls`	affiche la liste des fichiers avec leurs attributs (comme `ls -dils`)
`-exec` *cmd*	exécute la commande unix *cmd* sur les fichiers sélectionnés

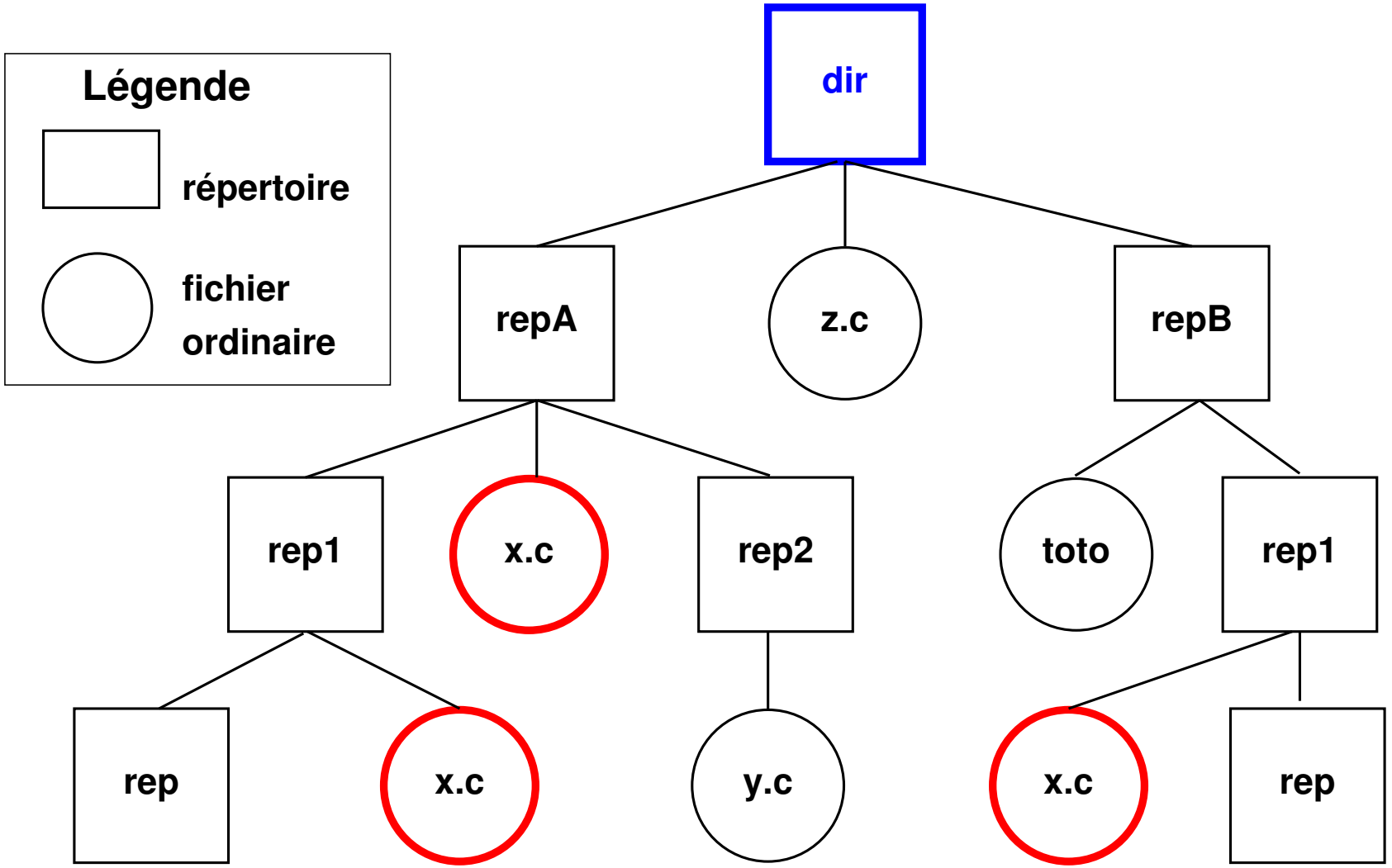


FIGURE 6 – **`find . -name x.c -print`** si **`dir`** est le répertoire de travail

$\Rightarrow$ trois fichiers

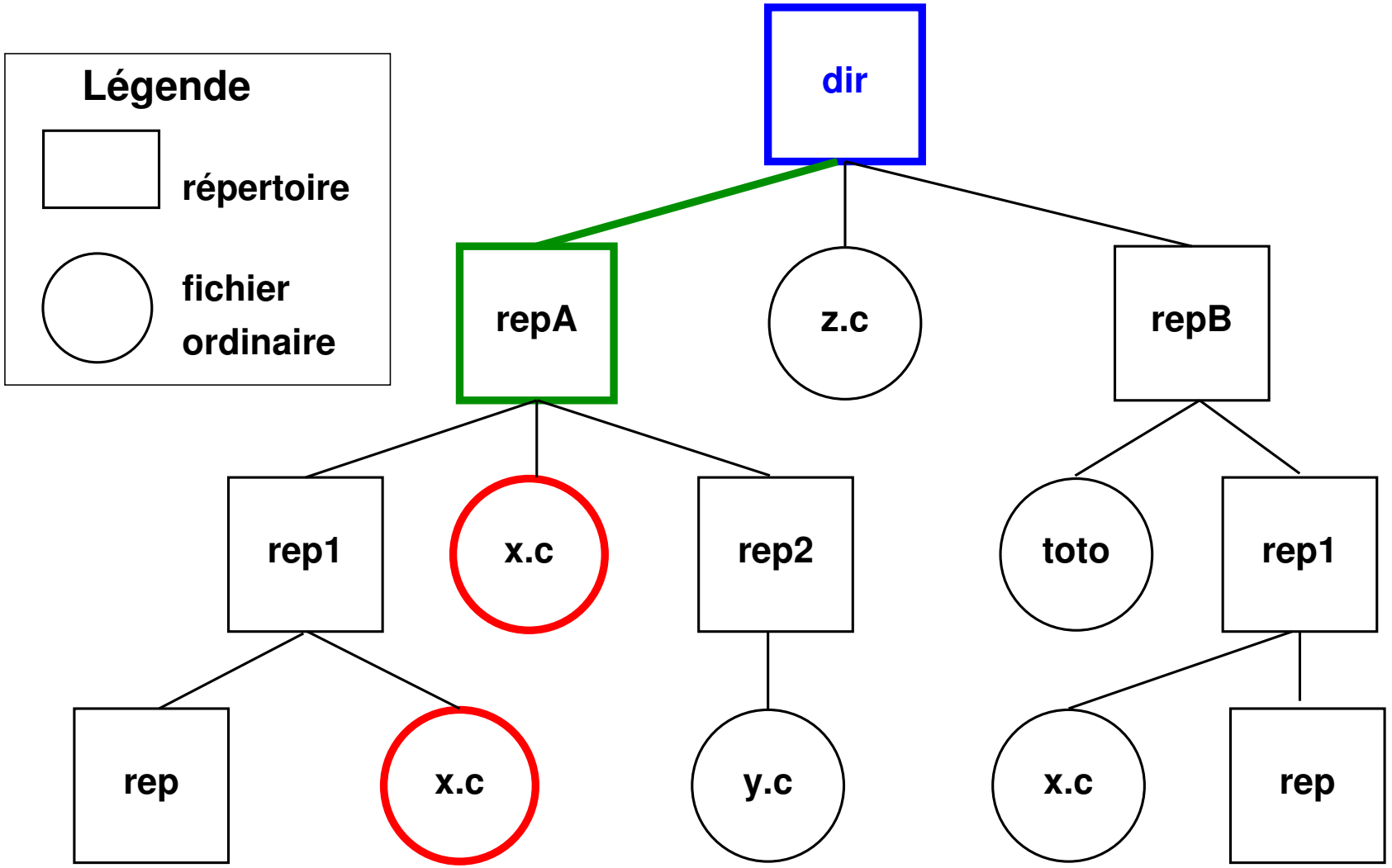


FIGURE 7 – **`find repA -name x.c -print`** à partir de **`repA`**

$\Rightarrow$ deux fichiers

Exemples de recherches avec find

`find . -name a.out -print`

affiche la liste des fichiers nommés `a.out` sous le répertoire courant

(dans toute la hiérarchie en dessous de ce répertoire)

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

(le shell ne doit pas interpréter le caractère `*`)

`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 {} \;` (syntaxe délicate de `exec`)

recherche les fichiers `a.out` et les supprime :

- `{}` désigne le nom de chacun des fichiers trouvés (avec son chemin)
- `\;` indique la fin de la commande à appliquer à chaque fichier.

Ne pas oublier le premier argument de find : le noeud (répertoire) de départ

Ne pas confondre avec `ls -R`

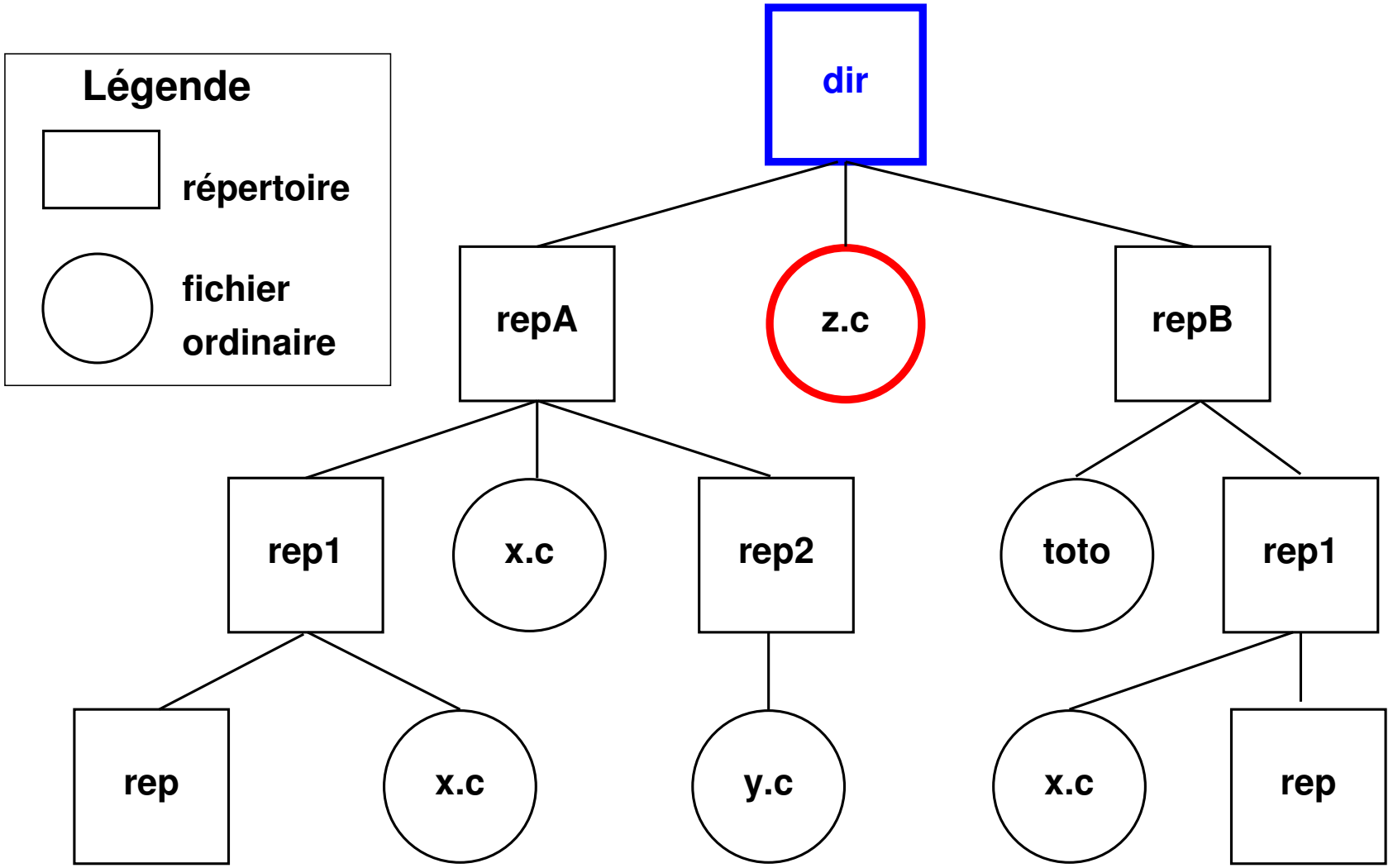


FIGURE 8 – **`find . -name *.c -print`** ($\star$ interprété par le shell) $\Rightarrow$ **`z.c`**

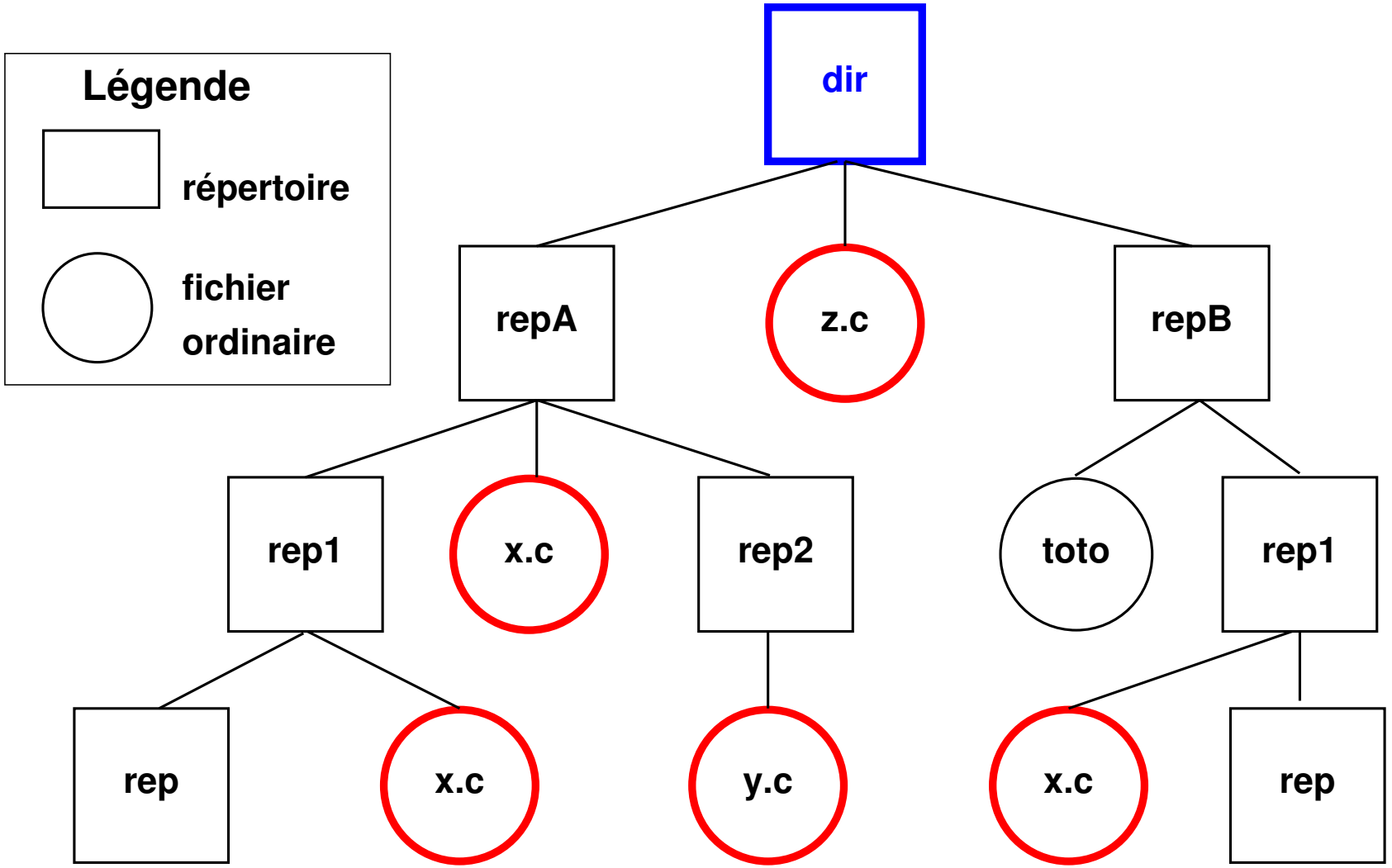


FIGURE 9 – **`find . -name '*.c' -print`** ⇒ cinq fichiers

7.3 Archivage d'arborescence avec `tar`

`tar` ***`options archive [répertoire]`***

Principales actions possibles (une et une seule) :

- **`-c`** (***c****reate*) création de l'archive à partir de l'arborescence
- **`-t`** (*lis****t***) liste des fichiers tels qu'ils seront extraits
- **`-x`** (***x****tract*) extraction des fichiers pour restaurer l'arborescence

Autres options combinables :

- **`-v`** (***v****erbose*) affiche des informations complémentaires
- **`-f`** *`archive`* (***f****ile*) précise le nom du fichier d'archive utilisé (nécessaire)
- **`f -`** si entrée standard (`tar x`) ou sortie standard (`tar c`)
- **`z`** ou **`j`** avec dé/compression (`gzip` ou `bzip2`) du fichier `.tar`
- **`--exclude=`***`motif`* sauf les fichiers répondant au motif indiqué

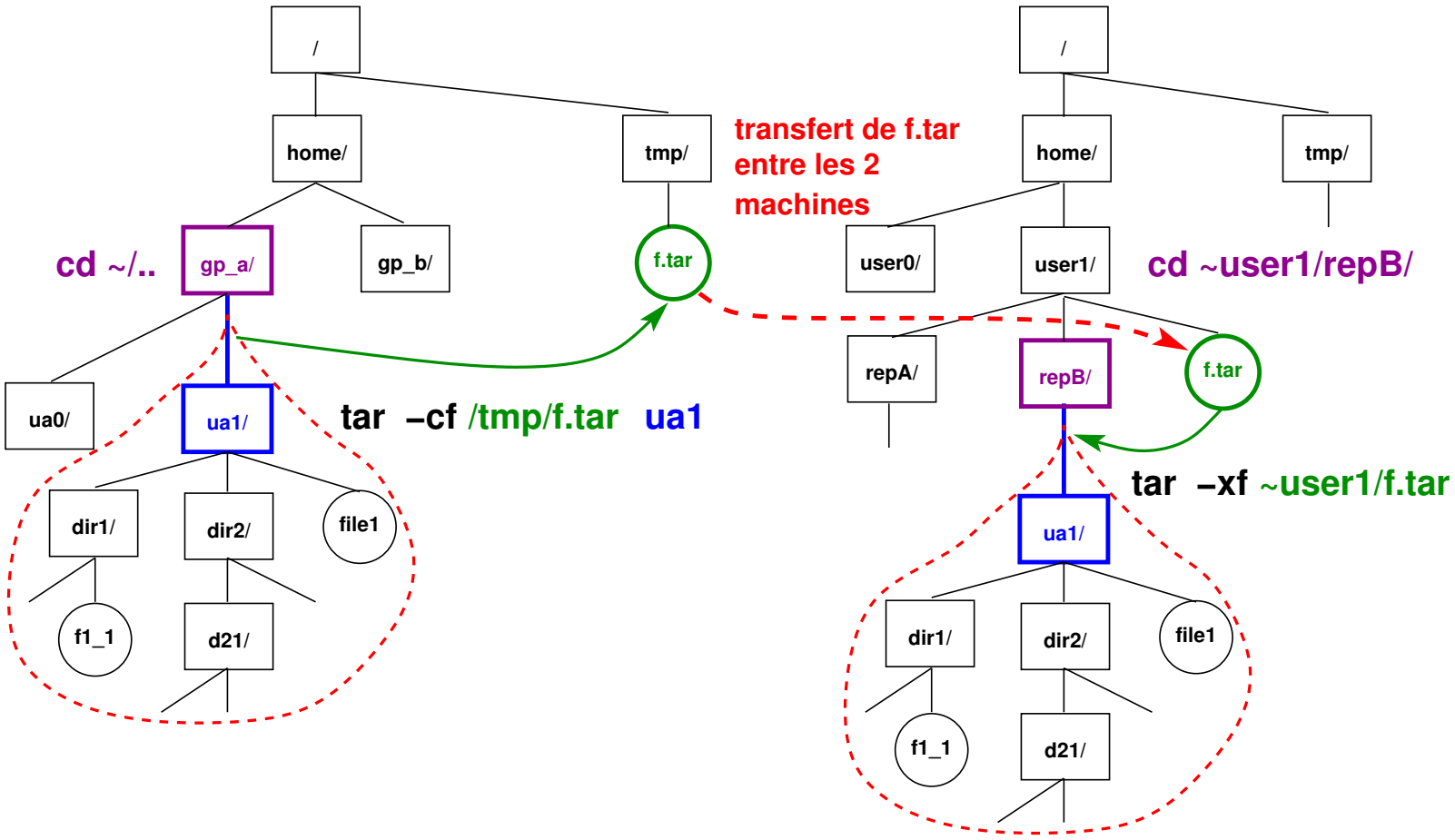


FIGURE 10 – Transfert de branche via **`tar`** : création de l'archive `f.tar`, transfert de l'archive et extraction sous `repB`

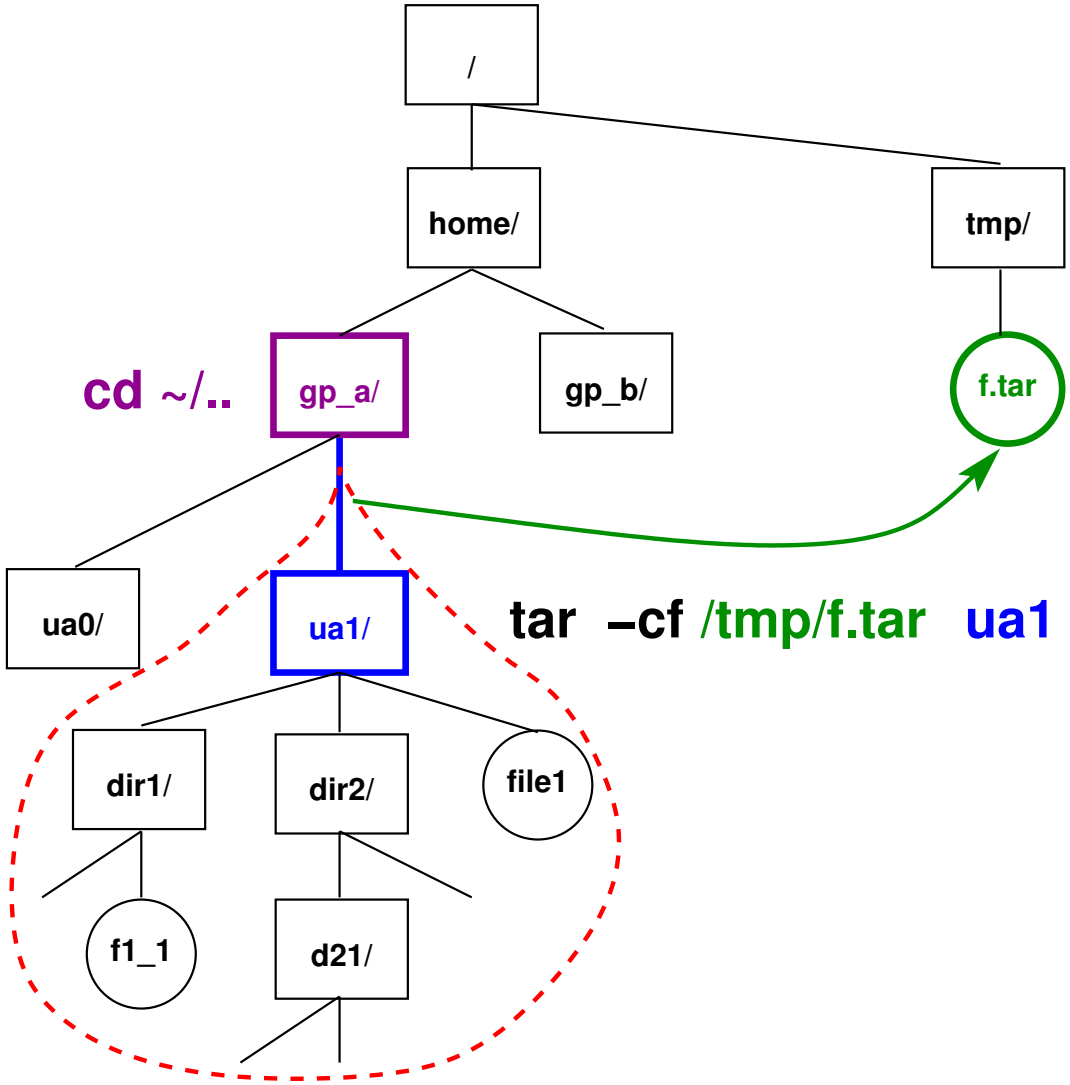


FIGURE 11 – Création (sous `/tmp`) de l'archive **`f.tar`** de la branche de l'utilisateur **`ua1`** :

1) `cd ~/..`

2) `tar -cf /tmp/f.tar ua1`

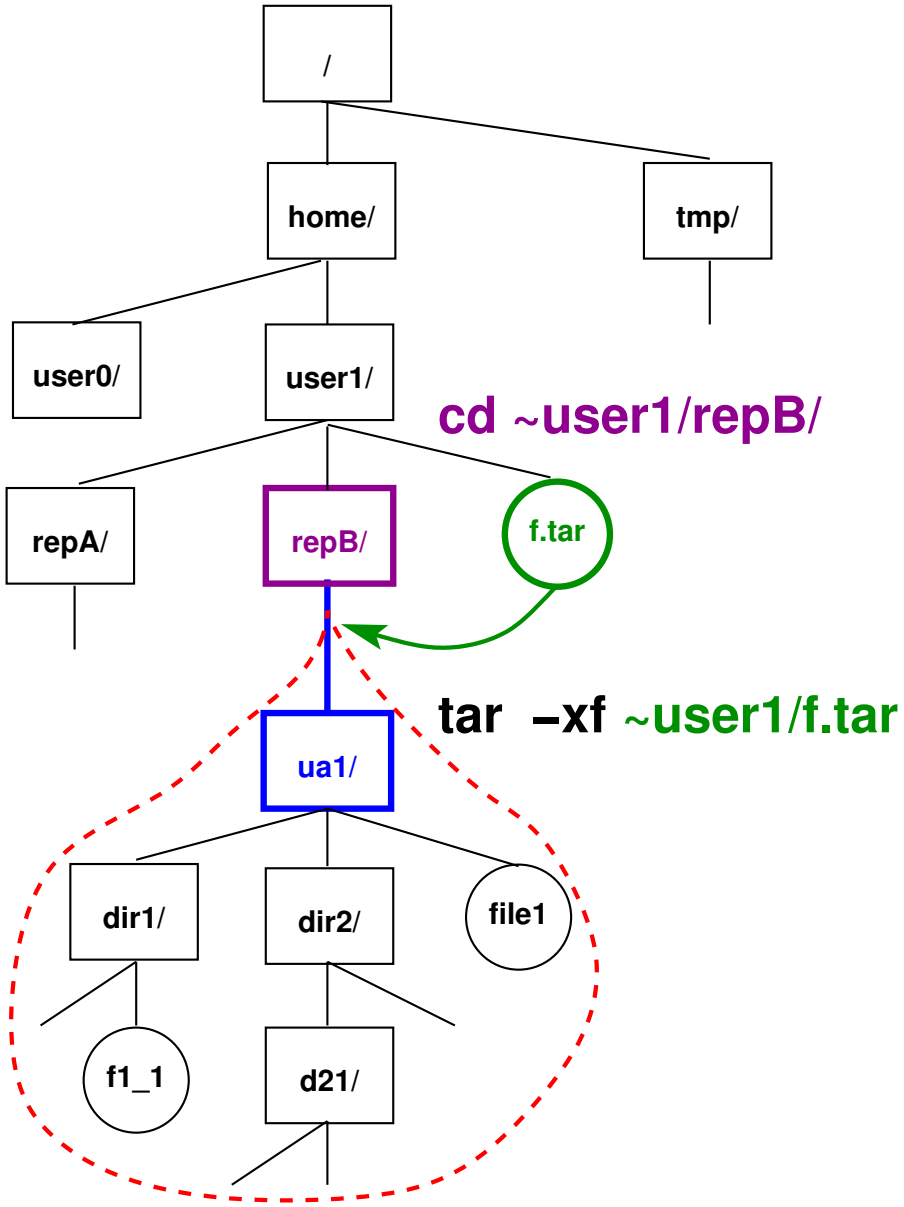


FIGURE 12 – Restauration de branche à partir de l'archive :

1) `cd ~/repB/`
2) `tar -xf ~/f.tar`

Exemples d'usage de `tar`

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

archive toute l'arborescence de l'utilisateur `user` dans `archive.tar`

(se placer un niveau au-dessus de la branche à archiver)

```
tar -tf /tmp/archive.tar
```

affiche la liste des fichiers archivés dans `archive.tar`

```
tar -xvf /tmp/archive.tar
```

restaure toujours l'arborescence dans le répertoire **courant** (à partir de l'archive)

(se placer au niveau où « greffer » la branche à restaurer)

Remarques : éviter les chemins absolus dans les sauvegardes,

sinon les fichiers seront obligatoirement restaurés au même endroit.

l'option avec argument `-f fichier_archive` est en fait obligatoire.

Copie de branche via un tube (`-` désigne entrée/sortie standards) :

```
tar -cf - . | (cd dest_dir ; tar -xvf - )
```

7.4 Copies et synchronisation de fichiers avec `rsync`

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

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

outil de copie plus puissant que **`scp`** $\Rightarrow$ synchronisation de répertoires

— sur une même machine ou à distance

— très rapide car ne transmet que les différences et compression possible

— nombreuses options pour les sauvegardes et les miroirs

`-r` récursif **`-v`** (***v****erbose*) prolixe **`-z`** (***z****ip*) compression avant transfert

`-u` (***u****pdate*) mise à jour : ne transmet que les fichiers plus récents côté source

`-t`, **`-p`** conserve la date (***t****ime*), les droits (***p****ermissions*)

`--exclude=`*`motif`* sauf les fichiers répondant au motif indiqué

`-n` essai à vide avant d'activer

Remarque : rôle des / terminaux

`rsync -r rep1 user@host:/tmp` crée le répertoire `/tmp/rep1/`
sur la machine distante et y recopie récursivement le contenu de `rep1`

`rsync -r rep1/ user@host:/tmp` recopie récursivement le contenu
de `rep1/` dans `/tmp/` sur la machine distante sans y créer de niveau `rep1`

`rsync -r rep1 user@host:/tmp` est donc équivalent à
`rsync -r rep1/ user@host:/tmp/rep1/`

Exemple

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

met à jour (`u`) récursivement (`r`) le répertoire local `~/unix-mni` à partir du
répertoire `~user/mni/unix/` du serveur en conservant droits (`p`) et dates (`t`),
mais sans transférer les fichiers de sauvegarde de suffixe `~` (`--exclude`)

8 Droits d'accès aux fichiers

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

r	***r**ead*	lecture
w	***w**rite*	écriture
x	***ex**ecute*	exécution
-		droit refusé

8.1 Affichage des droits d'accès avec `ls -l`

Exemple : `ls -l ~lefrere/M1/Config/`

```
drwxr-xr-x 2 lefrere personnel 1024 sep 17  2009 lisp
-rwxr-xr-x 1 lefrere personnel 1076 oct  7  2009 MNI.bash_profile
-rwxr-xr-x 1 lefrere personnel 3101 oct 22  2009 MNI.bashrc
lrwxrwxrwx 1 lefrere personnel   15 sep 15 17:40 motd -> motd.16sept2010
-rw-r--r-- 1 lefrere personnel  434 sep 15 21:18 motd.16sept2010
```

première colonne : **d** si répertoire **l** si lien (***l**ink*) symbolique (raccourci vers **->**)

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

Les permissions peuvent être aussi représentées de la valeur exprimée en base huit (octal) des droits d'accès (`r=4, w=2, x=1`).

Permission **s** (***s**etuid bit* ou ***s**etgid bit*) à la place de **x** sur un fichier exécutable

$\Rightarrow$ droits du propriétaire (*effective user*) ou du groupe lors de l'exécution du fichier.

Ex : la commande **passwd** (changement de mot de passe) doit permettre à n'importe quel utilisateur authentifié d'écrire dans **/etc/passwd**, mais on ne peut pas donner ce droit en permanence.

```
-rw-r--r--  1 root root  1091 Jun 30 09:16 /etc/passwd
-r-s--x--x  1 root root 15540 Jun 20  2004 /usr/bin/passwd
```

8.2 Changement des droits d'accès avec `chmod`

`chmod` ***`mode liste_de_fichiers`*** où *mode* représente~:

— la portée, **`u`**, (***u****ser*), **`g`**, (***g****roup*), **`o`**, (***o****thers*) ou **`a`** (***a****ll*).

— suivie de **`=`** (définit les droits), **`+`** (ajoute un droit), ou **`-`** (enlève un droit),

— suivi de la permission **`r`**, **`w`**, ou **`x`**.

Exemple 1 : **`chmod go-r fichier`**

supprime les droits de lecture au groupe et aux autres

Exemple 2 : **`chmod u+w,go-w fichier`**

donne le droit d'écriture au propriétaire et le supprime au groupe et aux autres

Droits par défaut : la commande interne `umask` (**u**ser mask) affiche ou fixe les droits par défaut (masque en octal : complément à 7, ou droits symboliques avec **`-S`**)

affichage des droits

`umask`	`umask -S`
`022`	`u=rwx,g=rx,o=rx`

fixation des droits par défaut

`umask 027`	`umask -S u=rwx,g=rx,o=`

8.3 Signification des droits sur les répertoires

r nécessaire pour afficher la liste des fichiers du répertoire

w permet d' ajouter, de renommer, de supprimer des fichiers dans le répertoire (pas nécessaire pour modifier le contenu d'un fichier)

x permet d'agir sur les fichiers du répertoire, à condition de connaître leurs noms (même si on ne peut pas afficher leur liste) : par exemple traverser le répertoire, afficher les attributs des fichiers du répertoire

Exemple :

```
drwx--x--x  42 lefrere personnel 4096 sep 20 18:17 lefrere
```

peut être traversé par tout le monde pour accéder à `lefrere/M1/`

mais seul son propriétaire peut lister son contenu ou le modifier

NB : utiliser `ls -ld repertoire` pour afficher les droits d'un répertoire

(au lieu de ceux des fichiers du répertoire)

9 Édition de fichiers textes

9.1 Les éditeurs sous unix et leurs modes

9.1.1 Éditeurs sous unix

— ligne : **ex** et **ed** très robustes (terminal quelconque), mais d'interface rigide

— pleine page : (nécessitent une connaissance du terminal utilisé)

– **vi** sur-couche de ex, très puissant, présent sur tous les unix,
version **vim** sous linux, éditeur sensible au langage (C, fortran, latex, ...)
avec mise en valeur de la syntaxe par des couleurs

– **emacs** encore plus puissant, mais plus gourmand en ressources

— en environnement graphique multifenêtres, avec menus, gestion de la souris, ...
xemacs, **gvim**, **gedit**, **kwrite**...

attention au codage des fichiers texte : ASCII/ISO-8859-1/UTF-8

$\Rightarrow$ transcodage au vol par l'éditeur dans certains cas

9.1.2 Les modes des éditeurs

Deux modes principaux :

— **commande** : les caractères saisis sont **interprétés** comme des ordres (requêtes)
⇒ immédiatement exécutés

— **insertion** : les caractères saisis sont directement **insérés** dans le fichier.

Le mode **par défaut** est :

— le mode **commande** sous **vi** ⇒ déroutant au premier abord
passage en mode insertion par une requête

— le mode insertion sous **emacs**
requêtes introduites par des caractères de contrôle : Ctrl, Échap
exemple : ^X^C pour terminer l'édition emacs

9.2 Principes de l'éditeur `vi`

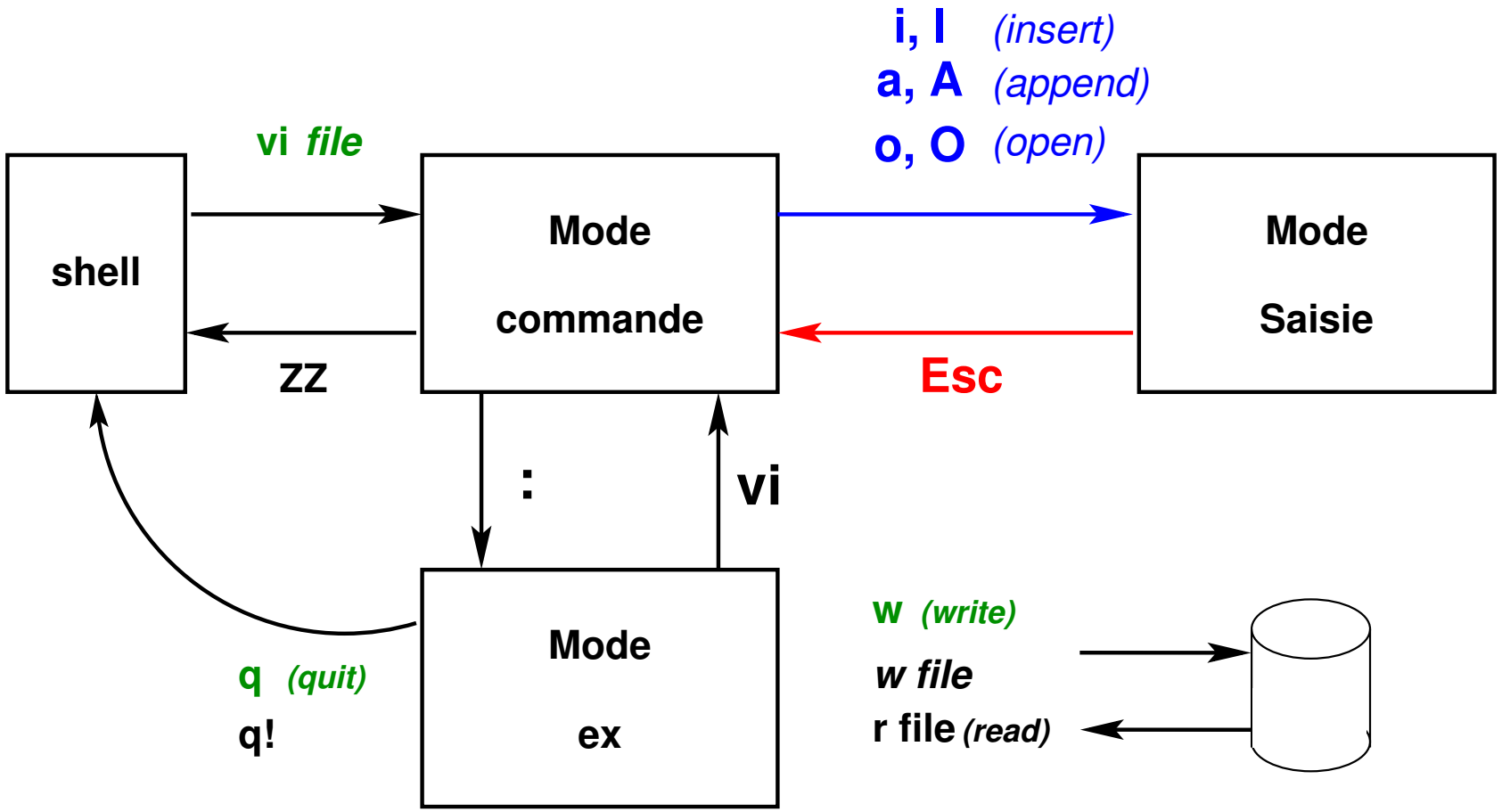


FIGURE 13 – Modes de fonctionnement de `vi`

— plusieurs requêtes pour passer en mode saisie :
  – **a** et **A** (***a**ppend*) ajout,
  – **i** et **I** (***i**nsert*) insertion,
  – **o** et **O** (***o**pen*) ouverture de ligne.

— une seule méthode pour revenir au mode commande :
  touche d'échappement **Échap** (*escape*)

Un troisième mode, le mode **dialogue** est accessible de façon temporaire (affichage sur la ligne d'état en bas du terminal) depuis le mode commande de vi pour :

— passer des requêtes ex via **:**

— rechercher de motifs dans le texte (via **/** ou **?**)

Requête activée par Entrée,
puis retour immédiat en mode commande.

9.3 Principales requêtes de l'éditeur `vi`

Déplacements

$\leftarrow \uparrow \downarrow \rightarrow$ les quatre flèches

0 première colonne, **$** fin de ligne

d'un écran : **^F** (***f**orward*) en avant, **^B** (***b**ackward*) en arrière

Modifications

r*c* (***r**eplace*) remplace un caractère par **un** (seul) autre

cw (***c**hange **word***) 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

Mise en mémoire

`y``y` (***y**ank*) mise en mémoire d'une ligne

`y``w` (***y**ank **w**ord*) mise en mémoire du mot à droite du curseur

Restitution de la mémoire tampon

`p` (***p**aste*) restitue après ; **`P`** restitue avant

Divers

`.` réitère la commande précédente

`u` (***u**ndo*) annule la précédente commande

`%` bascule le curseur d'un délimiteur à l'autre dans les paires `( ) [ ] { }`

`J` (***j**oin*) concatène la ligne suivante à la ligne courante

`^L` raffraîchit l'affichage

Recherche de chaînes de caractères

/*motif* recherche *motif* vers l'avant

?*motif* recherche *motif* vers l'arrière

n (***n**ext*) recherche l'occurence suivante

Répétitions de requêtes

La majeure partie des requêtes vi peut être préfixée par un entier qui indique combien de fois elle est appliquée :

25x détruit 25 caractères

20i+Échap insère 20 signes +

Auto-complétion des mots pendant la saisie (vim)

En mode saisie, les mots peuvent être complétés automatiquement s'ils existent déjà dans le texte :

^P (***p**revious*) si le mot est présent avant dans le texte

^N (***n**ext*) si le mot est présent après dans le texte

9.4 Requêtes ex

9.4.1 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`

9.4.2 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*	. = ligne courante
:*n*, $	$ = dernière ligne
:*n1*,+*n2*	+ adressage relatif
:*n1*,−*n2*	− adressage relatif
:%	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

9.4.3 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 (première occurrence de la ligne)
s/ch1/ch2/g	(***s****ubstitute*) substitue une chaîne à une autre (toutes les occurrrences de la ligne)
g/motif/	(***g****lobal search*) recherche un motif

9.4.4 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
:.,$d	détruit de la ligne courante à la fin du fichier
:1,5co32	copie les lignes 1 à 5 après la ligne 32
:1,3m$	déplace les trois premières lignes en fin de texte
:g/motif/d	recherche toutes les occurrences de motif dans le texte et détruit les lignes où il est présent

9.5 Configuration de **vi**

Nombreuses options accessibles par **:set**

:set nu	affiche les numéros des lignes (ou :set number)
:set nonu	n'affiche pas 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 du texte (latin1, utf-8, ...)
:set all	affiche l'état des options en cours

Fichier de configuration où stocker ces options : **~/.exrc**

Cas de **vim** (*improved*) : fichier de configuration **~/.vimrc**

10 Redirections et tubes

10.1 Flux standard

Commande UNIX $\Rightarrow$ trois flux standard de données :

descripteur	flux	défaut
0	entrée standard	le clavier
1	sortie standard	l'écran
2	sortie d'erreur standard	l'écran

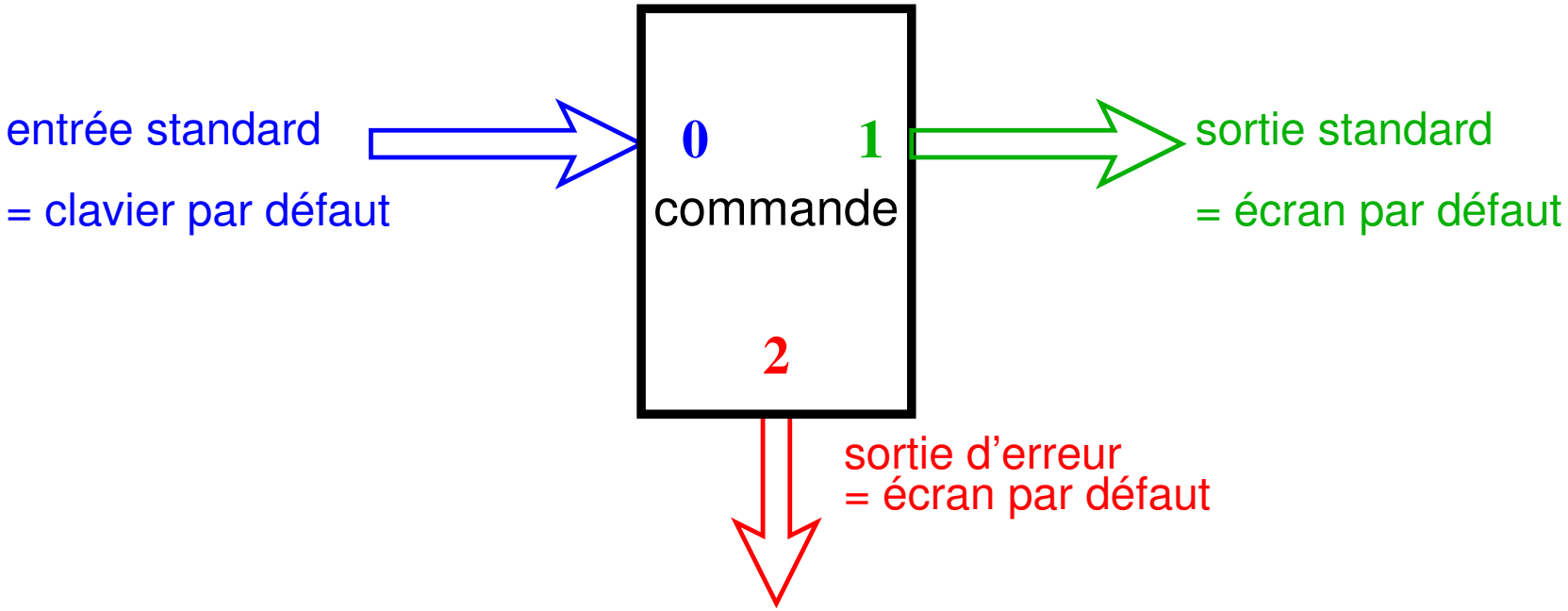


FIGURE 14 – Flux standard de données associés à une commande

10.2 Redirections

— Au lieu d'une saisie au clavier et d'un affichage à l'écran,
stocker de façon permanente l'information d'entrée ou de sortie
⇒ rediriger les flux standards à partir ou vers des **fichiers**

— Combiner des commandes de base pour effectuer des traitements complexes
⇒ rediriger les flux standards vers les entrées/sorties d'**autres commandes**.
(mécanisme des tubes ou pipe-lines)

⇒ grande souplesse du système UNIX

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

syntaxe

commande > fichier

Le fichier résultat est créé.

syntaxe

commande >> fichier

pour ajouter le résultat à la fin du fichier

Exemples :

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

```
ls *.f90 > liste_f+c
ls *.c  >> liste_f+c
```

Attention : le shell interprète très tôt les redirections

⇒ ne pas rediriger la sortie vers le fichier d'origine

`cat -n fic1 > fic1` **efface** le contenu du fichier `fic1`

Solution : `cat -n fic1 > tmp ; mv tmp fic1`

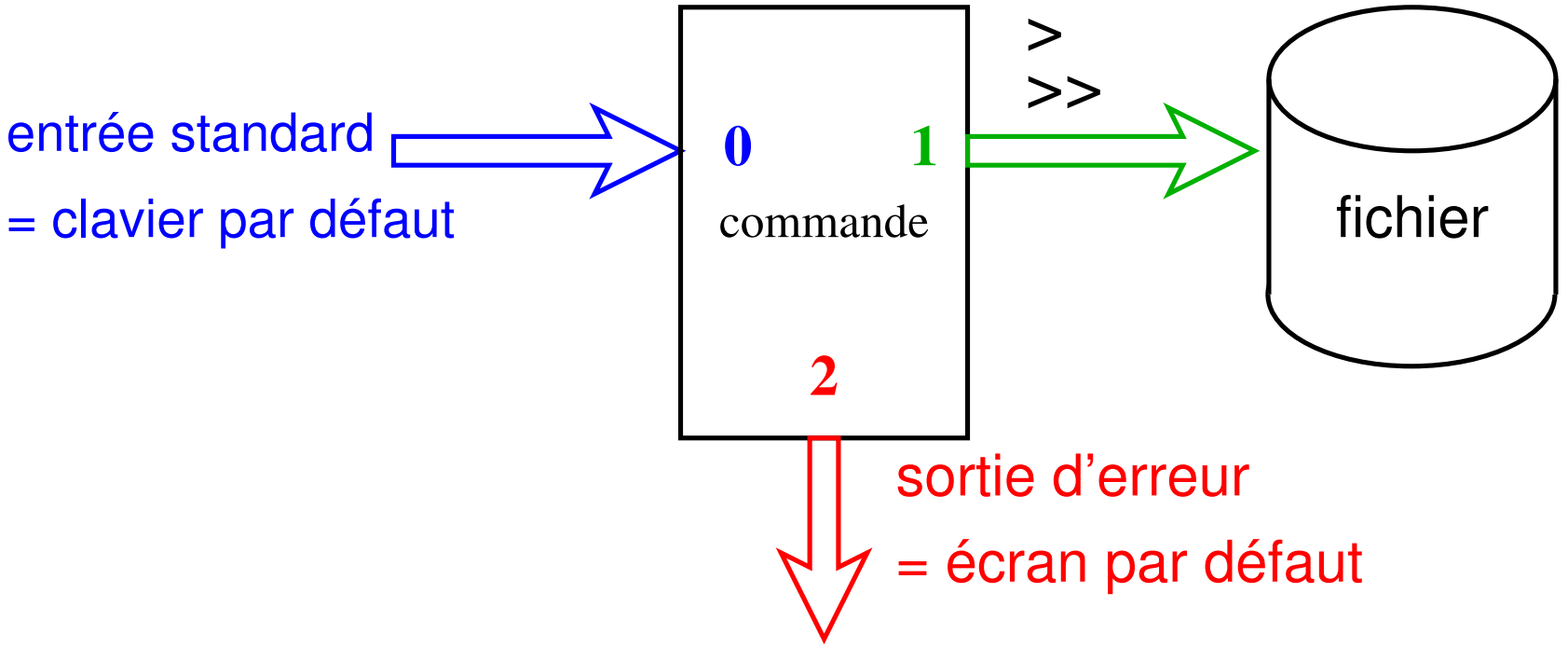


FIGURE 15 – Redirection de la sortie standard

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

syntaxe

commande < fichier

le fichier doit exister au préalable.

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

```
a.out < entrees
```

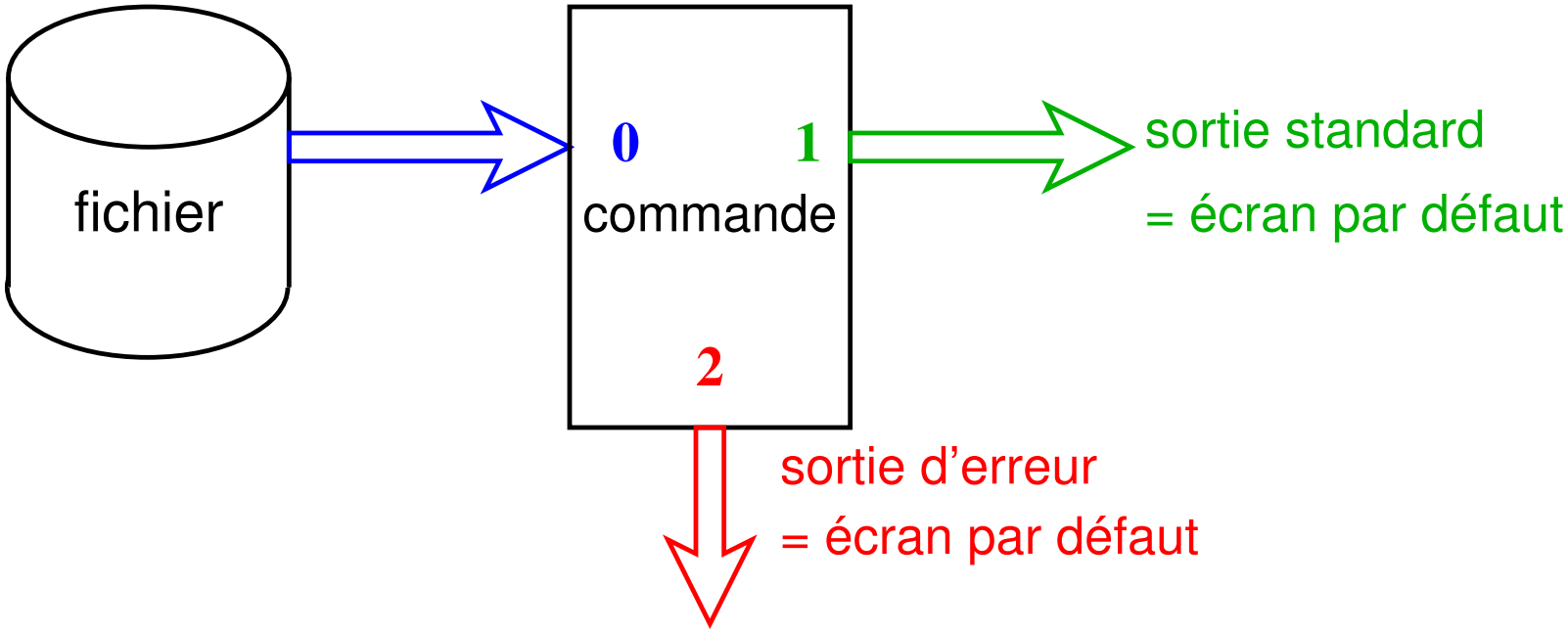
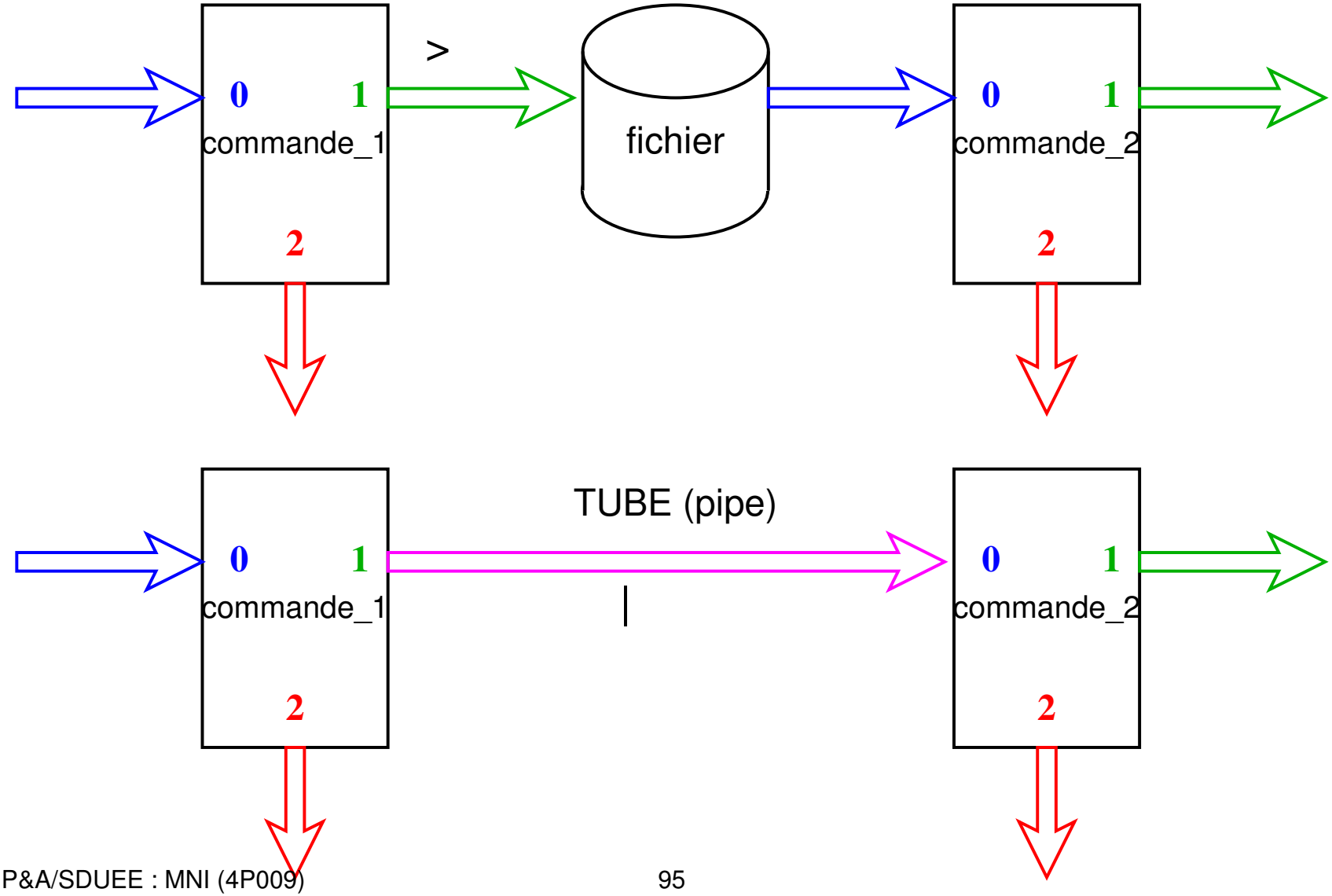


FIGURE 16 – Redirection de l'entrée

10.3 Tubes ou *pipes* (|)



Appliquer deux traitements successifs à un flux de données :

— Méthode **séquentielle** avec fichier intermédiaire :

```
commande_1 > fichier                    => attente éventuelle
commande_2 < fichier
rm fichier
```

— **Traitement à la chaîne** en connectant les deux processus par un **tube** ou *pipe* = zone mémoire ⇒ communication synchronisée entre les 2 processus

syntaxe

```
commande_1 | commande_2
```

plus **rapide** que le traitement séquentiel

Exemple 1 : affichage paginé de la liste des fichiers du répertoire courant

Méthode séquentielle (à éviter)

```
ls -l > liste
more liste
rm liste
```

Chaînage avec tube (à préférer)

```
ls -l | more
```

Exemple 2 : affichage de la 12e ligne du fichier `toto`

Méthode séquentielle (à éviter)

```
head -n 12 toto > tmp1
tail -n 1 tmp1
rm tmp1
```

Chaînage avec tube (à préférer)

```
head -n 12 toto | tail -n 1
```

Cas de plusieurs redirections

L'ordre des redirections sur la ligne est indifférent (avec une seule commande)

```
commande < entree > sortie
commande > sortie < entree
```

Avec un tube, ne pas détourner le flux : pas de redirection sur des fichiers en sortie de la première commande ni en entrée de la seconde

```
commande_1 < entree | commande_2 > sortie
```

```
commande_1 > sortie | commande_2 < entree
```

10.4 Compléments

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

syntaxe

```
commande 2> fichier
```

Attention : pas d'espace entre 2 et >

syntaxe

```
commande 2>> fichier
```

pour ajouter les erreurs à la fin du fichier.

Exemple : stockage des diagnostics d'une compilation dans un fichier pour éviter le défilement à l'écran (afin de localiser d'abord la première erreur)

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

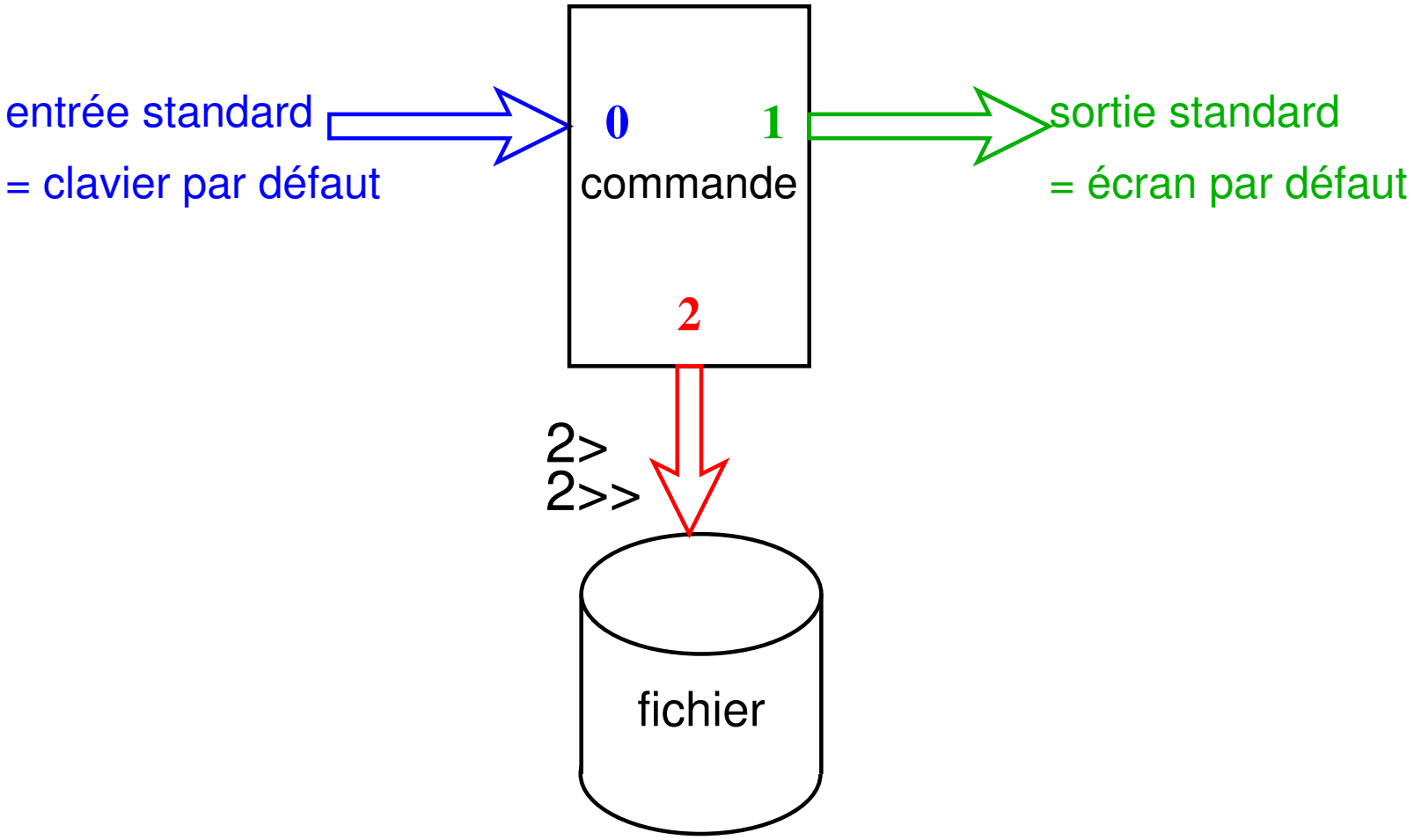


FIGURE 18 – Redirection de l'erreur

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

Regroupement dans un même flux de la sortie standard et de la sortie d'erreur :

syntaxe

```
commande 2>&1
```

Exemple (on suppose que `/etc/motd` est accessible) :

```
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

N.-B. : 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.

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

Répertoire **`/dev`** : *fichiers spéciaux* gérant des flux de données entre le calculateur et les périphériques (*devices*) : terminaux, imprimantes, disques, ...

`tty` affiche le nom du fichier spécial particulier attribué à un terminal
le fichier spécial **`/dev/tty`** désigne de façon générique le terminal attaché à la connexion.

`/dev/zero` = fichier spécial fournissant un flux de zéros binaires
$\Rightarrow$ utilisé en entrée pour écraser physiquement des informations.

`/dev/null` = fichier spécial « poubelle » (vide) ou trou noir !
$\Rightarrow$ utilisé pour se débarrasser de certaines sorties inutiles.

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

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

Exemple

`find` *rep* `-name "`*nom*`" -print` **`2> /dev/null`**

évite l'affichage des messages d'erreur
quand on tente d'accéder à des fichiers non autorisés.

10.4.4 Duplication de flux : `tee`

`tee` **duplique** le flux de son entrée standard vers le fichier passé en argument et vers sa sortie standard.
Exemple : `tee` permet de conserver une trace du résultat intermédiaire d'un tube :

`cmd_1 | tee f_intermediaire | cmd_2`

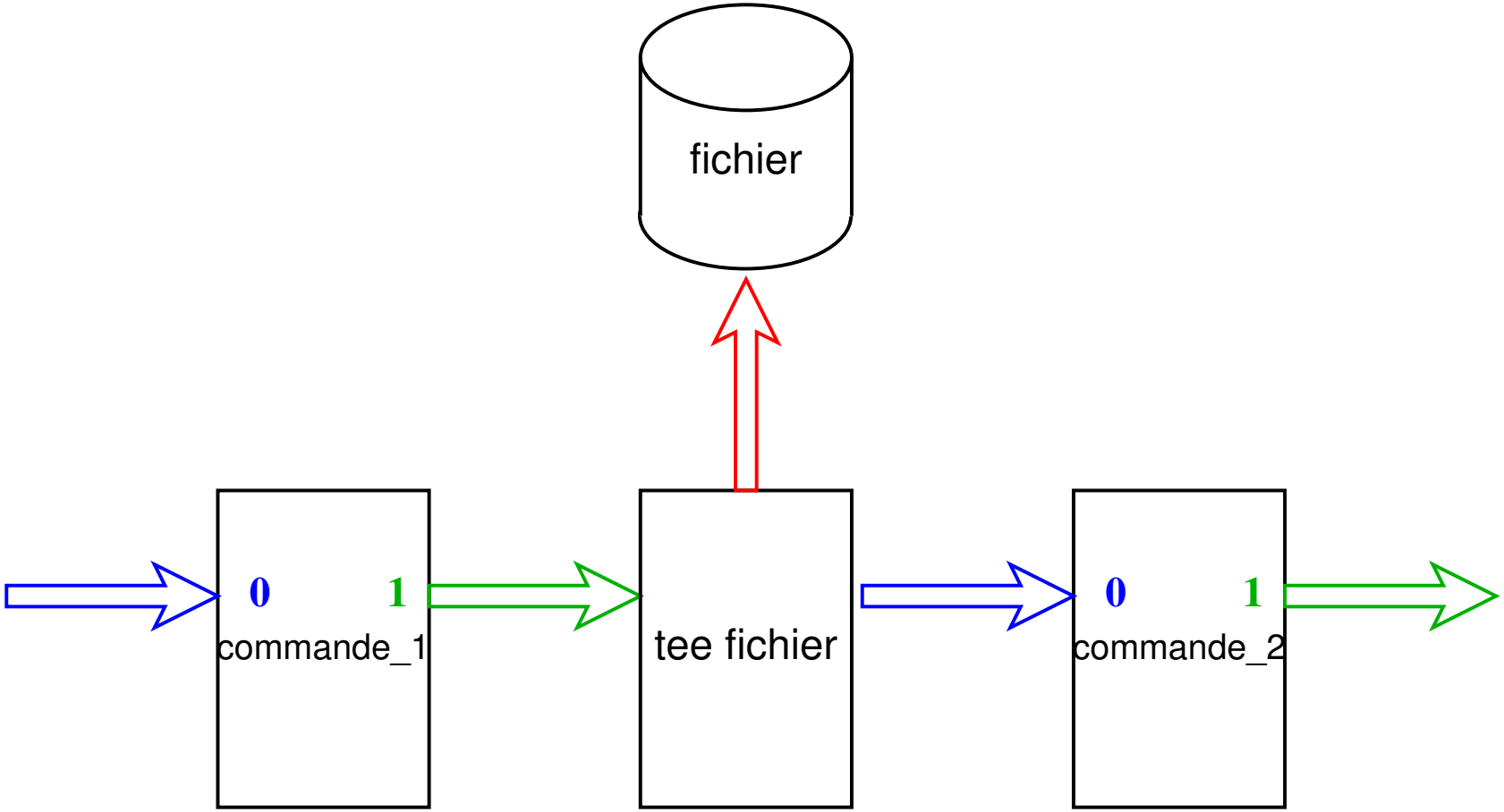


FIGURE 19 – Duplication d'un flux avec `tee`

10.4.5 Notion de document joint (<<)

Rediriger l'entrée d'une commande habituellement interactive

⇒ lui passer des requêtes depuis un fichier.

Faire suivre la commande par **<<** et un mot délimiteur quelconque (sans espace) ;

⇒ le texte qui suit est redirigé vers l'entrée de la commande,

jusqu'à la première ligne qui *commence par le délimiteur*.

Utilisé pour « embarquer » des données dans des fichiers de commande

Exemple de document joint (requêtes pour `vi`)

`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

11 Filtres élémentaires

11.1 Définition

filtre = commande qui lit l'entrée standard, effectue des transformations sur ces données et affiche le résultat sur la sortie standard.

Exemples de filtres : `cat`, `wc`, `tail`, `head`, `tr`, `sort`, `grep`, `sed`, `awk`...
mais `ls`, `date`, `vi`... ne sont pas des filtres.

Utilisations

— ***`filtre`*** + saisie des données d'entrée au clavier et **`^D`** pour terminer

— ***`filtre`*** **`<`** *`fic`* lit dans un fichier par redirection d'entrée

— mais beaucoup de filtres (sauf **`tr`**) admettent aussi comme paramètres un nom de fichier ou une liste de fichiers d'entrée :

```
filtre  fic1 fic2 ...
```

équivaut (presque car on perd ici les noms des fichiers) à :

```
cat  fic1 fic2 ... | filtre
```

Dans ce cas, le nom de fichier « **`-`** » représente l'entrée standard.

11.2 Classement avec `sort`

`sort` trie, regroupe ou compare toutes les **lignes** des fichiers passés en paramètre

Par défaut : ordre **lexicographique** sur tous les champs de la ligne

$\Rightarrow$ sensible aux variables de langue (`LANG` et `LC_`...)

Options :

- **`-r`** (***r**everse*) pour trier selon l'ordre inverse
- **`-f`** pour ignorer la casse (majuscule/minuscule)
- **`-n`** (***n**umeric*) pour trier selon l'ordre numérique croissant
- **`-u`** (***u**nique*) pour fusionner les lignes ex-æquo
- **`-k`***début*`[,`*fin*`]` classement selon les champs (***k**ey*)
  de numéros entre *début* (et *fin*)
- **`-t`***délim* choisit le séparateur de champs *délim* (blanc par défaut)
- **`-b`** (***b**lank*) pour ignorer les blancs en tête de champ (cas de blancs multiples)

Exemples

`sort /etc/passwd`	classe les lignes du fichier `/etc/passwd` par ordre lexicographique
`ls -l | sort -k8,8*`	classe les fichiers par ordre lexicographique
`ls -l | sort -k8,8*r`	classe les fichiers par ordre lexicographique inverse

* : date et heure sont formatées en seulement deux champs (mandriva, mais pas `sappli1` par défaut)

`ls -l | sort -k5,5n`	classe les fichiers par ordre de taille croissante
`ls -l | sort -k5,5n -u`	idem, mais n'affiche qu'un exemplaire pour une taille donnée (fusion des ex-æquo)
`wc -l * | sort -k1,1n`	classe les fichiers par nombre de lignes croissant
`sort -t: -k3,3n /etc/passwd`	classe les lignes du fichier `passwd` (séparateur de champ `:`) par numéro d'id croissant (champ 3)
`sort fic1 | cat pre - post`	présente le fichier `fic1` trié, précédé du fichier `pre` et suivi de `post`.

11.3 Transcription avec `tr`

`tr` *`jeu1 jeu2`* substitue à chaque caractère de l'ensemble fourni en premier paramètre son correspondant pris dans le deuxième ensemble.

`tr '123' 'abc'` change les **`1`** en **`a`**, les **`2`** en **`b`** et les **`3`** en **`c`**.

`tr -d` *`jeu`* pour supprimer (***d****elete*) les caractères d'un ensemble

`tr -s` *`jeu1 jeu2`* (***s****queeze-repeats*) pour supprimer les répétitions de caractères de l'ensemble **d'arrivée** (après substitution)

Restrictions

Filtre **pur** : n'admet pas de nom de fichier en paramètre $\Rightarrow$ redirections.

Attention aux différences selon les systèmes unix (manuel via `info tr`).

Travaille octet par octet $\Rightarrow$ pas encore compatible UTF-8

$\Rightarrow$ ne tient pas compte du contexte (ne traite pas des motifs)

$\Rightarrow$ séquentiel sans mémoire (avantage !)

Compléments

Peut utiliser les séquences de contrôle de l'ascii (**\r** =CR, **\n**= NL)

tr -d '\r' supprime les « retour chariot » des fins de lignes (issus de windows)

Peut utiliser des intervalles (ascii)

tr a-z A-Z

ou mieux des classes de caractères (avec les accentués selon `LC_COLLATE`)

tr '[:lower:]' '[:upper:]' (en iso-latin, pas en UTF)

11.4 Autres filtres élémentaires

head / **tail** affiche les lignes du début (entête) / de la fin du fichier

expand / **unexpand -a** traduit les tabulations en espaces et inversement

cut sélectionne des colonnes (champs) dans un flux

12 Expressions régulières ou rationnelles

Recherche de chaînes de caractères qui satisfont à un certain motif (*pattern*)

$\Rightarrow$ syntaxe particulière pour décrire des motifs **génériques** :

une *expression rationnelle*

Expressions rationnelles utilisées par les éditeurs **`ex`**, **`vi`** et **`sed`**, les filtres **`grep`** et **`awk`**, ainsi que `perl`, `python`, `php`, `JavaScript`...

Deux versions exclusives de la syntaxe :

— expressions rationnelles de base ***BRE** : **B**asic **R**egular **E**xpressions*

(`ex`, `vi`, `sed`, `grep`)

— expressions rationnelles étendues ***ERE** : **E**xtended **R**egular **E**xpressions*

(`awk`)

12.1 Signification des caractères spéciaux

. (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
* (étoile)	représente un nombre d'occurrences quelconque (**zéro**, une ou plusieurs occurrences) du caractère ou de la sous-expression qui précède

Ne pas confondre ces caractères spéciaux des expressions rationnelles avec les caractères génériques (*wildcards*) pour les noms de fichiers, `*` et `?` qui sont, eux, interprétés par le shell.

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

12.2 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

^a	une ligne commençant par un `a`
^a.*b$	une ligne commençant par `a` et finissant par `b`
^$	une ligne vide
^.*$	une ligne quelconque, y compris vide
^..*$	une ligne non vide
^$.*$$	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

12.3 Ensembles de caractères

Un et un seul caractère choisi parmi un ensemble de caractères spécifiés entre crochets : `[`*ensemble_de_caractères*`]`

À l'intérieur d'un tel ensemble, les caractères spéciaux sont :

- `-` utilisé pour définir des **intervalles** selon l'ordre lexicographique (dépend des variables de langue)
- `^` en tête pour spécifier le **complémentaire** de l'ensemble
- `]` qui délimite la **fin** de l'ensemble, sauf s'il est placé en première position

À l'intérieur des ces ensembles peuvent figurer des **classes de caractères**

`[:lower:]`, `[:upper:]`, `[:alpha:]`, `[:digit:]`, `[:alnum:]`

Exemples

`[a0+]`	un des trois caractères `a`, `0` ou `+`
`[a-z]`	une lettre minuscule
`[a-z:;?!]`	une lettre minuscule ou une ponctuation double
`[0-9]`	un chiffre
`[^0-9]`	n'importe quel caractère qui n'est pas un chiffre
`[]-]`	un crochet fermant `]` ou un signe moins `-`

Exemples avec une classe :

`[[:digit:]]` au lieu de `[0-9]`

`[-+.[:digit:]]` pour un chiffre, un point ou signe `+` ou `-`

13 Le filtre grep

grep (***g**lobal **r**egular **e**xpression **p**rint*)

affiche les lignes qui contiennent un motif passé en paramètre

syntaxe

```
grep motif [liste_de_fichiers]
```

où ***motif*** est une expression régulière décrivant un motif générique

Principales options :

- **-i** **i**gnore la casse (majuscule/minuscule)
- **-v** in**v**erse la sélection (affiche les lignes sans le motif)
- **-l** affiche la **l**iste des fichiers contenant le motif
- **-n** affiche les lignes contenant le motif précédées de leur **n**uméro
- **-c** (***c**ount*) affiche les noms des fichiers et le nbre de lignes qui contiennent le motif
- **-C** ***n*** affiche n lignes de contexte avant et après

Exemples :

`grep lefrere /etc/passwd`	affiche la ligne de cet utilisateur dans le fichier de mots de passe
`grep '^!' test.f90`	affiche les lignes commençant par `!` dans `test.f90` (commentaires)
`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 fortran
`grep -v '^ *$' test.f90`	affiche les lignes qui ne comportent pas que des blancs
`ls -l | grep ^d`	affiche la liste des sous-répertoires du répertoire courant avec leurs attributs

N.-B. : protéger les caractères spéciaux de l'interprétation par le shell, ici par des «'»

14 Le filtre sed

sed (***s**tream **ed**itor*) : éditeur de flux non interactif

filtre qui analyse ligne par ligne le flux d'entrée et le transforme selon des requêtes suivant une syntaxe similaire à celle de l'éditeur ed.

Deux syntaxes possibles suivant la complexité du traitement :

syntaxe

```
sed -e 'requête_sed' [liste_de_fichiers]
```

Les requêtes comportant des caractères spéciaux sont la plupart du temps protégées par des apostrophes de l'interprétation par le shell.

syntaxe

```
sed -f fichier_de_requêtes.sed [liste_de_fichiers]
```

où *fichier_de_requêtes.sed* contient des lignes de requêtes d'édition.

Autre option : **-n** n'affiche pas les lignes traitées (utiliser la requête p)

La plupart des requêtes sont adressables comme celles de ex.

Exemples : (**s** = ***s**ubstitute*)

`sed -e 's/ab/AB/'`	change **le premier** `ab` de chaque ligne en `AB`
`sed -e 's/10/20/g'`	change **tous** les `10` en `20`
`sed -e '3,$s/0/1/g'`	change **tous** les `0` en `1` à partir de la ligne 3
`sed -e 's/00*/(&)/g'`	entoure de parenthèses **tous** les groupes de `0` **&** représente le motif trouvé
`sed -e 's/[0-9]/(&)/g'`	insère des parenthèses autour de **tous** les chiffres
`sed -e '/motif/s/0/1/g'`	change **tous** les `0` en `1` dans les lignes contenant `motif`
`sed -e '3,$y/01/12/'`	change **caractère par caractère** (comme `tr`) les `0` en `1` et les `1` en `2` à partir de la ligne 3
`sed -e '/^#/d'`	**d**étruit les lignes commençant par #
`sed -n -e '/^#/p'`	affiche (***p**rint*) les lignes commençant par # **-n** pour éviter l'affichage par défaut de tout le fichier

15 Le filtre awk

awk : filtre programmable (agit ligne par ligne comme grep)

fonctionnalités de calcul de type **tableur**, syntaxe proche du langage C

⇒ sensible aux variables de langue (**LC_NUMERIC**) : virgule décimale ou point

Deux syntaxes, sur la ligne de commande ou dans un fichier :

syntaxe

```
awk 'instructions_awk' liste_de_fichiers_de_donnees
```

dans ce cas protéger les instructions de l'interprétation par le shell

syntaxe

```
awk -f fich_de_programme liste_fichiers_donnees
```

Autres Options :

- **-F** *délim* spécifie le séparateur de champs (blancs et tabulations par défaut)
- **-v** *variable=valeur* passe une valeur à une variable awk

15.1 Structure des données pour awk

Pour chaque ligne (*record*), les données sont découpées en champs (*field*) selon le séparateur **FS** (***f**ield **s**eparator*) :

— **$0** la ligne courante

— **NR** (***n**umber of **r**ecord*), son numéro d'enregistrement (de ligne)

— **NF** (***n**umber of **f**ields*), son nombre de champs

— **$1**, **$2**, ... **$NF** : son premier, deuxième, dernier champ

15.2 Structure d'un programme awk

Suite de couples : **sélecteur** **{action}**

⇒ Un **sélecteur** peut être :

— vide et il est vrai pour toutes les lignes

— une expression régulière étendue (ERE) entre **/** et **/**
le sélecteur est vrai si le motif est présent dans la ligne

— une expression logique évaluée pour chaque ligne

— une combinaison logique (via **&&**, **||** ou **!**) de sélecteurs

— un intervalle de lignes sous la forme : sélecteur1, sélecteur2

— **BEGIN** ou **END** qui introduisent des actions exécutées avant ou après la lecture des données

⇒ Une **action** est une suite d'instructions (affectations de variables, calculs, opérations sur des chaînes de caractères, ...) exprimées dans une syntaxe analogue à celle du langage C (structures de contrôle en particulier).

— Constantes chaînes de caractères entre « " ».

— Variables non déclarées et typées seulement lors de leur affectation

— Nombreuses fonctions, notamment numériques (`log`, `cos`, `int`, ...) et chaînes de caractères (`length`, `tolower`, ...) disponibles.

L'action s'applique séquentiellement à toutes les lignes sélectionnées

⇒ **pas de boucle explicite sur les lignes**

N.-B. : awk mal adapté s'il faut plusieurs lectures des données

(ex. : calcul de pourcentage)

15.3 Exemples de programmes awk

— affichage des lignes ayant la valeur **numérique** 2004 pour premier champ

```
$1 == 2004 {print $0}
```
mais protéger du shell si hors fichier

```
awk '$1 == 2004 {print $0}' fichier
```

— affichage des lignes ayant la **chaîne** toto pour deuxième champ

```
awk '$2 == "toto"' fic
```
ne pas oublier les guillemets pour la chaîne

— affichage des lignes avec leur numéro (équivalent de cat -n)

```
awk '{print NR, $0}' fichier
```

— échange des champs 1 et 2 et affichage :

```
awk '{a=$1 ; $1=$2; $2=a; print $0}' fic
```

— affichage du nombre de lignes du fichier (équivalent de wc -l)

```
awk 'END {print NR}' fic
```

— Calcul de la moyenne du champ 1 :

```
BEGIN{ n=0; s=0}                          (initialisation facultative)
{n++ ; s+=$1}                                              (cumul)
END{ print "moyenne = ", s/n}                          (affichage)
```

— Calcul de la moyenne des valeurs supérieures à 10 du champ 1 :

```
BEGIN{ n=0; s=0}                          (initialisation facultative)
$1 > 10 {n++ ; s+=$1}                        (cumul conditionnel)
END{ if (n  > 0 ) {
      print "moyenne = ", s/n                          (affichage)
                  }
      else {
      print "pas de valeurs > 10"
           }
   }
```

15.4 Mise en garde sur les caractères non-imprimables

Les **caractères de contrôle** dans les fichiers texte ne sont **pas toujours visibles** à l'affichage et l'édition.

extrait de man ascii

Oct	Dec	Hex	Car	
010	8	08	BS	'\b' (backspace)
011	9	09	HT	'\t' (horizontal tab)
012	10	0A	LF	'\n' (new line)
013	11	0B	VT	'\v' (vertical tab)
014	12	0C	FF	'\f' (form feed)
015	13	0D	CR	'\r' (carriage ret)

⇒ risques d'erreur avec les filtres.

— confusion entre espaces et **tabulations** : erreur sur motif `grep` ou `sed`

— fin de ligne avec **\r\n** (DOS) : erreur si ajout de caractères en fin de ligne

awk '{print $0 "texte"}' ⇒ **texte** entre **\r** et **\n**

⇒ **texte** écrase le début de ligne

Comment visualiser ces caractères de contrôle ?

avec **cat** : `cat -A` (***A**ll*) ou `cat -vET` affiche les retour chariot (**^M** avec **-v**),

les fins de ligne (**$**, avec **-E**, (***E**nd*)) et les tabulations (**^I** avec **-T**, (***T**ab*)).

sous **vi** : option **:set list** ⇒ **^I** pour tabulation, **$** pour fin de ligne

si `vi fic` affiche **[dos]** sur la ligne d'état, `fic` comporte des **\r**

`vi -b fic` (***b**inary*) affiche **^M** pour les retour chariot

avec **od** : `od -tc` affiche **\t** pour tabulation, **\r** retour chariot, **\n** retour ligne

Rappels : **expand** transforme les tabulations en espaces

tr comprend les séquences d'échappement comme **\t**, **\r**, **\n** et **\b**

16 Gestion des processus

16.1 Généralités : la commande `ps`

Processus = tâche élémentaire identifiée par un numéro unique ou *pid* (***p**rocess **id**entifier*).

Afficher la liste des processus avec la commande **`ps`**

$\Rightarrow$ par défaut ceux de l'utilisateur et associés au même terminal

3 syntaxes pour sélectionner les processus et les informations affichées par **`ps`** :

System V, BSD, et Posix en cours d'implémentation (contrôler avec **`man -a`**).

Principales options :

- **`-e`** (posix **`-A`**) affiche tous les processus de tous les utilisateurs
- **`-U`** *`user_list`* sélectionne les processus appartenant à cette liste d'utilisateurs ou d'UID (séparés par des virgules sans espace)
- **`-f`** (***f**ull*) affiche une liste complète d'informations sur chaque processus
- **`-o`** *`sorties`* (***o**utput*) permet de choisir les informations affichées

Exemples de sélection des processus

```
$ ps
    PID    TTY   TIME CMD
1212592  pts/2   0:00 ps
1294516  pts/2   0:01 bash
```

les processus de l'utilisateur sur le pseudo-terminal courant, affiché par `tty` : `/dev/pts/2`

```
$ ps -U lefrere
    PID    TTY   TIME CMD
 307400      -   0:02 sshd
1212590  pts/2   0:00 ps
1294516  pts/2   0:01 bash
1294620      -   0:02 sshd
1294621  pts/6   0:01 bash
```

les processus de l'utilisateur `lefrere` sur toutes les consoles (ici **2** et **6**) accédant au serveur

N.-B. : la commande **ps** se voit agir.

Format de sortie de ps

Principaux champs affichés :

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

$ ps -f (*full*)

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

En précisant après **-o** la liste (entre virgules sans espace) des champs à afficher et éventuellement leur intitulé selon la syntaxe ***champ*[=*intitulé*]** :

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

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

option **-H** ou option longue (GNU) **--forest**

⇒ affiche la **hiérarchie des processus** (tous fils de `init` de pid **1**)

ps -Af --forest

```
UID   PID  PPID  C STIME TTY          TIME CMD
jal  1935     1  0 03:45 ?        00:00:00 xterm -ls
jal  1936  1935  0 03:45 pts/6    00:00:00  \_ -bash
jal  2114  1936  0 03:53 pts/6    00:00:00      \_ xterm -bg yellow
jal  2115  2114  0 03:53 pts/14   00:00:00      |   \_ bash
jal  2166  2115  0 03:55 pts/14   00:00:00      |       \_ xclock
jal  2167  1936  0 03:55 pts/6    00:00:00      \_ xclock
jal  2188  1936  0 03:57 pts/6    00:00:00      \_ ps -Af --forest
```

Comparer le PPID et le PID du père...

Affichage interactif des processus : commande **top** (**u user** pour sélectionner)

(attention : commande puissante mais consomme des ressources !)

16.2 Caractères de contrôle et signaux

Caractères de contrôle (notés **^X** pour `Ctrl X`) interprétés par le shell

$\Rightarrow$ gestion des processus attachés au terminal et des flux d'entrées/sorties.

^? ou **^H**	`erase`	effacement du dernier caractère
^L	`clear`	efface 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 (fin de session en shell)
^C	`int`	interruption du processus
^	`quit`	interruption forcée du processus avec production d'une image mémoire (fichier `core`)
^Z	`susp`	suspension du processus en cours

`stty` gère l'affectation des caractères de contrôle à certaines fonctions

`stty -a` indique leur affectation courante (ex : `erase=^?; eof=^D`)

Un caractère de contrôle ne peut agir que sur le processus en interaction avec le terminal auquel il est attaché.

16.3 Commandes `kill` et `killall`

Intervenir sur un autre processus (ex. : application graphique qui ne répond plus)

$\Rightarrow$ le désigner par son numéro et lui envoyer un *signal*

`kill` ***`pid`*** où *pid* est le numéro du processus

`kill` envoie par défaut un signal de terminaison = **`kill -s TERM`**

si le processus ne s'interrompt pas, `kill -s` **`KILL`** (ou `kill -s 9`)

`killall` : envoi à une liste de processus désignés par des noms de commandes

exemple : `killall -r '.*mozilla.*'` (**`-r`** comme **`r`**`egexp`)

16.4 Processus en arrière plan

Système UNIX multi-tâche :

— commandes longues non-interactives en arrière-plan (*background*)

— « garder la main » pour d'autres commandes pendant cette tâche de fond (asynchrone)

syntaxe

`commande &`

Gestion des processus en arrière-plan :

— **jobs** affiche la liste des processus en arrière-plan avec leur numéro ($\neq$ `pid`)

— **fg** (***foreground***) passe le job courant en premier plan

fg %`num_job` (passe le job `num_job` en premier plan)

— **bg** (***background***) passe le job courant en arrière-plan

Processus en arrière-plan $\Rightarrow$ plus d'entrées au clavier

$\Rightarrow$ redirections de l'entrée et de la sortie vers des fichiers

mais arrêté par la fermeture du terminal.

Exemples

— **xterm** en premier-plan $\Rightarrow$ on « perd la main » dans la fenêtre initiale.

Dans la nouvelle fenêtre, terminer ce processus par `exit` ou `^D`

$\Rightarrow$ retrouver la main dans la fenêtre initiale.

— **xterm &** $\Rightarrow$ conserve la main dans la fenêtre initiale.

Depuis la fenêtre initiale, terminer ce processus `xterm`

par `kill` *`pid`* ou par `fg` puis `^C`

— si on oublie le **&**, **^Z** pour suspendre le processus, puis **bg** pour le passer en arrière-plan

16.5 Compléments

16.5.1 Processus détaché

Processus détaché = tâche sans interaction avec un terminal,

⇒ continue après la fin de session interactive

⇒ rediriger tous les flux standards

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

Autres modes d'exécution des processus détachés et différés :

— soumission dans une file d'attente ou `batch` (via **qsub** par exemple)
⇒ le système gère les priorités d'exécution

— à un instant déterminé par exemple via le service `cron` :
at *date commande*

— processus *périodiques* pour la gestion du système (sauvegardes, ménage, MAJ des bases de données pour le manuel, ...) : fichier `/etc/crontab`.

— cas des machines qui ne fonctionnent pas en permanence : utiliser **anacron** (`/etc/anacrontab`)

16.5.2 Groupement de commandes

Plusieurs commandes sur une même ligne ⇒ les séparer par un point-virgule **;**

`cd ; pwd ; ls`

Grouper plusieurs commandes grâce aux **parenthèses**

⇒ s'exécutent dans un **sous-shell**, fils du shell interactif.

commande	affichage	remarque
`cd /tmp`		
`(cd /bin ; pwd)`	`/bin`	⇒ dans le sous-shell seulement
`pwd`	`/tmp`	retour au précédent

Groupement et redirections :

`date ; hostname > memo1` contient le nom du calculateur, la date est affichée

`(date ; hostname) > memo2` contient la date et le nom du calculateur

17 Code de retour d'une commande, commande `test`

17.1 Code de retour d'une commande (`$?`)

Toute commande UNIX renvoie en fin d'exécution un code entier : valeur de retour (cf. `exit()` dans `main` en C) ou statut de fin (*return status*) accessible via **`$?`**

Code de sortie = **0** $\Longleftrightarrow$ la commande s'est bien déroulée.

`cd /bin`

`echo $?` affiche **0**

`cd /introuvable` affiche un message d'erreur

`echo $?` affiche **1**

Certains codes de retour non nuls ont une signification particulière (par ex. 127 pour commande introuvable).

17.2 Inversion du statut de retour (!)

syntaxe

! *commande*	un espace après le !

17.3 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.

Par exemple, on lance un exécutable seulement si sa production (compilation et lien) s'est effectuée sans erreur.

```
gfortran source.f90 && a.out
```

17.4 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.

Exemple : gestion minimale des erreurs

```
cp fichier1 fichier2 || echo la copie a échoué
```

18 La commande `test`

`test` permet de comparer des nombres entiers ou des chaînes de caractères et de tester les propriétés de fichiers.

commande invocable sous deux formes :

- **`test`** *`expression`*
- **`[`** *`expression`* **`]`** en plaçant simplement la condition entre crochets

Les arguments de `test` **doivent être séparés par un espace**

un espace doit séparer la condition à tester des délimiteurs `[` et `]`.

`test` fournit un code de retour `$?` (`0` si vrai / `1` si faux)

$\Rightarrow$ exploitation à l'aide de structures de contrôle du shell de type

`if`, `while` ou `until`.

Commandes à code de retour fixe

`true` $\Rightarrow$ code de retour fixe = 0

`false` $\Rightarrow$ code de retour fixe = 1

18.1 Comparaisons arithmétiques

-le	$\leqslant$	***l**ower or **e**qual*	inférieur ou égal à
-lt	$<$	***l**ower **t**han*	inférieur à
-eq	$=$	***eq**ual*	égal à
-ge	$\geqslant$	***g**reater or **e**qual*	supérieur ou égal à
-gt	$>$	***g**reater **t**han*	supérieur à
-ne	$\neq$	***n**ot **e**qual*	différent de

Exemples

```
test 3 -gt 2
echo $?
--> 0
```

```
test 03 -eq 3
echo $?
--> 0
```

```
test 3 -lt 2
echo $?
--> 1
```

18.2 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
```

chaînes $\neq$ nombres

```
[ 03 = 3 ]
echo $?
--> 1
```

```
var1=aaa
var2=aaa
[ ${var1} != ${var2} ]
echo $?
--> 1
```

18.3 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

Exemple

```
test -r /etc/motd
echo $?
--> 0
test -w /etc/motd
echo $?
--> 1
```

```
test -d /tmp
echo $?
--> 0
test -f /tmp
echo $?
--> 1
```

18.4 Combinaisons de conditions

Parenthèses **( )** pour grouper des conditions,

Mais protéger les parenthèses par **** $\Rightarrow$ pas interprétées par le shell

!	**-o**	**-a**
négation	ou logique (*or*)	et logique (*and*)

Exemple `test \( ${a} -ge 2 \) -a \( ${a} -le 5 \)`

est vrai si $2 \leqslant a \leqslant 5$.

19 Variables shell

Variables de l'interpréteur de commandes :

— non déclarées

— non typées a priori ⇒ **chaînes de caractères**

19.1 Affectation et référence

— Syntaxe d' affectation (en shell de type BOURNE) :

syntaxe

variable*=*valeur (sans espace autour du signe =)

— Référence à la valeur de la variable :

syntaxe

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

La commande interne **set** (sans argument)

affiche la liste des variables et leurs valeurs.

Exemples

```
alpha=toto ; b=35 ; c2=3b
echo alpha, b, c2 contiennent ${alpha}, ${b}, ${c2}
```

⇒ **alpha, b, c2 contiennent toto, 35, 3b**

```
set | grep alpha
```

⇒ **alpha=toto**

```
bb=${b}+${b} ; echo b vaut ${b}, bb vaut ${bb}
```

⇒ **b vaut 35, bb vaut 35+35**

pas d'arithmétique directement avec le shell ⇒ utiliser **expr** ou **$(())**

```
expr 3 + 12
echo $((3+12))
```

affichent **15**

19.2 Saisie interactive d'une liste de variables

commande interne **read** : lecture au clavier

syntaxe

read liste_de_variables
liste_de_valeurs saisies sur une seule ligne

```
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
```

Un mot par variable sauf éventuellement la dernière qui stocke la fin de la ligne

19.3 Portée des variables ordinaires du shell

Par défaut, **pas d'héritage** des variables normales par les processus fils.

```
a=bonjour                                   affectation dans le processus père
echo a contient +${a}+
⇒ a contient +bonjour+
bash ____________________________________   lancement d'un shell fils
    echo a contient +${a}+                  variable locale
    ⇒ a contient ++                         pas affecté
    a=salut                                 affectation dans le fils
    echo a contient +${a}+
    ⇒ a contient +salut+
    exit ____________________________       retour au shell père
echo a contient +${a}+
⇒ a contient +bonjour+                      valeur avant appel du shell fils
```

19.4 Extension de la portée d'une variable : variables d'environnement

Exportation d'une variable vers les processus fils (shell de type Bourne) :

syntaxe

```
export variable
```

En fait copie locale, donc pas d'héritage inverse, du processus fils vers le père.

Variables d'environnement systématiquement **héritées par les processus fils**.

Liste des variables d'environnement et de leur valeur : **env**

Variables d'environnement standard :

- **SHELL** : interpréteur de commandes utilisé (`bash`, `ksh`, `tcsh`, ...)
- **TERM** : type de terminal utilisé (`vt100`, `xterm`, ...)
- **HOME** : répertoire d'accueil

— **USER** : identifiant (nom) de l'utilisateur

— **PATH** : liste des chemins de recherche des commandes séparés par des « **:** »

Quand on lance une commande ou un exécutable :

— **avec /** dans le nom, on précise le chemin d'accès explicitement :

par ex. `./a.out`

— **sans /** dans le nom, la recherche se fait dans tous les répertoires listés dans `PATH` en respectant l'ordre, par ex. `a.out`

Par exemple, `echo $PATH` montre que le répertoire courant n'est pas scruté.

/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/lefrere/bin

Si on l'ajoute à la fin, il est scruté en dernier :

PATH="${PATH}:." ; echo ${PATH} donne

/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/lefrere/bin:.

Ne pas placer le point au début du `PATH` pour des raisons de sécurité !

Attention : `PATH=""` $\Rightarrow$ seules les commandes avec chemin sont trouvées

$+$ mémoire (cache) des chemins des commandes utilisées gérée par **hash**

19.5 Variables de localisation (langue, ...)

- **LANG**,
- **LC_ALL** qui résume les suivantes
- **LC_CTYPE** détermine la classification des caractères (`[:lower:]` par ex.)
- **LC_NUMERIC** (détermine le séparateur décimal : à positionner correctement pour `awk`),
- **LC_COLLATE** (qui influe sur l'ordre lexicographique : important pour le classement avec `sort` et les expressions régulières avec des intervalles)
- **LC_TIME** pour la date et l'heure
- **LC_PAPER** `A4` en Europe ou `Letter` aux États-Unis
- ...

Influent sur de nombreuses commandes (`man`, `wc`, `awk`, `sort`, `ls`, ...)

Affichées par la commande **locale**.

2 minuscules (langue) + _ + 2 majuscules (variante locale) + . + nom du codage

Exemples : **C** (norme `POSIX`), **fr_FR.ISO-8859-1** ou **fr_CA.UTF-8**

19.6 Complément : valeur par défaut d'une variable

Substitution avancée de variables : **${*variable*-*valeur*}** donne

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

— ***valeur*** sinon

Exemple : `${TMPDIR-/tmp}`

Variante `${`*`variable`*`:-`*`valeur`*`}` qui rend la valeur par défaut aussi lorsque la variable est définie mais vide.

Nombreuses autres constructions de ce type.

20 Caractères interprétés par le shell

20.1 Substitution de commande

Résultat d'une commande (sa sortie standard) → chaîne de caractères
stocké dans une variable ou repris comme argument d'une autre commande.

syntaxe

```
$(commande)
```

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

— paramétrage de shell-scripts,

— calculs sur les entiers avec la commande **`expr`**,

— manipulation de noms de fichiers et de chemins avec les commandes
`basename` et **`dirname`**

Exemples

```
qui=$(whoami) ; echo ${qui}
```
affectation de la variable puis affichage

```
echo je suis $(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}
```
⇒ ***la somme de 12 et 2 est 14***

```
echo 12+2+1 vaut $(expr $(expr 12 + 2) + 1)
```
imbrication

⇒ ***12+2+1 vaut 15***

```
#
sans_suffixe=$(basename source.for .for)
```
suppression du suffixe

```
source90=$(basename source.for .for).f90
```
changement du suffixe

```
echo ${sans_suffixe} ${source90}
```
⇒ **source source.f90**

20.2 Métacaractères du shell

␣ ou TAB	espace ou tabulation : séparateur (IFS)
`*`, `?`, `[...]`	constructeurs de noms de fichiers
`~`	répertoire d'accueil
`<`, `<<`, `>`, `>>`, `|`	redirections
`$` ou `${...}`	évaluation de variable
`$(...)` ou `` `...` ``	substitution de commande
`;`	séparation de commandes
`(...)` `{...}`	groupement de commandes
`||` et `&&`	associations de commandes
`!`	non logique
`&`	lancement en arrière plan
`#`	introducteur de commentaire

Les caractères de protection

Toujours deux étapes d'interprétation : le shell, puis la commande

1. En premier lieu, le **shell** interprète la ligne de commande
   (espaces, caractères jokers, redirections, variables, ...)
2. Puis, la **commande** interprète certains caractères spéciaux pour elle.
   (expressions régulières pour `grep`, `sed`, ...)

Généralement, ne pas exposer ces caractères à l'interprétation par le shell :

$\Rightarrow$ les protéger.

Protections :

\	**protection individuelle** du caractère suivant (*backslash*)
'...'	**protection forte** (*quote*) : aucune interprétation
"..."	**protection faible** (*double quote*) : substitution de paramètres ou de commandes (**$** interprété)

Exemples

— **`grep '^[0-9][0-9]*' fic`**

affiche les lignes de `fic` commençant par un chiffre

— Affectation d'une chaîne comportant des blancs à une variable ou à un paramètre de commande : (nom de fichier avec des blancs)

```
v1="avec blanc1" ; v2='avec blanc2' ; v3=avec\ blanc3
echo ${v1}, ${v2}, ${v3}
```

⇒ **`avec blanc1, avec blanc2, avec blanc3`**

— **`echo ${TERM} \${TERM} "${TERM}" '${TERM}'`**

⇒ **`xterm ${TERM} xterm ${TERM}`**

— **`find ~lefrere -name '*.f90' -print`**

Sans protection, le shell remplace **`*.f90`** par la liste des fichiers de suffixe `.f90` dans le *répertoire courant* avant l'exécution de `find`.

S'il y en a plus d'un, erreur de syntaxe sur `find` qui attend **un** seul paramètre après `name`.

20.3 La commande `expr`

La commande **`expr`** évalue des expressions contenant des opérateurs arithmétiques sur les entiers ou logiques ou portant sur des chaînes de caractères.

Chaque terme de l'expression à évaluer doit être séparé du suivant par un espace.

Protéger (par des **`\`**) les autres métacaractères de l'interprétation par le shell.

Utilisée pour affecter des valeurs à des variables
via une substitution de commande du type
`var=$(expr ${var1} + ${var2})`

Dans certains shells, syntaxes plus concises, du type **`c=$((${a}+${b}))`**
$\Rightarrow$ arithmétique plus rapide grâce aux commandes intégrées au shell

20.3.1 Opérateurs arithmétiques sur les entiers

`+`	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                                                protéger * et ()
var3=$(expr 10 \* \( ${var1} + ${var2} \))
echo ${var3}
--> 50
```

20.4 Autres opérateurs

La commande `expr` permet aussi des comparaisons grâce aux opérateurs binaires **<**, **<=**, **=**, **!=**, **>**, **>=**,
qui utilisent des métacaractères ⇒ à protéger d'une évaluation par le shell.

Résultat de la comparaison : (ne pas confondre avec `test`)

1 si la condition est réalisée

0 si elle n'est pas réalisée.

Exemples

Ne pas confondre valeur affichée et statut de retour

```
expr 2 \> 1
--> 1
echo $?
--> 0
```

```
expr 2 \< 1
--> 0
echo $?
--> 1
```

21 Shell-scripts

21.1 Fichiers de commandes ou shell-scripts

Fichier texte contenant des commandes, créé avec un éditeur de textes

Trois méthodes d'exécution :

1. **`bash` *`fichier_de_cmdes`*** éviter
2. puis rendre le script **exécutable** par **`chmod +x` *`fichier_de_cmdes`***
   et donner le chemin d'accès du fichier de commandes
   `./fichier_de_cmdes` faire suivre des paramètres éventuels
3. Ajouter dans **`PATH`** le chemin d'accès au fichier. Saisir
   `fichier_de_cmdes`
   qui est alors une commande recherchée dans l'ordre du `PATH`
   $\Rightarrow$ éviter les noms des commandes existantes sur le système

En pratique, répertoire courant (**`.`**) à la fin du `PATH`,
mais mieux : scripts dans **`${HOME}/bin/`** et `${HOME}/bin/` dans le `PATH`

Exemple de shell-script sans paramètre

```
#!/bin/sh
#
# shell script sans paramètre
echo nous sommes le ; date
echo mon login est $(whoami)
echo "le calculateur est $(hostname)"
```

introduit les commentaires... sauf

sur la première ligne commençant par **#!**

$\Rightarrow$ précise le shell d'interprétation du script.

$\Rightarrow$ assure le portabilité du script

21.2 Les paramètres des scripts

Variables positionnées dans la procédure lors du lancement :

$0	nom du fichier de commande (tel que spécifié lors de l'appel)
$1, **$2**, ... **$9**	*paramètres positionnels* (arguments) avec lesquels la procédure a été appelée
	le nombre de paramètres peut dépasser 9
	$\Rightarrow$ accéder au dixième paramètre via `${10}`
$*	chaîne formée par l'ensemble des paramètres d'appel **"**`$1 $2 $3 ...`**"**
$#	nombre de paramètres positionnels lors de l'appel
$$	numéro du processus lancé (pid)

Exemples de procédure avec des paramètres

```
#!/bin/sh
# fichier proc0.sh
echo la procédure $0
echo a été appelée avec $# paramètres
echo le premier paramètre est $1
echo la liste des paramètres est $*
echo le numéro du processus lancé est $$
```

```
#!/bin/sh
# fichier concat.sh
# permet de concatener (cf "cat") deux fichiers ($1 et $2)
# dans le fichier $3
# en habillant le résultat avec le nom
# des fichiers initiaux en entete
#
ficin1=$1
ficin2=$2
ficout=$3
echo commande $0 avec les $# parametres $*
echo  et le numero de processus $$
echo "début de la concatenation de\
 $ficin1 et $ficin2 sur $ficout"
echo '-------------------------------' > $ficout
```

```
echo \| $ficin1 \| >> $ficout
echo '----------------------------------------' >> $ficout
cat  $ficin1 >> $ficout
echo '----------------------------------------' >> $ficout
echo \| $ficin2 \| >> $ficout
echo '----------------------------------------' >> $ficout
cat $ficin2 >> $ficout
echo termine
exit 0
# fin du fichier concat.sh
```

21.3 La commande interne `shift`

`shift` permet de décaler d'une position les paramètres positionnels de la procédure : utiliser après usage du premier (`$1`) car il est perdu par `shift`

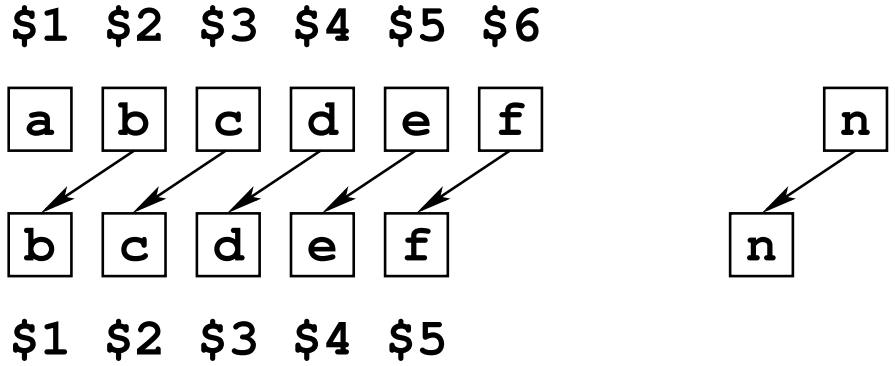


FIGURE 20 – Décalage des paramètres positionnels par `shift`

`shift` suivie d'un entier p décale de p positions les paramètres.

`shift` utilisée dans une boucle sur les paramètres positionnels d'une procédure pour se débarrasser des paramètres après leur prise en compte.

```
#!/bin/sh
# fichier shift1.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
```

`shift1.sh aa b c ddd ee ff` donne

```
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
```

21.4 Distinction entre $* et $@

$@** est équivalente à **$*, 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** soit `"$1" "$2" "$3" ....`

21.5 Compléments sur la commande `set`

`set` est une commande interne du shell aux usages multiples :

— `set` sans paramètre **affiche** la liste des variables et leur valeur

— `set` suivie d'une liste de mots **affecte chaque mot** à un paramètre positionnel ($1, $2, ...)

```
set aaa bbb "ccc d"
echo $# ⇒ 3
echo $1 ⇒ aaa
echo $2 ⇒ bbb
echo $3 ⇒ ccc d
```

— **`set --`** pour affecter au premier paramètre une valeur commençant par **`-`**

```
set -- $(ls -al .profile)
echo le fichier $9 comporte $5 octets
```

— **`set -o`** affiche les options en cours du shell

`set -o emacs` ⇒ édition de la ligne de commande en mode `emacs` au lieu de `vi`

— **`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 chaque commande (sans évaluation) avant de l'exécuter
- **`set -x`** (***x****trace*) affiche chaque commande (précédée du signe **`+`**) après évaluation des substitutions de commandes, `$(...)` et de variables, `${...}` avant de l'exécuter

Plusieurs possibilités lors de la mise au point :

- Placer la commande `set -vx` en tête du shell-script
- Ajouter ces options à la ligne `#!/bin/sh` ⇒ `#!/bin/sh -vx`
- Lancer le script avec `sh -vx` *`shell-script`*

22 Structures de contrôle en shell (sh)

22.1 Introduction

le shell = interpréteur de commandes

= langage de programmation

⇒ variables, paramètres des procédures

structures de contrôle

Pas de typage des variables ⇒ condition = code de retour des commandes

Mais souvent, code de retour de la commande **test**

Mots clefs réservés du shell : **if**, **then**, **else**, **fi**, **elif**, **for**, **do**, ...

Remarque : syntaxe différente en csh

⇒ préciser le shell dans les scripts par **#!/bin/sh** pour assurer la portabilité

22.2 Conditions

22.2.1 Structure `if ... fi`

syntaxe

```
if cmd
then
    commande(s)        (exécutées si le code de retour de cmd est 0)
else
    commande(s)        (exécutées si le code de retour de cmd est ≠ 0)
fi
```

Partie **`else`** optionnelle.

Exemple avec la commande test

```
#!/bin/sh
if test $# -eq 0
   then
   echo commande lancée sans paramètre
else
   echo commande lancée avec au moins un paramètre
fi
```

Exemple avec un tube

```
#!/bin/sh
# indique si l'utilisateur de nom $1 est connecté
if who | grep "^$1 "          code de retour = celui de grep
   then                  (grep rend 0 si le motif est trouvé)
   echo $1 est connecté
fi
```

22.2.2 Structures `if` imbriquées : `elif`

Remplacer **`else if`** par **`elif`** $\Rightarrow$ un seul **`fi`** (plus d'imbrication)

syntaxe

```
if ...                          if ...
   then                            then
      ...                             ...
else if ...                     elif ...
        then          ⇒                 then
             ...                        ...
     else                       else
             ...                        ...
     fi
fi                              fi          (un seul)
```

Exemple de `elif`

```
#!/bin/sh
if test $# -eq 0
then
   echo Relancer la cmde en ajoutant un paramètre
elif who | grep "^$1 " > /dev/null # sans affichage
then
   echo $1 est connecté
else
   echo $1 n\'est pas connecté
fi
```

22.2.3 Énumération de motifs (cas) : `case ... esac`

syntaxe

```
case variable in
   motif1) commande(s)
                        ;;
   motif2) commande(s)
                        ;;
   ...
esac
```

La valeur de la variable est comparée avec les motifs successifs : à la première coïncidence, les commandes associées au motif sont exécutées jusqu'au `;;`, qui provoque la fin de l'exploration.

Syntaxe des motifs :

- `*` = 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 `|`

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
```

Remarques

Les motifs peuvent se recouvrir, mais **seule** la première coïncidence provoque l'exécution de commandes

$\Rightarrow$ l'**ordre des motifs** est important.

En C-shell ou langage C, structure **`switch`** mais où chacun des motifs en coïncidence provoque l'exécution de commandes.

Structure **`switch`** équivalente à **`case`**

...**si** chaque cas est terminé par **`breaksw`** / **`break;`**

22.3 Les structures itératives

22.3.1 La structure `for ... do ... done`

syntaxe

```
for variable [in liste de mots]
do
    commande(s)
done
```

Liste des mots par défaut : les paramètres du script

`"$@"` (`"$1" "$2" "$3"` ...)

Exemple avec liste explicite

```
#! /bin/sh
for mot in 1 5 10 2 "la fin"
do
    echo mot vaut ${mot}
done
```

$\Rightarrow$ boucle avec 5 passages

Exemple avec liste implicite

soit le script `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+
```

Liste générée par le joker *

```
#!/bin/sh
for fichier in *.f90
do
    echo fichier ${fichier}
done
```

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
```

22.3.2 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
```

22.3.3 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
```

22.4 Compléments : branchements

22.4.1 La commande `exit`

exit [*statut_de_fin*] arrête l'exécution de la procédure
et rend le *statut_de_fin* (0 par défaut) à l'appelant.

Utilisé pour arrêter le traitement en cas d'erreur après envoi d'un message
$\Rightarrow$ rendre alors un code $\neq 0$

```
...
if [ $# -lt 1 ]       # test sur le nb d'arguments
  then
    echo "il manque les arguments" >&2
    # message sur sortie d'erreur
    exit 1    # sortie avec code d'erreur
fi
...
```

22.4.2 La commande `break`

`break` $\Rightarrow$ sortie d'une boucle avant la fin ;

`break` n sort des n boucles les plus intérieures.

Nécessaire dans les boucles a priori infinies (`while true`, `until false`)
insérée dans un bloc conditionnel pour arrêter la boucle

```
#!/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
```

22.4.3 La commande `continue`

`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.

insérée dans un bloc conditionnel pour court-circuiter la fin de boucle

```
#!/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
```

22.4.4 La commande `trap`

`trap` intercepte les signaux envoyés à un processus en cours d'exécution et exécute des commandes **au lieu** des actions habituellement requises par ces signaux.

syntaxe

```
trap "action" liste de signaux à intercepter
```

action vide ⇒ signaux interceptés sans effet sur le processus

action=**"-"** ⇒ restitue l'action normale des signaux

Exemple Suppression des fichiers temporaires avant l'arrêt d'un processus par l'un des signaux `INT KILL TERM`.

```
trap "/bin/rm -f /tmp/fichier_tmp; exit 1" INT KILL TERM
```

```
1  #! /bin/sh
2  # fichier trap.sh
3  for f in *
4  do
5    echo début de boucle $f
6    trap "echo fin provoquée par signal; \
7          exit 3" INT QUIT
8    echo "fin de boucle $f"
9  done
10 echo "fin de script"
11 exit 0
```

NB : l'ensemble des commandes déclenchées par `trap` (lignes 6 et 7) est délimité par des guillemets `"`. Les deux commandes sont séparées par un point virgule `;`
Il faut par ailleurs protéger le changement de ligne dans la chaîne par `\`.

22.4.5 Redirections et boucles

redirection (d'entrée ou de sortie) après `done`

$\Rightarrow$ s'applique à la structure itérative

Exemple

```
#!/bin/sh
# redirection et structure itérative
# version à conseiller
for i in 1 2 3
do
  echo $i
done > resultat # redirection après done
exit 0
```

Éviter la méthode suivante :

```
#!/bin/sh
# redirection et structure itérative
# version à déconseiller
# partir d'un fichier vide
cat /dev/null > resultat
for i in 1 2 3
do
  echo $i >> resultat # accumuler dans la boucle
done
exit 0
```

23 Exemple commenté d'un script

23.1 Introduction

Comment passer tous les noms des fichiers d'un répertoire en minuscules ?

(le chemin du répertoire sera passé en argument de la commande)

Principales commandes utilisées :

— Changer le nom d'un fichier

```
mv FIC1.F90 fic1.f90
```

— Passer en minuscules (pas de signes diacritiques en UTF8)

```
tr '[:upper:]' '[:lower:]'
```

— Faire une boucle sur tous les fichiers du répertoire

```
for f in *
do
    ...
done
```

23.2 Le cœur de script

```
for NOM in *
do
    passer NOM en minuscules avec tr ⇒ nom
    mv ${NOM} ${nom}
done
```

Mais **tr** est un filtre qui transforme l'entrée standard,

donc il faut afficher le nom initial sur la sortie standard par **echo**.

```
echo ${NOM} | tr '[:upper:]' '[:lower:]'
```

Puis récupérer la sortie standard de ce tube

dans une variable **nom** grâce à la syntaxe **$()** .

```
nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
```

23.3 Version minimale du script

```
#!/bin/sh
# fichier min-noms-0.sh
# passage en minuscules des noms des fichiers d'un répertoire
# version minimale
cd $1
for NOM in *
do
  # traduction du nom en minuscules
  nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
  # changement effectif du nom de fichier
  mv ${NOM} ${nom} && echo ${NOM} '=>' ${nom}
done
echo "fin"
exit
```

Problèmes :

— on peut écraser par exemple **`fic1`** en traitant **`FiC1`**

⇒ ne pas renommer dans ce cas ... sauf si `fic1` est vide.

⇒ vérifier si `mv` risque d'écrire sur un autre fichier déjà présent et non vide.

```
if [ "${nom}" != "${NOM}" ]
then # les noms diffèrent effectivement
    echo on va essayer de changer ${NOM} en ${nom}
    if [ -s "${nom}" ]
    then  # risque d'écraser un fichier non vide
        echo ...
    else
        mv ${NOM} ${nom}
    fi
fi
```

— que faire s'il ne s'agit pas d'un fichier ordinaire ?

⇒ ne pas renommer dans ce cas (répertoire par exemple)

23.4 Version élémentaire du script

```
#!/bin/sh
# fichier min-noms-1.sh
# passage en minuscules des noms des fichiers d'un répertoire
# version élémentaire
cd $1
for NOM in *
do
  if [ -f "${NOM}" ]
  then
    # le fichier $NOM existe et est un fichier ordinaire
    # traduction du nom en minuscules
    nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
    if [ "${nom}" != "${NOM}" ]
    then
```

```
      # les noms diffèrent effectivement
      echo "peut-on changer ${NOM} en ${nom} ?"
      if [ -s "${nom}" ]
      then # risque d'écraser un fichier non vide
        echo ${NOM} devrait écraser ${nom} '=>' non traité >&2
      else # changement effectif du nom de fichier
        mv ${NOM} ${nom} && echo ${NOM} '=>' ${nom}
      fi
    fi
  else
    # le fichier n'existe pas ou n'est pas un fichier ordinaire
    echo "${NOM} n'est pas un fichier ordinaire => non traité" >&2
  fi
done
echo "fin"
exit
```

Problème plus grave : que se passe-t-il si on ne passe pas d'argument ?

`cd $1` $\Rightarrow$ `cd` et on modifie les noms dans **le répertoire d'accueil** !

$\Rightarrow$ Vérifier s'il y a un argument,

sinon on peut **choisir de** travailler dans le répertoire courant.

S'il y a un argument, s'assurer que la commande `cd $1` réussit,

ou plus précisément, arrêter le script si elle échoue.

```
if ! cd ${repertoire}
then
   echo "$repertoire inaccessible" >&2
   exit 2
fi
```

Ajouter quelques éléments de contrôle avec une liste avant et après les renommages, mais sans déposer de fichier temporaire dans le répertoire de travail.

23.5 Version plus robuste du script

```
#!/bin/sh
# fichier min-noms.sh
# passage en minuscules des noms des fichiers d'un répertoire
# test sur le nombre d'arguments
case $# in
  0) repertoire="."
  ;;
  1) repertoire=$1
  ;;
  *) echo erreur nombre d\'arguments
     echo "usage: $0 [répertoire]" >&2
     exit 1
  ;;
esac
```

```
if [ ! -d "${repertoire}" ]
then
  echo "${repertoire} n'est pas un répertoire => abandon" >&2
  exit 2
fi
if ! cd ${repertoire}
then
  echo "répertoire inaccessible => abandon" >&2
  exit 2
fi
echo "passage en minuscules du nom des fichiers de $(pwd)"
echo 'Confirmez svp par O/N [N]'
OK=O
read reponse
if [ "${reponse}" != "${OK}" ]
then
```

```
  echo abandon demandé
  exit 0
fi
TEMPFILE="/tmp/$(whoami).$$" # fichier temporaire de nom unique
ls -l > ${TEMPFILE}          # liste avant modifications
for NOM in *
do
  if [ -f "${NOM}" ]
  then
    # le fichier $NOM existe et est un fichier ordinaire
    # traduction du nom en minuscules
    nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
    if [ "${nom}" != "${NOM}" ]
    then # les noms diffèrent effectivement
      echo "peut-on changer ${NOM} en ${nom} ?"
      if [ -s "${nom}" ]
```

```
      then # risque d'écraser un fichier non vide
        echo ${NOM} devrait écraser ${nom} '=>' non traité >&2
      else # changement effectif du nom de fichier
        mv ${NOM} ${nom} && echo ${NOM} '=>' ${nom}
      fi
    fi
  else
    # le fichier n'existe pas ou n'est pas un fichier ordinaire
    echo "${NOM} n'est pas un fichier ordinaire => non traité" >&2
  fi
done
ls -l > ${TEMPFILE}+                    # liste après modifications
echo "Bilan"
diff ${TEMPFILE} ${TEMPFILE}+           # comparaison des listes
/bin/rm ${TEMPFILE} ${TEMPFILE}+ # ménage
exit 0
```

23.6 Limitations

— Fichiers cachés (commençant par « `.` ») non traités

⇒ remplacer `for NOM in *` par

`for NOM in $(ls -a)` (`.` et `..` éliminés car répertoires)

— Pas d'action en cas de collision de noms,

mais on pourrait demander de saisir un autre nom

— On peut interrompre le déroulement en cours de boucle

⇒ insérer `trap '...' INT` pour nettoyer les fichiers temporaires

— Cas des noms comportant des caractères spéciaux non étudié

— **`tr`** ne traite que les caractères sur un octet,

⇒ aucun caractère accentué en UTF-8 dans les noms n'est traité

24 Compléments sur le shell

24.1 Commandes internes

Certaines commandes intégrées au shell (*builtin*) ⇒ plus rapides, ne lancent pas un nouveau processus, permettent d'affecter le shell courant...

`cd`, `echo`, `pwd`, `read`, `set`, ...

`eval`, `exec`, `getopts`, ...

Pas de `man`, sauf celui du shell, mais **`help`** *`cmde_interne`*

24.2 Exécution dans le shell courant

⇒ hériter des variables

en sh, bash ou ksh

`.␣`*`commande`*

Exemple : `.␣.profile`

24.3 Autres commandes internes

24.3.1 La commande `eval`

Dans certaines circonstances, nécessité de faire agir le shell 2 fois sur la ligne de commande ⇒ double interprétation par le shell.

Cas le plus classique : accès au contenu du contenu d'une variable :

`${${`*`variable`***`}}`** ⇒ utiliser **`eval`**

`eval` `valeur=\${${`*`variable`*`}}`

protéger le premier `$` de la première interprétation par le shell,

sinon erreur de syntaxe

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
```

24.3.2 La commande `exec`

`exec` ***`commande`*** vient remplacer le processus courant par celui de *`commande`*

Si `exec` *`commande`* est lancé en interactif, il y a fermeture du shell, donc de la session à la fin de la commande. ⇒ `exec csh` pour passer en `csh`.

`exec >` *`fichier`* (sans commande) en début de script
⇒ redirection de sortie pendant tout le script.

24.3.3 La commande `getopts`

`getopts` simplifie le découpage (*parsing*)
des paramètres optionnels passés à un script

24.4 Divers

24.4.1 Fonctions en shell

Fonctions prédéfinies et possibilité d'en définir $\Rightarrow$ scripts plus modulaires

```
gfortran2003-mni ()
{
    gfortran -Wextra -Wall -fimplicit-none -Wunused  \
     -ffloat-store -fbounds-check -Wimplicit-interface \
     -fexceptions -pedantic -fautomatic -std=f2003  $* ;
}
```

24.4.2 Alias du shell

Notion d'alias scrutés **avant** les commandes

Choix d'options des commandes existantes, raccourcis pour des commandes,...

`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

`\ls` permet de retrouver la commande `ls` native.

24.4.3 Identifier une commande `type`

`type` permet de savoir comment est interprété un identificateur,

avec l'ordre de priorité suivant :

alias, mot-clef, fonction, commande interne, shell-script ou exécutable

`type ls` affiche **`ls est un alias vers « ls -F »`**

24.4.4 Affichage d'une progression arithmétique `seq`

`seq` [*premier* [*incrément*]] *dernier*

`seq` affiche la progression arithmétique depuis `premier` jusqu'à `dernier` par pas de `incrément` ; les paramètres optionnels = 1 par défaut.

Séparateur par défaut = retour ligne sauf option **`-s`** :

`seq -s' ' 10 2 15` affiche **`10 12 14`**

`seq -s'/' 2 5` affiche **`2/3/4/5`**

`seq -s'-' 5` affiche **`1-2-3-4-5`**

24.4.5 Récursivité

Un script peut s'appeler lui-même, tant que le nombre de processus lancés ne dépasse pas la limite fixée par l'administrateur (voir `ulimit`).

Méthode récursive élégante, mais souvent peu performante.

24.4.6 Fichiers d'initialisation du shell

En `ksh` et `bash` (différent avec `csh` et `tcsh`)

— **`/etc/profile`** pour tous au login

— **`${HOME}/.profile`** ou **`${HOME}/.bash_profile`** personnels au login

— éventuellement le fichier défini par la variable `ENV`
`${HOME}/.kshrc` ou **`${HOME}/.bashrc`**

25 Autres outils

25.1 Automatisation des tâches avec la commande `make`

Outil de gestion des dépendances entre des fichiers sur la base de leur date de modification et de règles de dépendances.
Application la plus classique : reconstituer un programme exécutable à partir des fichiers sources en ne recompilant que ceux qui ont été modifiés.

— cible (*target*) : fichier à produire

— règle de production (*rule*) : liste des commandes à exécuter pour produire une cible (compilation pour les fichiers objets, édition de lien pour l'exécutable)

— dépendances : ensemble des fichiers nécessaires à la production d'une cible

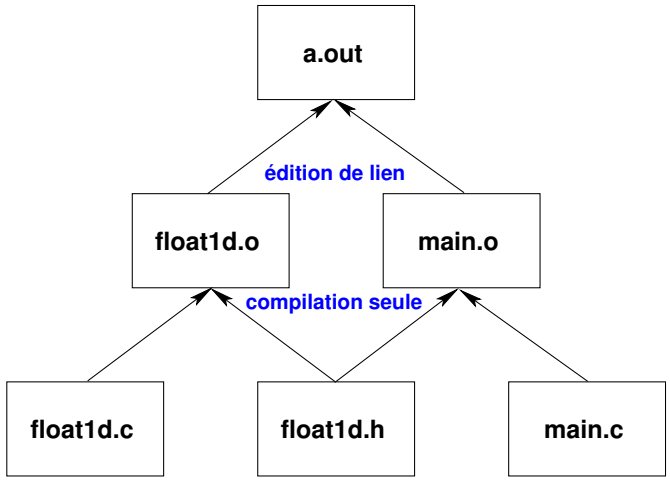
Le fichier **`makefile`** liste les cibles, décrit les dépendances et les règles.
Il est construit à partir de l'arbre des dépendances.

`make` *`cible`*

lance la production de la *`cible`* en exploitant le fichier `makefile`

Exemple élémentaire de `makefile` pour du C

Arbre des dépendances




```
# première cible = celle par défaut
a.out : float1d.o  main.o
        gcc float1d.o  main.o
# cibles des objets
float1d.o : float1d.c float1d.h
        gcc -c float1d.c
main.o : main.c float1d.h
        gcc -c main.c
# suppression des fichiers reconstructibles
clean:
        /bin/rm -f a.out *.o
```

N.-B : une **tabulation** en début de ligne pour les règles de production

25.2 L'outil `perl`

PERL = *Practical Extraction and Report Language*

— sous un même outil :

  – manipulation des chaînes de caractères avec expressions rationnelles (comme avec `sed`),

  – listes, tableaux et tableaux associatifs,

  – structures de contrôle (comme en [C]-shell),

  – opérateurs arithmétiques et logiques (comme sous `awk`),

  – fonctions, ...

— bibliothèques très riches (CPAN : Comprehensive Perl Archive Network)

— outil du domaine public porté sur plusieurs systèmes d'exploitation

`http://www.perl.org/`