Asli Grimaud
Gilles Grimaud

Informatique MP2I

Cours complets, exercices corrigés et projets guidés pour les classes préparatoires

RESSOURCES TÉLÉCHARGEABLES

ellipses

Chapitre 1
Gestion des fichiers

Le système d'exploitation est un logiciel central qui gère le matériel d'un ordinateur et les autres programmes. Il orchestre la mémoire et le processeur, organise les disques et gère l'accès aux périphériques. Il garantit la sécurité des applications, empêchant qu'une défaillance de l'une affecte le fonctionnement des autres. Il fournit aussi des abstractions simples du matériel, facilitant ainsi son exploitation par les programmes. Sur ses fondations, des logiciels de base comme les éditeurs de texte, les outils de développement et les navigateurs web sont proposés aux utilisateurs.

Les premiers systèmes d'exploitation ont été conçus au début des années 60. En 1964, IBM lance *OS/360*, suivi en 1967 par *Multics*, développé par le *MIT*, *General Electric* et *Bell Labs*. *Multics* a introduit des concepts novateurs, dont une structure de fichiers hiérarchique supportant des accès multi-utilisateurs. Ces concepts ont inspiré le système de fichiers d'*Unix*, conçu par Ken Thompson et Dennis Ritchie, qui les a simplifiés et améliorés pour plus de performance et d'accessibilité.

Au début des années 80, la distribution *BSD – Berkeley Software Distribution –* (initialement un ensemble de logiciels pour *Unix*) développe ses premières variantes du système d'exploitation. Dans les années 90, *BSD* devient peu à peu un logiciel libre. Avec l'arrivée du noyau *Linux*, créé par Linus Torvalds, tous les composants d'un système *Unix* fonctionnel deviennent disponibles en source ouverte. Cela concrétise les aspirations de Richard Stallman, figure majeure du mouvement du logiciel libre.

À la fin des années 90, l'essor des logiciels libres transforme l'écosystème *Unix*. Les distributions *Linux* et *BSD* rendent accessibles les fichiers sources des applications permettant d'en contrôler la sécurité et invitant à une innovation continue. Cette ouverture favorise une collaboration mondiale sans précédent, tout en contribuant à l'amélioration et à l'enrichissement des systèmes *Unix*.

En 2004, Linus Torvalds propose le système de gestion de version de fichiers `git`. Cet outil permet à une communauté mondiale d'acteurs de collaborer efficacement, facilitant la gestion des contributions et améliorant la traçabilité des modifications.

En 2020, les systèmes *Unix* sont omniprésents. Ils administrent la majorité des serveurs qui soutiennent l'Internet, le Web et le Cloud. Ils constituent la base logicielle des smartphones *Android* et *iOS*, où ils s'habillent d'interfaces graphiques dédiées, disparaissant totalement aux yeux de leurs utilisateurs. Ils sont le socle des systèmes d'exploitation *macOS* et des distributions telles qu'*Ubuntu*.

C'est en comprenant la centralité de ses fichiers que l'on peut s'approprier l'étonnante flexibilité d'*Unix* et sa capacité à structurer des usages complexes et variés à partir d'éléments simples et homogènes.

> *« La tâche la plus importante d'Unix est de fournir un système de fichiers. »*
>
> Denis Ritchie et Ken Thompson, *The unix time-sharing system*, [61] page 366.

1.1 Interface système

L'*interface système* est un programme qui fournit à l'utilisateur un moyen d'interagir avec le système d'exploitation et, par là, avec le matériel informatique.

Remarque 1.1 Historiquement, les différentes couches d'un système sont nommées par une analogie à une noix : la coque (*shell*) et le noyau (*kernel*). Le terme *shell* fait référence à la couche la plus haute des interfaces des systèmes Unix, par opposition à la couche de bas niveau, appelée *kernel*. L'idée est que pour accéder à la partie comestible d'une noix, il faut passer par la coque. □

Le shell interprète une série de caractères comme une commande à exécuter. Il donne à cette série de caractères un sens et sollicite le noyau pour qu'il réalise les opérations appropriées. Une fois le traitement terminé, le shell produit une série de caractères en réponse, pour retranscrire sous une forme intelligible le résultat obtenu. Il est parfois traduit en français par le terme *interprète de commandes*.

Cependant, cette série de caractères reçue, puis renvoyée par le shell n'est au mieux qu'une série d'impulsions électriques, qui doit être retranscrite depuis un clavier ou vers un écran. Historiquement, c'était d'abord un matériel qui se chargeait de cela. Ce matériel était appelé *terminal* et l'un des plus connus était le terminal `VT100`. Aujourd'hui, le terminal qui permet de passer des commandes n'est plus un dispositif matériel, mais un logiciel, généralement intégré à l'interface graphique de l'ordinateur utilisé pour interagir. Ce dernier n'est cependant pas nécessairement celui qui interprète et exécute les commandes puisque des « connexions distantes » sont possibles sur un système Unix. Dans ce cas, le terminal et le shell sont exécutés sur des machines distinctes qui échangent des séquences de caractères *via* un réseau informatique.

Il convient donc de distinguer un minimum de trois « couches » de logiciels qui interagissent :

- *le terminal* saisit des séquences de caractères, les transmet au shell et présente sur un écran des séquences de caractères produites en réponse par le shell ;
- *le shell (interprète de commande)* décode les commandes et les transcrit en autant d'appels au kernel que nécessaire, puis transcrit les réponses du kernel en texte pouvant être présenté à l'utilisateur ;
- *le kernel* exécute les opérations appropriées sur le matériel informatique.

Lorsque le système d'exploitation propose une interface graphique, le terminal est souvent présenté comme une application graphique qui peut être ouverte dans une fenêtre de l'interface graphique. Il existe aussi parfois des raccourcis clavier par défaut pour ouvrir un terminal. Sous Ubuntu, pour ouvrir un terminal, on peut faire `Ctrl + Alt + T`. Le shell le plus répandu sous Linux est le *bash* (*Bourne-again shell*). La *console* est la combinaison d'un terminal et d'un shell.

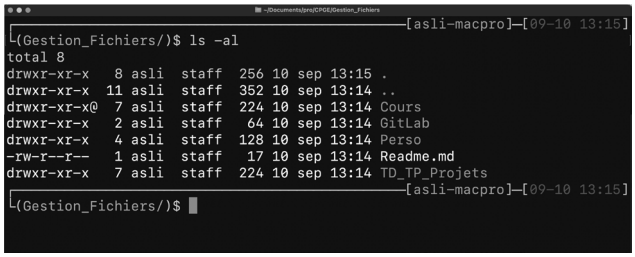


FIGURE 1.1 – Terminal

Les distributions Linux, ainsi que le système macOS et l'environnement WSL Windows permettent de lancer un terminal. La plupart des shells (par exemple *bash* sous Linux, *zsh* sous macOS) adhèrent à des standards communs mais ce n'est pas le cas pour ceux disponibles par défaut sous Windows. Il est cependant possible d'installer, par exemple, *Ubuntu for Windows*. Sous Android, il faut installer un terminal tel que *Android Terminal Emulator* pour avoir accès au shell du système Linux sous-jacent. Comme ce shell est minimaliste, les utilisateurs avertis lui préfèrent une console telle que *Termux* incluant un shell *bash*.

Ce chapitre a été écrit en assumant que le système d'exploitation utilisé serait compatible *POSIX* (*P*ortable *O*perating *S*ystem *I*nterface UNI*X*). POSIX est une famille de normes techniques définie par l'*IEEE* (*I*nstitute of *E*lectrical and *E*lectronics *E*ngineers) qui définit, notamment, un ensemble minimal de commandes que le shell doit supporter.

1.2 Système de fichiers

Cette section introduit les notions fondamentales de l'utilisation d'un système de fichier *Unix*. Des ouvrages plus spécifiquement dédiés aux systèmes d'exploitation permettent d'en approfondir les différents aspects. On recommande notamment la lecture de Pons [60] et du plus ancien Bouzefrane [9].

Un support persistant de stockage de l'information, tel qu'une clé USB, un disque SSD ou même un disque *Blu-ray* ne contient pas *a priori* de fichiers. Le système informatique auquel ils sont connectés ne peut les solliciter que pour obtenir la lecture ou l'écriture de blocs associés à un « numéro de bloc ». Au sein d'un dispositif de stockage persistant, un bloc est composé de n octets (n est typiquement égal à 512 ou 4096, parfois 65536). Il serait fort peu commode d'avoir à mémoriser un numéro de bloc, ou une liste de numéros de blocs, pour savoir retrouver l'information confiée à un tel dispositif. Un système de fichiers permet donc, en premier lieu, de retrouver une séquence d'octets d'une taille arbitraire formant une unité d'information persistante à partir d'un nom. Le service rendu ici est double. D'une part, la taille d'une unité d'information n'est plus contrainte par la taille d'une unité de stockage dans le dispositif matériel et d'autre part, l'identification de l'information ne passe plus par un numéro unique (le numéro du bloc) mais par le nom d'un fichier (une séquence de caractères arbitraire).

Définition 1.1 - Système de fichiers

Un *système de fichiers* est un ensemble de conventions, de structures de données et d'algorithmes destinés à organiser les informations enregistrées dans les blocs d'un support persistant de stockage de l'information pour en faciliter la manipulation.

En premier lieu, un système de fichiers permet donc de créer des fichiers, d'en lire les octets, de les écrire ou encore d'ajouter des octets à la fin d'un fichier existant. Pour cela, les structures de données mises en place par le système de fichiers sur le support de stockage [1] de l'information listent les blocs non encore associés à des fichiers, ce qui constitue l'espace libre du support de stockage. Ils listent aussi les blocs associés à un nom de fichier afin de pouvoir retrouver les données associées au nom du fichier sur le disque. Ils permettent aussi la suppression de fichiers qui libère les blocs utilisés par le fichier et les rend disponibles pour d'autres. Les systèmes de fichiers mettent aussi en œuvre, parfois, des techniques pour protéger l'information, pour en contrôler l'accès ou encore pour garantir l'intégrité des données en cas de panne du dispositif informatique.

Définition 1.2 - Fichier informatique

Un *fichier* informatique est un ensemble de données numériques réunies sous un même nom, enregistrées sur un support de stockage permanent.

Techniquement, un fichier est une information numérique constituée d'une séquence d'octets, c'est-à-dire une séquence de valeurs entre 0 et 255 (correspondant à un nombre composé de 8 bits). Ainsi, quel que soit le fichier considéré, une image, un texte, une musique, il n'est, en pratique, qu'une suite de nombres que les systèmes de fichiers permettent de parcourir en séquence. Certains outils d'édition de fichiers permettent de consulter et de modifier directement cette série de nombres. Ainsi, le mode `M-x hexl-mode` de l'éditeur de texte `emacs` permet de lire et de modifier le contenu d'un fichier sous la forme de paires de chiffres hexadécimaux (une paire de chiffres hexadécimaux est une façon élégante d'afficher l'ensemble des valeurs possibles d'un octet).

Définition 1.3 - Répertoire

Un *répertoire* est un fichier dont le contenu est géré par le système d'exploitation afin d'y enregistrer une liste de fichiers indexés.

Les répertoires sont des fichiers. Simplement, le contenu (la séquence d'octets) d'un répertoire n'est accessible qu'au système d'exploitation lui-même. Un marqueur de

1. Il est possible d'utiliser des supports de stockage volatiles pour y stocker des fichiers, mais ces derniers sont alors perdus à l'arrêt du système informatique.

répertoire (*directory* en anglais) est associé au répertoire afin de le distinguer des autres fichiers. Lorsque ce marqueur est présent, le kernel ne permet pas la consultation des octets qui composent le répertoire, mais seulement son utilisation pour l'indexation de nouveaux fichiers et répertoires en son sein. La notion d'arborescence qui découle de l'utilisation de répertoires sera détaillée dans la section 1.4 Arborescence, page 7.

Un nom de fichier est communément terminé par une « extension ». Par exemple, le fichier nommé `README.md` a pour extension « `.md` ». L'extension d'un fichier est un marqueur ajouté à la fin du nom du fichier pour en suggérer le contenu à l'utilisateur (et parfois au système d'exploitation). Ainsi, l'extension `.md` indique généralement que le fichier est un document *Markdown*, un format de texte léger utilisé pour créer des fichiers lisibles et formatés qui respectent une certaine syntaxe.

Remarque 1.2 Sous Linux, un fichier peut être nommé avec 255 caractères au plus, principalement des lettres et des chiffres mais aussi `.`, `-`, `_`, ∼, `+`, `%`. Les espaces et les accents sont autorisés mais peuvent générer une confusion en pratique. □

Un système de fichiers Unix contient typiquement des millions de fichiers. Pour pouvoir s'orienter dans cette immensité de données, les répertoires sont un outil indispensable. À chaque instant, le shell interprète la commande sollicitée par un utilisateur au regard de ce qui est défini comme le répertoire courant. Le répertoire courant est le nom du répertoire dans lequel sont *a priori* recherchés les fichiers et les répertoires auxquels la commande fait référence.

Les commandes que l'on peut utiliser *via* le terminal permettent de travailler sur des fichiers et des répertoires. Il est possible de connaître le répertoire courant utilisé par le shell avec la commande `pwd` :

```
$ pwd
/home/asli
```

Ici, par défaut, les commandes seront donc interprétées comme devant concerner des fichiers qui se trouvent dans le répertoire `asli` qui est un sous-répertoire du répertoire `home`.

La commande `ls` permet d'afficher les fichiers et les répertoires qui se trouvent dans le *répertoire courant*.

```
$ ls
Cours     GitLab    Perso     README.md   TD_TP_Projets
```

Comme chaque commande, `ls` accepte des options qui permettent d'avoir des informations plus précises sur le contenu du répertoire courant. Par exemple, on peut essayer les commandes suivantes :

```
$ ls -a
.     ..    Cours    GitLab   Perso    Readme.md   TD_TP_Projets
```

L'option `-a` permet d'afficher tous (*all*) les fichiers, y compris ceux qui commencent par un point (`.`). En particulier, dans l'exemple précédent, les répertoires `.` (pour le répertoire courant) et `..` (pour le répertoire *parent*) sont affichés (voir la section 1.4 Arborescence, page 7).

```
$ ls -l
total 15
drwxr-xr-x   8 asli  staff  256 27 aoû 16:40 Cours
drwxr-xr-x   2 asli  staff   64 27 aoû 16:41 GitLab
drwxr-xr-x   4 asli  staff  128 27 aoû 16:40 Perso
-rw-r--r--   1 asli  staff   41 27 aoû 16:40 Readme.md
drwxr-xr-x  10 asli  staff  320 27 aoû 16:43 TD_TP_Projets
```

L'option `-l` permet d'afficher les informations du contenu du répertoire courant en format long : droits du fichier ou du répertoire, nombre de liens, nom du propriétaire, nom du groupe, nombre d'octets dans le fichier, date de la dernière modification en jour, en mois et en heure et le nom du fichier ou du répertoire. Il affiche également, sur la première ligne, le nombre total de blocs, occupés sur le support physique par les fichiers du répertoire courant. De plus, il est possible de combiner des options.

```
$ ls -al
total 15
drwxr-xr-x   8 asli  staff  256 27 aoû 16:43 .
drwxr-xr-x  18 asli  staff  576 27 aoû 16:50 ..
drwxr-xr-x   8 asli  staff  256 27 aoû 16:40 Cours
drwxr-xr-x   2 asli  staff   64 27 aoû 16:41 GitLab
drwxr-xr-x   4 asli  staff  128 27 aoû 16:40 Perso
-rw-r--r--   1 asli  staff   41 27 aoû 16:40 Readme.md
drwxr-xr-x  10 asli  staff  320 27 aoû 16:43 TD_TP_Projets
```

Chaque commande possède plusieurs options. La commande `man` (pour *manual*) permet d'accéder aux pages du manuel des commandes. Ces pages constituent *la* référence pour apprendre l'utilisation des commandes en ligne. Pour accéder aux pages de manuels de la commande `ls`, il suffit de taper `man ls` et ensuite naviguer dans les explications grâce aux flèches du clavier. Pour revenir au terminal, il suffit de taper `q` (pour *quit*). La commande `tldr` (pour *too long, didn't read*) donne accès aux pages `tldr` qui sont un effort communautaire pour simplifier les pages de manuel avec des exemples pratiques.

1.3 Droits d'accès

Les droits d'accès permettent de protéger les fichiers et les répertoires contre toute opération de lecture, d'écriture ou d'exécution non autorisée. De fait, c'est le kernel qui contrôle les droits d'accès lorsque le shell ou tout autre programme lui demande l'accès aux données contenues dans un fichier.

La commande `ls -l` permet d'afficher ces droits d'accès. La première lettre est `d` (pour *directory*) pour les répertoires et `-` pour les fichiers. Ensuite sont affichés trois groupes de trois symboles. Les trois groupes correspondent respectivement au

propriétaire du fichier, au groupe dans lequel le propriétaire se trouve et à tous les autres utilisateurs. Un groupe représente les droits en lecture (`r`), en écriture (`w`) et en exécution (`x`). Si une lettre est affichée, la permission est accordée. Si un tiret est affiché, la permission est refusée. Concernant les répertoires, le droit `x` consiste au droit d'accès à son contenu.

```
-rw-r--r--  1 asli  staff   41 27 aoû 16:40 Readme.md
```

Dans l'exemple ci-dessus, on voit les informations concernent le fichier markdown nommé `Readme.md`. La propriétaire (`asli`) a le droit de lire et d'écrire sur ce fichier. Le groupe d'utilisateurs auquel le fichier est associé (ici `staff`) a le droit de lire, ainsi que tous les autres utilisateurs. Mais les autres droits ne leur sont pas attribués. Si on veut donner les droits d'écriture aux autres utilisateurs, alors il faut utiliser la commande `chmod` (pour *change mode*). Pour ce faire, il faut choisir,

- à qui s'applique le changement : `u` pour *user*, `g` pour *group*, `o` pour *others*, `a` pour *all* ;
- la modification que l'on veut faire : `+` pour ajouter, `-` pour supprimer, `=` pour affecter ;
- le droit que l'on veut modifier : `r` pour *read*, `w` pour *write*, `x` pour *execute*.

Par exemple,

```
$ chmod o+w Readme.md
```

ajoutera le droit d'écriture pour les autres ;

```
$ chmod a+x Readme.md
```

ajoutera le droit d'exécution à tout le monde.

On peut aussi combiner plusieurs actions en même temps.

```
$ chmod u+rwx,g+rx-w,o+r-wx Readme.md
```

1.4 Arborescence

Le système de fichiers est une *arborescence* de fichiers. Dans les systèmes POSIX, les fichiers sont organisés en démarrant par un *répertoire racine*, c'est-à-dire un répertoire contenant tous les autres. Ce répertoire est appelé « `/` ». Le répertoire personnel d'un utilisateur, sur Linux, est le répertoire `/home/xxx`, où `xxx` est le nom de l'utilisateur (voir la figure 1.2), dont `~/` est un raccourci.

Définition 1.4 - Chemin d'un fichier

Le *chemin* d'un fichier ou d'un répertoire est la chaîne de caractères qui permet d'indiquer sa position dans l'arborescence de fichiers. Un chemin *absolu* est le chemin depuis la racine. Un chemin *relatif* est le chemin depuis le répertoire courant.

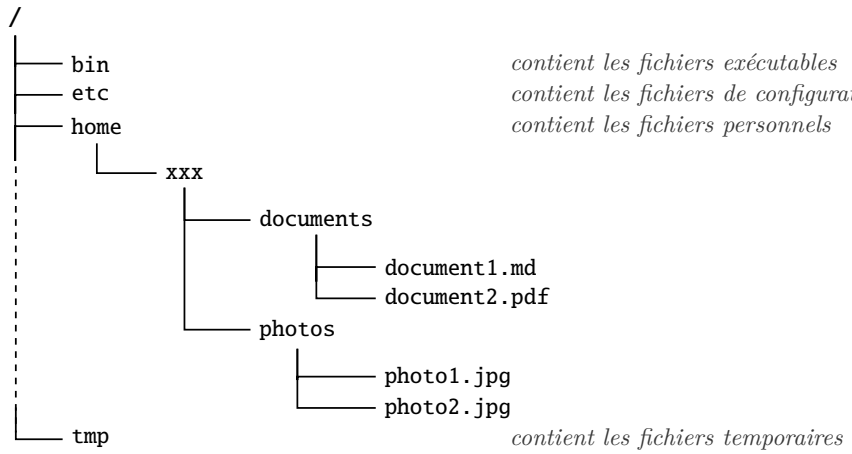


FIGURE 1.2 – Illustration d'une arborescence de fichiers et de répertoires

- La commande `tree` affiche l'arborescence d'un répertoire donné en paramètre. L'option `-d` permet de ne lister que les répertoires. L'option `-L` permet d'indiquer la profondeur de récursion.
- La commande `pwd` affiche le chemin absolu du répertoire où on est situé dans l'arborescence du système de fichiers.
- La commande `cd` permet de se rendre dans un répertoire donné en paramètre. Le répertoire peut être donné avec son chemin absolu ou relatif.
- La commande `mv` permet de déplacer un fichier dans un autre répertoire ou d'en modifier le nom.
- La commande `cp` permet de copier un fichier sous un autre nom.
- La commande `rm` permet de supprimer un fichier. Pour supprimer un répertoire, il faut l'utiliser avec l'option `-r`.
- La commande `touch` permet de créer un fichier vide ou de mettre à jour ses horodatages, tels que la date de dernière modification ou d'accès.
- La commande `mkdir` permet de créer un nouveau répertoire.
- La commande `cat` permet d'afficher le contenu d'un ou plusieurs fichiers (en les concaténant).

1.5 Liens physiques et liens symboliques

Dans les systèmes respectant la norme POSIX, chaque fichier (et donc chaque répertoire) possède un identifiant unique, appelé *inode* (pour *index node*, *nœud d'index* en français). Cet inode permet de déterminer le numéro du bloc associé à la structure de données décrivant le fichier, sur le support matériel de stockage.

L'inode est un entier identifiant le fichier par une structure de données contenant des métadonnées, entre autres :

- les droits du fichier ;
- les propriétaires et groupes du fichier ;
- la taille en octet du fichier ;
- les dates de modification, d'accès et (éventuellement) de création du fichier.

L'option `-i` de la commande `ls` permet d'afficher l'inode de chaque fichier et répertoire listé.

```
$ ls -li
total 16
28160790 drwxr-xr-x 7 asli  staff  224 21 jui 16:18 Cours
28169282 drwxr-xr-x 4 asli  staff  128 21 jui 16:35 Perso
28161114 drwxr-xr-x 5 asli  staff  160 21 jui 16:20 Projets
28171504 -rw-r--r-- 1 asli  staff   29 21 jui 17:24 Readme.md
28161285 drwxr-xr-x 7 asli  staff  224 21 jui 16:33 TD_TP
```

Chaque fichier a un seul inode mais peut avoir plusieurs noms, chacun faisant référence au même inode. Les inodes contiennent toutes les informations sur les fichiers à part le ou les noms. Les *liens* sont utiles pour faire apparaître un même fichier dans plusieurs répertoires, ou sous des noms différents. Ils évitent les duplications et assurent la cohérence des mises à jour.

Liens physiques

Un *lien physique* (*hard link*) est une référence directe à un fichier *via* son inode. La commande `ln` permet de créer un lien physique. L'option `-i` vérifie si le fichier demandé existe.

```
$ ln -i Readme.md Readme_HL.md
$ ls -li
total 24
28160790 drwxr-xr-x  7 asli  staff  224 21 jui 16:18 Cours
28169282 drwxr-xr-x  4 asli  staff  128 21 jui 16:35 Perso
28161114 drwxr-xr-x  5 asli  staff  160 21 jui 16:20 Projets
28171504 -rw-r--r--  2 asli  staff   29 21 jui 17:24 Readme.md
28171504 -rw-r--r--  2 asli  staff   29 21 jui 17:24 Readme_HL.md
28161285 drwxr-xr-x  7 asli  staff  224 21 jui 16:33 TD_TP
$ cat Readme.md
Readme file.
$ cat Readme_HL.md
Readme file.
```

Les fichiers `Readme.md` et `Readme_HL.md` ne sont que deux noms distincts associés à un unique fichier : ils ont la même inode et donc les mêmes droits ainsi qu'une même liste de blocs, contenant un même contenu, etc. On peut changer le contenu ou la localisation du fichier original, le lien physique restera valide. Si l'un des deux fichiers est effacé l'autre continue à exister.

```
$ echo "Hello World" >> Readme.md
$ cat Readme.md
Readme file.
Hello World
$ cat Readme_HL.md
Readme file.
Hello World
$ rm Readme.md
$ cat Readle_HL.md
Readme file.
Hello World
$ mv Readme_HL.md Readme.md
```

Il n'est possible de créer de lien physique qu'entre des fichiers, et pas entre des répertoires. De plus, il faut que ces fichiers se trouvent sur le même support physique. En effet, deux disques logiques différents peuvent avoir chacun une même inode.

Liens symboliques

Un *lien symbolique* (*symbolic link*) est une référence à un nom contrairement à un lien physique, qui, lui, réfère à un inode. La commande `ln` permet de créer un lien symbolique lorsque l'option `-s` est utilisée. L'option `-i` vérifie toujours si le fichier demandé existe.

```
$ ln -is Readme.md Readme_SL.md
$ ls -li
total 16
28160790 drwxr-xr-x  7 asli  staff  224 21 jui 16:18 Cours
28169282 drwxr-xr-x  4 asli  staff  128 21 jui 16:35 Perso
28161114 drwxr-xr-x  5 asli  staff  160 21 jui 16:20 Projets
28171504 -rw-r--r--@ 1 asli  staff   41 21 jui 18:26 Readme.md
28178949 lrwxr-xr-x  1 asli  staff    9 21 jui 20:45 Readme_SL.md
    ->Readme.md
28161285 drwxr-xr-x  7 asli  staff  224 21 jui 16:33 TD_TP
$ cat Readme_SL.md
Readme file.
Hello World
```

Les fichiers `Readme.md` et `Readme_SL.md` ont des inodes différents et sont de tailles différentes. La première information du fichier `Readme_SL.md` est `l`, pour *link*. Si la source est effacée, déplacée ou renommée, le lien symbolique est cassé même si le fichier lien continue à exister. Si un nouveau fichier est créé avec le même nom, le lien pointe vers ce dernier.

```
$ mv Readme.md Readme_tmp.md
$ cat Readme_SL.md
cat: Readme_SL.md: No such file or directory
$ touch Readme.md
$ echo "Hello World 2" > Readme.md
$ cat Readme_SL.md
Hello World 2
$ rm Readme.md; rm Readme_SL.md; mv Readme_tmp.md Readme.md
```

Avec un lien symbolique, il est possible de créer des liens entre des fichiers se trouvant sur différentes partitions. Il est également possible de créer des liens symboliques pointant vers un répertoire.

```
$ ln -is Projets Projets_SL
$ ls -li
total 16
28160790 drwxr-xr-x  7 asli  staff  224 21 jui 16:18 Cours
28169282 drwxr-xr-x  4 asli  staff  128 21 jui 16:35 Perso
28161114 drwxr-xr-x  5 asli  staff  160 21 jui 16:20 Projets
28179053 lrwxr-xr-x  1 asli  staff    7 21 jui 20:54 Projets_SL ->
     Projets
28171504 -rw-r--r--@ 1 asli  staff   41 21 jui 18:26 Readme.md
28161285 drwxr-xr-x  7 asli  staff  224 21 jui 16:33 TD_TP
$ rm Projets_SL
```

1.6 Redirections

Trois *flux standards* peuvent être utilisés sans qu'il soit nécessaire de les ouvrir (et donc de les fermer) :

- le flux standard d'entrée *stdin* (pour *standard input*), par défaut l'entrée est le clavier ;
- le flux standard de sortie *stdout* (pour *standard output*), par défaut la sortie est l'écran ;
- le flux standard d'erreur *stderr* (pour *standard error*), par défaut l'erreur s'affiche sur l'écran.

Dans un shell POSIX, l'opérateur > permet de rediriger la sortie standard d'un programme vers un fichier.

```
$ ls -al Projets > contenu.md
$ cat contenu.md
total 0
drwxr-xr-x  5 asli  staff  160 21 jui 16:20 .
drwxr-xr-x  9 asli  staff  288 22 jui 14:03 ..
drwxr-xr-x  3 asli  staff   96 21 jui 16:22 EscapeGame
drwxr-xr-x  4 asli  staff  128 21 jui 16:21 Mandelbrot
drwxr-xr-x  5 asli  staff  160 21 jui 17:17 Structures
```

La commande précédente n'a aucun affichage sur la sortie standard et tout ce qui devait être affiché sur le terminal, a été écrit dans le fichier `contenu.md`. Si ce dernier existe, son contenu est ecrasé, sinon il est créé et écrit.

Si l'exécution d'une commande provoque une erreur, le message associé est affiché sur le terminal.

```
$ ls -al Projet 2> erreur.md
$ cat erreur.md
ls: Projet: No such file or directory
```

L'opérateur `2>` permet de rediriger la sortie d'erreur d'un programme vers un fichier mais la sortie standard est écrite sur le terminal. Afin de récupérer les deux sorties standards sur des fichiers, on peut les utiliser d'une façon séquentielle.

```
$ ls -al TD_TP_Projets Projet 2> erreur.md > contenu.md
```

Les opérateurs `>` et `2>` écrasent le contenu du fichier vers lequel la sortie est redirigée. Afin d'ajouter les sorties à la fin d'un fichier existant, et donc ne pas écraser son contenu, il faut utiliser les opérateurs `>>` et `2>>`. Par ailleurs, le flux standard d'entrée, typiquement les entrées *via* le clavier, peut être récupéré *via* un programme.

1.7 Commandes indispensables et tube

Commandes

La commande `echo` affiche un message. On peut afficher également le contenu d'une variable d'environnement, la date, l'évaluation d'une expression, etc.

```
$ echo "Hello World"
Hello World
$ echo $HOME $LANG
/home/asli fr_FR.UTF-8
$ echo "Nous sommes le"  `date +%d/%m/%y`
Nous sommes le 22/06/23
$ echo `expr 2 + 2`
4
```

La commande `find` cherche récursivement dans l'arborescence à partir du répertoire indiqué.

```
$ find . -name "stack*"
./Projets/Structures/stack_link.c
./Projets/Structures/stack_tab.c
```

La commande `head` affiche les premières lignes d'un fichier. Si le nombre de lignes à afficher est omis, la commande affiche les 10 premières.

```
$ head -n 2 ./Projets/Structures/stack_link.c
struct stack_s {
  int value;
```

La commande `tail` affiche, de la même façon, les dernières lignes d'un fichier.

```
$ tail -n 2 ./Projets/Structures/stack_link.c
    return 0;
}
```

La commande `wc` affiche le nombre de lignes (`-l`), le nombre de mots (`-w`) et le nombre de caractères (`-c`) d'un fichier donné en paramètre. Sans option (`-l`, `-w` ou `-c`) les trois informations sont affichées. Il faut encore noter que les caractères comptés sont les codes `ASCII` (donc les octets). Ainsi les caractères étendus tels que les caractères accentués ou les kanji par exemple, sont codés sur plus d'un octet, et comptent donc pour plus d'un caractère selon `wc`.

```
$ wc ./Projets/Structures/stack_link.c
111     276    2665 ./Projets/Structures/stack_link.c
```

La commande `grep` permet de filtrer un texte, soit écrit dans un fichier soit fourni par une autre commande, en cherchant le mot donné.

```
$ grep "stack_delete" ./Projets/Structures/stack_tab.c
void stack_delete(struct stack_s *stack) {
  stack_delete(my_stack);
  printf("stack_delete\n");
$ grep -c "stack_delete" ./Projets/Structures/stack_tab.c
3
$ grep -n "stack_delete" ./Projets/Structures/stack_tab.c
56:void stack_delete(struct stack_s *stack) {
107:    stack_delete(my_stack);
108:    printf("stack_delete\n");
```

L'option `-c` compte le nombre d'occurrences du mot recherché. L'option `-n` affiche le numéro des lignes associées. Cette commande possède plusieurs autres options très utiles dans la recherche dans un texte.

```
$ grep "return 0;$" ./Projets/Structures/stack_tab.c
if(*string=='\0') return 0;
    return 0;
  return 0;
$ grep "^const" ./Projets/Structures/stack_tab.c
const int MAX_STACK_SIZE=100;
```

Lorsque l'on ajoute le caractère `$` à la fin d'une chaîne de caractères recherchée, cela indique que l'on souhaite trouver uniquement les lignes qui se terminent par cette chaîne. De la même manière, le caractère ^ placé au début d'une chaîne de caractères signifie que l'on recherche uniquement les lignes qui commencent par cette chaîne.

Tube

Un tube (*pipeline*), est un mécanisme de communication entre les processus. L'utilisation d'un tube permet de combiner toutes les commandes vues précédemment. La sortie d'une commande est redirigée vers l'entrée d'une autre. L'opérateur `|` réalise cette opération.

```
$ ls -al | grep "^-"
-rw-r--r--  1 asli  staff   41 21 jui 18:26 Readme.md
-rw-r--r--  1 asli  staff  266 22 jui 14:03 contenu.md
-rw-r--r--  1 asli  staff   38 22 jui 14:04 erreur.md
```

Cette commande permet de récupérer que la liste des fichiers du répertoire courant.

```
$ head -n 87 ./Projets/Structures/stack_link.c | tail -n 1
        if(strcmp(argv[i],"#")==0) {
```

Cette commande permet d'afficher la ligne 87 du fichier donné en paramètre.

Expressions régulières

Une expression régulière (*regex* pour *regular expression*), est une chaîne de caractères qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles. Leur utilisation facilite l'analyse et la manipulation des langages informatiques. Quelques règles de base pour les expressions régulières et quelques exemples d'utilisation de ces dernières sont donnés ci-dessous.

- `*` désigne une chaîne de caractères quelconque ;
- `?` désigne un caractère quelconque ;
- `[...]` désigne les caractères entre crochets, définis par énumération ou par un intervalle :
  - `[Aa]` désigne les caractères `A` ou `a` ;
  - `[0-9a-zA-Z]` désigne un caractère alphanumérique quelconque ;
  - `[!0-9]` désigne l'ensemble des caractères sauf les chiffres.

```
$ ls R*
Readme.md
$ ls TD_TP
td1.cmi td1.cmo td1.exe td1.ml  td2.ml
$ ls TD_TP/*1*
TD_TP/td1.cmi TD_TP/td1.cmo TD_TP/td1.exe TD_TP/td1.ml
$ ls TD_TP/*.cm[io]
TD_TP/td1.cmi TD_TP/td1.cmo
$ ls TD_TP/???.ml
TD_TP/td1.ml TD_TP/td2.ml
$ ls TD_TP/td1.cm[\!i]
TD_TP/td1.cmo
```

Historique

Le shell bash enregistre toutes les commandes tapées et permet de les rappeler pour les réexécuter soit telles quelles, soit modifiées.

- La commande `history` permet de lister le contenu de l'historique des commandes, de façon numérotée.
- La commande `!n` rappelle la commande numéro `n`.
- La commande `!chaine` rappelle la dernière commande qui commence par `chaine`.
- Les flèches haut et bas du clavier permettent de naviguer dans l'historique des commandes.

1.8 Makefile

La réalisation d'un projet informatique consiste généralement en la création d'un ensemble de fichiers, qu'il s'agisse de programmes, de vidéos, ou de documents manuscrits, par exemple. La production de ces fichiers est souvent obtenue en exécutant des outils qui transforment des *fichiers sources* en *fichiers cibles*. Par exemple, un fichier PDF (`.pdf`) peut être produit à partir d'un fichier markdown (`.md`) en utilisant l'outil `pandoc`. Il n'est généralement pas nécessaire de régénérer tous les fichiers cibles à partir des fichiers sources, mais seulement ceux dont les sources ont été modifiées depuis la dernière production. C'est pour automatiser cette production sans réexécuter toutes les opérations que la commande `make` prend tout son sens.

Les `Makefile`s sont des fichiers utilisés par le programme `make(1)` afin d'automatiser un ensemble d'actions, qui peuvent permettre la génération de fichiers cibles, la plupart du temps résultant de l'exécution de programmes à partir d'autres fichiers, appelés *dépendances*.

Structure des règles dans un `Makefile` Un `Makefile` est composé d'un ensemble de règles de la forme suivante.

```
cible [cible...]: [dépendance...]
    commande
    ...
    commande
```

Chaque règle commence par une ou plusieurs cibles suivies par le caractère `:` et éventuellement une liste de dépendances dont dépend la cible. Une cible ou une dépendance peut être un fichier ou un simple label. Une commande doit être précédé nécessairement d'une tabulation. Chacune des commandes est une ligne de commande shell. Les commandes sont exécutées dans l'ordre donné pour la cible.

Les dépendances peuvent être des fichiers dont le contenu est édité par l'utilisateur, ou bien eux-mêmes des fichiers cibles d'autres règles. Dans le premier cas, la commande `make` réexécute les commandes qui génèrent le ou les fichiers cibles lorsque

ces fichiers sont plus anciens que l'une des dépendances. Dans le second cas, la commande `make` évalue en premier lieu la nécessité de générer la dépendance à partir de la règle qui la concerne, puis elle évalue la nécessité de générer la cible comme elle l'aurait fait pour une dépendance sans règle associée.

Exemples spécifiques On considère un fichier nommé `Gestion_Fichiers.md`. Pour créer un fichier `.pdf` du même nom, on peut utiliser la commande suivante.

```
$ pandoc -o Gestion_Fichiers.pdf Gestion_Fichiers.md
```

Pour automatiser cette création, on peut écrire un `Makefile` :

```
Gestion_Fichiers.pdf: Gestion_Fichiers.md
    pandoc -o Gestion_Fichiers.pdf Gestion_Fichiers.md
```

On peut également compiler des fichiers `.ml` et `.c` en exécutables.

```
additionocaml: addition.ml
    ocamlc -o additionocaml addition.ml

additionc: addition.c
    gcc -o additionc addition.c
```

Utilisation des variables. Les variables dans un `Makefile` rendent les règles plus modulaires et faciles à maintenir. Par exemple :

```
CC := gcc
CFLAGS := -Wall

additionc: addition.c
    $(CC) $(CFLAGS) -o addition addition.c
```

Il existe des variables automatiques qui sont des raccourcis, propre à la syntaxe des fichiers `Makefile`, qui permettent d'exprimer des valeurs déjà présentes dans le fichier afin d'éviter les répétitions. Elles sont généralement formées de deux caractères, comme par exemple : `$@`, `$<` ou encore `$^`.

- `$@` fait référence au nom de la cible.
- `$<` fait référence au nom de la première dépendance.
- `$^` fait référence à la liste des dépendances.

En utilisant ces variables automatiques, le `Makefile` précédent devient alors comme ci-dessous.

```
CC := gcc
CFLAGS := -Wall

additionc: addition.c
    $(CC) $(CFLAGS) -o $@ $<
```

Règles spéciales et directive `.PHONY` Il est souvent utile d'avoir des règles qui ne produisent pas de fichiers, comme `all` pour compiler plusieurs cibles par défaut, ou `clean` pour supprimer les fichiers générés. Ces règles exécutent simplement des commandes.

```
FILE := Gestion_Fichiers

all: $(FILE).pdf $(FILE).html $(FILE).docx additionocaml additionc

clean:
    rm $(FILE).pdf $(FILE).html $(FILE).docx additionocaml
    additionc *.o *.cmi *.cmx *.cmo
```

Une fois ces règles définies, il est possible d'exécuter :

```
$ make clean
```

pour que la règle spécifique soit spécifiquement appliquée.

Pour indiquer à `make` ces règles spéciales, on utilise la directive spéciale `.PHONY` :

```
.PHONY: all clean
```

Exemple avancé Voici un exemple de `Makefile` qui génère une page de « titre », une page de « chapitre » et une page de « révision ». La page de révision dépend de la page de titre et de la page de chapitre et utilise les commandes `echo` et `date` pour générer son contenu. Le fichier final est un fichier PDF concaténé à l'aide de la commande `pdfunite`.

```
PANDOC := pandoc
PDFUNITE := pdfunite

.PHONY: all clean
all: final_document.pdf

final_document.pdf: title.pdf chapter.pdf revision.pdf
  $(PDFUNITE) $^ $@

title.pdf: title.md
  $(PANDOC) -o $@ $<

chapter.pdf: chapter.md
  $(PANDOC) -o $@ $<

revision.pdf: title.pdf chapter.pdf
  echo "Document revision generated on " > /tmp/revision.md
  date >> /tmp/revision.md
  $(PANDOC) -o $@ /tmp/revision.md

clean:
    rm -f *.pdf /tmp/revision.md
```

Ce `Makefile` commence par définir deux variables pour les commandes `pandoc` et `pdfunite` afin de rendre les règles plus modulaires. La première règle `all` spécifie que la cible principale est `final_document.pdf`. Cette cible dépend de `title.pdf`, `chapter.pdf` et `revision.pdf`. La règle `final_document.pdf` utilise la commande `pdfunite` pour concaténer les trois fichiers PDF en un seul fichier final. Les règles `title.pdf` et `chapter.pdf` spécifient comment générer les fichiers PDF à partir des fichiers markdown (`.md`) respectifs en utilisant `pandoc`. La règle `revision.pdf` dépend des fichiers `title.pdf` et `chapter.pdf`. Elle utilise les commandes `echo` et `pandoc` pour générer une page de révision qui inclut la date de production. Enfin, la règle `clean` supprime tous les fichiers PDF générés pour permettre une reconstruction propre.

Conformément à ce fichier `Makefile`, lorsque le fichier de titre (`title.md`) est modifié, les étapes suivantes sont exécutées par la commande `make` :

- `title.pdf` est régénéré à partir de `title.md`.
- `revision.pdf` est régénéré puisque `title.pdf` est une de ses dépendances.
- `final_document.pdf` est régénéré en concaténant les trois fichiers `title.pdf`, `chapter.pdf` et `revision.pdf`.

Ici le fichier `chapter.pdf` n'est pas régénéré puisque e fichier `chapter.md`, qui est sa dépendance, n'a pas été modifié. À l'inverse, lorsque seul le fichier de chapitre (`chapter.md`) est modifié, les étapes suivantes sont exécutées :

- `chapter.pdf` est régénéré à partir de `chapter.md`.
- `revision.pdf` est régénéré puisque `chapter.pdf` est une de ses dépendances.
- `final_document.pdf` est régénéré en concaténant les trois fichiers `title.pdf`, `chapter.pdf` et `revision.pdf`.

Cet exemple illustre les nombreux avantages de l'utilisation d'un `Makefile`. Par exemple, cela permet d'automatiser des mises à jour du contenu, comme le montre la règle `revision.pdf`. La date de production du PDF est mise à jour sans intervention des auteurs. La recompilation des différentes parties du document sont menées de façon indépendante et tout le traitement n'est pas refait à chaque exécution de la commande `make`. Lorsque le nombre de chapitres de ce document sera plus important, le bénéfice sera particulièrement important, en temps d'exécution. Seul la concaténation finale et la partie du document modifiée seront réexécutées.

Remarque 1.3 Cet ouvrage utilise des `Makefile` pour compiler de manière organisée des fichiers écrits en `C` et en `OCaml`. L'outil `Dune`, spécialement conçu pour gérer les projets développés en `OCaml`, aurait pu être choisi. Il automatise de nombreux aspects du processus de compilation et simplifie la gestion des dépendances. Cependant, une approche différente a été retenue pour des raisons pédagogiques. Le recours à un `Makefile` explicite chaque étape de la compilation, ce qui facilite la compréhension des liens entre les fichiers source, les dépendances et les fichiers générés. Cette méthode vise à démystifier le processus de compilation tout en offrant une vue plus transparente des mécanismes sous-jacents. □

1.9 TD/TP - Gestion des fichiers

Exercice 1 : `Chmod` en octal ⇝ *p.561*

1) Ouvrir un terminal et utiliser la commande `man` pour consulter l'aide associée à la commande `chmod` en octal. Pour ce faire, lancer la commande suivante :

```
$ man chmod
```

Utiliser les touches suivantes pour naviguer dans le manuel :

- les flèches du curseur ↑ et ↓ permettent de défiler le texte d'une ligne à la fois ;
- la touche ␣ (espace) permet de défiler le texte vers le bas une page à la fois ;
- la touche `b` permet de remonter d'une page ;
- la touche `/` suivi d'un mot-clé permet de rechercher un texte spécifique ;
- la touche `q` permet de quitter le manuel.

Lire la section expliquant l'utilisation de `chmod` avec des valeurs octales.

2) Justifier l'utilisation du mot « octal ».

3) Justifier la correspondance entre `r` et 4, `w` et 2, `x` et 1 et enfin entre `-` et 0.

4) On a un fichier `fichier.md` et on exécute la commande suivante.

```
$ chmod 640 fichier.md
```

Donner les droits du propriétaire, les droits du groupe et les droits de tous les utilisateurs sur ce fichier.

5) On souhaite maintenant que le fichier `fichier.md` de la question précédente puisse être lu, écrit et exécuté par le propriétaire ; lu et exécuté par le groupe et lu par les autres utilisateurs. Donner une commande qui peut être utilisée.

Exercice 2 : Manipulation de répertoires et de fichiers ⇝ *p.561*

Le but de cet exercice est de manipuler les répertoires et les fichiers en utilisant des commandes Unix. Pour chacune des questions suivantes, donner les commandes permettant de réaliser les actions demandées.

1) Faire afficher le chemin absolu du répertoire courant.

2) Se déplacer directement à la base du répertoire de travail.

3) Se déplacer dans le répertoire `Documents`.

4) Réaliser les actions suivantes dans l'ordre :

- créer un répertoire `TPs_informatique` ;
- créer un répertoire `TP_Gestion_Fichiers` à l'intérieur de ce répertoire ;
- en faire le répertoire courant.

5) Créer un répertoire `TP1` dans `TP_Gestion_Fichiers`, se déplacer dans ce répertoire, et puis créer un fichier `README.md`.

6) Indiquer dans le fichier `README.md` l'ensemble des réponses aux questions de l'exercice, en utilisant (par exemple) l'éditeur `emacs`.

7) Limiter les droits d'accès au fichier `README.md` de telle sorte que l'usage en lecture et écriture soit limité au seul propriétaire du fichier. Faire de même avec le répertoire `TPs_informatique`.

Exercice 3 : Manipulation de fichiers et de commandes ⇝ *p.561*

Pour chacune des questions suivantes, donner les commandes permettant de réaliser les actions demandées.

1) Se placer dans le répertoire `TP1` créé à l'issue de l'exercice précédent.

2) Afficher le traditionnel *Hello World !* avec la commande `echo`.

3) Afficher la date et l'heure courantes avec la commande `date`.

4) Créer un fichier vide intitulé `exercice3.md` avec la commande `touch`.

5) Utiliser la commande `echo` pour mettre dans le fichier `exercice3.md` le texte « `Exercice 3` », puis le texte « `=====` ». Ajouter ensuite la date courante à la fin de ce fichier.

6) Utiliser la commande `cat` pour vérifier que le contenu du fichier est bien formé de trois lignes : la première avec le titre, la seconde avec le souligné de « = », et la troisième avec la date du jour.

7) Utiliser la commande `hexdump` avec l'option `-C` pour observer le contenu octet par octet du fichier. Donner les valeurs des trois premiers octets de ce fichier en hexadécimal.

8) Retirer tous les droits d'accès au fichier pour tous les utilisateurs possibles du système sur le fichier, puis réitérer la consultation du fichier avec les commandes `cat` puis `hexdump`. Expliquer ce qui est observé.

9) Consulter le contenu du répertoire courant et déterminer le nom du propriétaire du fichier.

10) Réautoriser le propriétaire à avoir l'accès au fichier, et vérifier que les commandes de consultation refonctionnent.

11) Créer une copie du fichier dans `exercice3.bis.md`, un lien symbolique nommé `exercice3.sl.md` et un lien physique nommé `exercice3.hl.md`.

12) Retirer les droits utilisateur du fichier `exercice3.md`, puis tester si les commandes `cat` et `hexdump` fonctionnent sur les trois fichiers précédents.

Exercice 4 : Arborescence ⇝ *p.562*

1) Créer un répertoire `arborescence` en ligne de commande et se déplacer dans ce répertoire.

2) On exécute les commandes ci-dessous dans l'ordre dans un terminal.

```
$ mkdir Cours Perso TD_TP_Projets
$ cd Perso
$ mkdir images
$ echo "Vacances le 19 Octobre" > dates.txt
$ cd ../TD_TP_Projets
$ mkdir TP_Gestion_Fichiers TP_Booleens
$ date > TP_Gestion_Fichiers/exercice.md
$ echo "Exercice sur les arborescences" >> TP_Gestion_Fichiers/
    exercice.md
$ cd ..
$ cd Cours
$ mkdir Gestion_Fichiers Organisation
$ cd Organisation
$ echo "Inscription à la fac" > todo.md
$ echo "Inscription à la cantine" >> todo.md
$ cp todo.md ../../TD_TP_Projets
$ echo "Finir l'exercice 4 en info" > todo.md
```

Dessiner l'arborescence finale des fichiers et répertoires depuis le répertoire `arborescence`.

3) Exécuter les commandes de la question précédente et vérifier l'arborescence *via* la commande `tree`.

Exercice 5 : ★ Liens logiques, liens physiques ⇝ *p.563*

1) Consulter la documentation de l'option `-i` de la commande `ls`.

On considère la séquence de commandes suivante :

```
$ echo "hello" > f1.txt
$ echo "bye" > f2.txt
$ cp f1.txt f3.txt
$ ln f1.txt f4.txt
$ ln -s f1.txt f5.txt
```

2) Exécuter ces commandes dans un répertoire de test nommé `test_ln` créé préalablement.

3) Noter les numéros d'inode des fichiers créés. Indiquer la différence observée entre `f4.txt` et `f5.txt`.

4) Donner le contenu affiché avec `cat f4.txt` et `cat f5.txt`.

5) Exécuter la commande `rm f1.txt`. Indiquer la conséquence de cette commande sur ce qui est affiché avec `cat f4.txt` et `cat f5.txt`.

6) Recréer le lien physique entre `f4.txt` et `f1.txt`.

7) Exécuter la commande `mv f2.txt f1.txt` et noter les numéros d'inode. Donner le nouveau contenu des fichiers `f4.txt` et `f5.txt`. Expliquer le fonctionnement de la commande `mv`.

8) Supprimer à nouveau le fichier `f1.txt`, puis recréer le lien physique entre le fichier `f4.txt` et le fichier `f1.txt`.

9) Recréer le fichier `f2.txt` avec `echo "bye" > f2.txt`, puis exécuter `cp f2.txt f1.txt`. Observer à nouveau les numéros d'inode. Donner le nouveau contenu des fichiers `f4.txt` et `f5.txt`. Expliquer le fonctionnement de la commande `cp`.

10) Justifier en quoi il est possible d'affirmer qu'un lien symbolique crée une redirection, alors qu'un lien physique crée une référence depuis un fichier vers un fichier tiers. Expliquer la différence entre ces deux mécanismes.

Exercice 6 : ★ Comprendre un projet par son `Makefile` ⇝ *p.564*

Dans cet exercice, il s'agit de comprendre l'organisation d'un projet informatique en analysant le `Makefile` mis en œuvre pour le produire à partir du travail des contributeurs dans des fichiers source.

Caractère joker (`%`) Le caractère joker `%` est utilisé dans les règles du `Makefile` pour représenter une partie variable d'un nom de fichier. Par exemple, dans la règle `%.o: %.c`, le `%` peut correspondre à n'importe quel nom de fichier, de sorte que `f1.c` est transformé en `f1.o`, `f2.c` en `f2.o`, etc.

`$(wildcard)` La fonction `$(wildcard PATTERN)` renvoie une liste de fichiers correspondant au motif `PATTERN`. Par exemple, `$(wildcard tmp/*.c)` renvoie tous les fichiers se terminant par `.c` dans le répertoire `tmp/`.

`$(patsubst)` La fonction `$(patsubst PATTERN, REPLACEMENT, TEXT)` remplace toutes les occurrences de `PATTERN` par `REPLACEMENT` dans `TEXT`. Par exemple, la commande `$(patsubst %.c, /tmp/%.o, f1.c f2.c f3.c))` transforme une liste de fichiers source `.c` en une liste de fichiers objets `.o`, plus précisément en `/tmp/f1.o /tmp/f2.o /tmp/f3.o`.

```
1  # Makefile for C project
2  # Authors: A. & G. Grimaud
3  # License: LGPL
4
5  # Directory variables
6  SRC := src
7  BLD := build
8  BIN := bin
9  INC := include
10
11 # Default executable name
12 EXEC := add
13
14 # Compiler and flags
15 CC := gcc
16 CFLAGS := -I$(INC) -Wall -Wextra -pedantic
```

```
17 # Source files and object files
18 SRCS := $(wildcard $(SRC)/*.c)
19 HDRS := $(wildcard $(INC)/*.h)
20 OBJS := $(patsubst $(SRC)/%.c,$(BLD)/%.o,$(SRCS))
21
22 # Phony targets
23 .PHONY: all clean
24
25 # Target for the default executables
26 all: $(BIN)/$(EXEC)
27
28 # Rule to link the object files into the executables
29 $(BIN)/$(EXEC): $(OBJS)
30   mkdir -p $(BIN)
31   $(CC) $(OBJS) -o $@
32
33 # Rule to compile source files into object files
34 $(BLD)/%.o: $(SRC)/%.c $(HDRS)
35   mkdir -p $(BLD)
36   $(CC) $(CFLAGS) -c $< -o $@
37
38 # Rule to clean the generated files
39 clean:
40   rm -rf $(BLD)/*.o $(BIN)/$(EXEC)
```

1) Donner le fichier produit par le processus de compilation avec `make`.

2) Donner les cibles du `Makefile` correspondant à des règles spéciales. Expliquer leur fonction respective.

3) Analyser la structure des répertoires du projet informatique. Décrire le type de fichiers qu'on retrouve dans chaque répertoire.

4) Identifier les fichiers de dépendance utilisés pour générer le fichier principal lors de l'exécution de la commande `make`.

5) Identifier les fichiers de dépendance nécessaires pour créer les fichiers de dépendances identifiés dans la question précédente.

En exécutant la commande `ls src include` à la base du projet (au niveau du `Makefile`), on obtient :

```
1 $ ls src include
2 include:
3 f1.h f2.h f3.h
4 src:
5 f1.c f2.c f3.c
```

6) Après une première compilation du projet avec `make`, on modifie le contenu du fichier `src/f1.c`. Donner les commandes exécutées pour produire le fichier principal lors de la réexécution de `make`.

7) On modifie ensuite le fichier `f1.h` dans le répertoire `include`. Donner les commandes exécutées pour produire le fichier principal lors de la réexécution de `make`.

Chapitre 2
Booléens

Les fondements de l'algèbre de Boole, initialement nommée « l'algèbre de la logique », ont été posés dès 1844 par George Boole avec de notables contributions d'Augustus De Morgan. L'ambition de G. Boole et de ce qu'il convient d'appeler l'École Algébrique Anglaise s'attache, en cette première moitié du XIX^e^ siècle, à développer une définition à la fois symbolique et algorithmique de l'algèbre [28].

George Boole compile finalement ses réflexions en la matière dans un ouvrage intitulé *An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities*, publié en 1854 [8]. Il s'agit de transcrire sous forme algébrique des opérations de raisonnement habituellement confiées à l'argumentation verbale (ou argumentation logique).

> *« La forme à laquelle ces lois donnent naissance semble, en effet, correspondre à la forme d'une langue parfaite. Imaginons une langue connue ou existante, débarrassée de ses idiotismes et dépouillée de superfluités, et exprimons dans cette langue toute proposition donnée de la manière la plus simple et littérale, la plus conforme aux principes de la pensée pure et universelle sur lesquels toutes les langues sont fondées, dont toutes portent la manifestation, mais dont toutes s'écartent plus ou moins. La transition d'une telle langue à la notation de l'analyse consisterait en rien de plus qu'une substitution d'un ensemble de signes par un autre, sans changement essentiel ni de forme ni de caractère. [...] Sa réalité et sa complétude seront rendues plus apparentes par l'étude de ces formes d'expression qui se présenteront dans les applications ultérieures de la présente théorie, vues en comparaison plus immédiate avec cet instrument de pensée imparfait mais noble qu'est la langue anglaise. »*
>
> George Boole, *An Investigation of the Laws of Thought*, [8], page 174

Ces travaux marquent la première étape du processus de mathématisation de la logique, après plus de vingt siècles d'une structuration de la logique à partir de l'analyse aristotélicienne du langage. Cependant, à la fin du XIX^e^ siècle, l'École Algébrique Anglaise décline, et ses résultats sont oubliés. Les travaux postérieurs de C.S. Peirce ne seront publiés qu'en 1931, à titre posthume, dans *Collected Papers* [58].

En 1938, Claude E. Shannon montre, dans sa thèse de maîtrise intitulée *A Symbolic Analysis of Relay and Switching Circuits*, que l'algèbre de Boole est à la fois celle du calcul des propositions et celle des circuits à relais et commutateurs [65]. Il démontre alors que les circuits électriques peuvent implémenter des opérations logiques telles que `ET`, `OU`, et `NON`, ainsi que, par extension, n'importe quelle opération arithmétique.

Les travaux de C.E. Shannon jettent les fondations théoriques de la conception des circuits logiques, qui sont depuis utilisés pour réaliser des calculs et des traitements de données. Cette avancée est au cœur de technologies telles que les processeurs et fait de l'algèbre de Boole la pierre angulaire de l'ère de l'information.

2.1 Algèbre de Boole

Dans l'algèbre de Boole [7], les valeurs des variables sont des *valeurs de vérité vrai* ou *faux*, aussi appelées *booléens*, généralement notés par 1 et 0. Une *proposition* est une assertion qui ne peut prendre que ces deux valeurs. L'algèbre de Boole utilise des opérateurs logiques comme :

- la conjonction *et* notée $\wedge$ (ou la multiplication de propositions : $A \cdot B$ ou AB) ;
- la disjonction *ou* notée $\vee$ (ou l'addition de propositions : $A + B$) ;
- la négation *non* notée $\neg A$ ou encore $\overline{A}$.

La conjonction de deux propositions est vraie si et seulement si les deux propositions sont vraies. Cela correspond au sens usuel du mot « et » en français. Par exemple la proposition « Vous avez votre stylo bleu *et* votre stylo vert. » est vraie, si et seulement si vous avez d'une part votre stylo bleu, et d'autre part votre stylo vert.

La disjonction de deux propositions est vraie dès lors que l'une des deux est vraie. Ainsi la proposition « Vous avez votre stylo bleu *ou* votre stylo vert. » est vraie si vous n'avez que votre stylo bleu, ou si vous n'avez que votre stylo vert, mais aussi, si vous avez les deux.

Il faut ici prendre le temps d'observer que la disjonction se distingue de l'alternative que la langue française peut faire porter à la conjonction de coordination « ou ». On considère les propositions suivantes :

- « Dessine les contours en bleu ou en vert. » ;
- « Nous pourrons prendre rendez-vous mardi ou mercredi. » ;
- « Fais-le ou ne le fais pas. »

Chacune présente une alternative. Les contours du dessin seront en bleu ou en vert, mais pas dans un mélange des deux. Le rendez-vous sera pris mardi ou mercredi, mais pas les deux jours, l'action sera faite ou ne sera pas faite... Au regard de l'algèbre de Boole l'opération algébrique booléenne en question est appelée « ou exclusif ». Elle est parfois notée $\veebar$ (ou encore $A \oplus B$).

En cherchant à définir une « algèbre de la logique », George Boole visait à substituer à l'argumentation verbale une approche systématique, exprimée dans la « langue abstraite des mathématiques ». Ces opérations avaient pour objet, dans l'esprit de George Boole et de Augustus De Morgan, d'exprimer des propositions logiques de manière purement symbolique, puis de les traiter de manière algébrique pour raisonner « mathématiquement ».

2.1.1 Tables de vérités

Les opérations booléennes sont des fonctions qui prennent en paramètres un ou plusieurs booléens et produisent en résultat un et un seul booléen. Comme ces fonctions sont appliquées à un ensemble de valeurs très limité (deux valeurs : $0, 1$), il est raisonnable de définir une fonction booléenne en énumérant simplement l'ensemble des résultats possibles. Cette énumération est présentée dans ce qui est appelée une *table de vérité*.

La fonction booléenne $\neg$ est définie, pour son paramètre x de type booléen, par la table de vérité suivante :

x	$\neg x$
0	1
1	0

Les fonctions booléennes $\wedge$ et $\vee$ sont complètement définies, pour leurs paramètres x et y de type booléen, par les tables de vérité suivantes :

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Les valeurs de vérité *vrai* (1) et *faux* (0) sont des symboles abstraits. Si les valeurs de vérités 1 et 0 sont interprétées comme des entiers, alors les opérateurs logiques peuvent être exprimées par des opérations arithmétiques :

- $x \wedge y = x \cdot y$;
- $x \vee y = x + y - x \cdot y$;
- $\neg x = 1 - x$

où $+$ correspond à l'addition, $-$ à la soustraction et $\cdot$ à la multiplication.

Les opérateurs de conjonction et de disjonction peuvent également être exprimées à l'aide des fonctions *minimum* et *maximum* :

- $x \wedge y = min(x, y)$;
- $x \vee y = max(x, y)$.

2.1.2 Lois

Un terme booléen est une expression construite à partir des variables booléennes et des constantes 0 et 1 en utilisant les opérations $\wedge$, $\vee$ et $\neg$. Dans l'algèbre de Boole, une loi est une égalité entre deux termes booléens. L'algèbre booléenne respecte les lois du tableau 2.1.

Loi	$\wedge$	$\vee$
Commutativité	$x \wedge y = y \wedge x$	$x \vee y = y \vee x$
Associativité	$x \wedge (y \wedge z) = (x \wedge y) \wedge z$	$x \vee (y \vee z) = (x \vee y) \vee z$
Distributivité de $\wedge$ sur $\vee$	$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
Élément neutre	$x \wedge 1 = x$	$x \vee 0 = x$
Élément absorbant	$x \wedge 0 = 0$	$x \vee 1 = 1$
Idempotence	$x \wedge x = x$	$x \vee x = x$
Complémentarité	$x \wedge \neg x = 0$	$x \vee \neg x = 1$
Double négation	$\neg\neg x = x$	
De Morgan	$\neg(x \wedge y) = \neg x \vee \neg y$	$\neg(x \vee y) = \neg x \wedge \neg y$

Tableau 2.1 – Lois de l'algèbre booléenne

À l'aide de ces lois, il devient possible de traiter les problèmes logiques comme l'on traite les problèmes numériques. C'était là le sens des travaux de Boole.

2.2 Fonctions booléennes

Comme il n'existe que deux valeurs booléennes, il est possible de définir complètement une fonction booléenne en énumérant pour chaque valeur possible de ses paramètres, la valeur renvoyée par la fonction.

Ainsi, les fonctions booléennes à une variable peuvent être définies en indiquant pour chaque valeur possible du paramètre la valeur renvoyée par la fonction, elle aussi, booléenne. Les valeurs possibles du paramètre sont au nombre de deux : 0 ou 1. Pour définir une fonction booléenne à un paramètre, il faut définir la valeur renvoyée lorsque le paramètre vaut 0 et la valeur renvoyée lorsque le paramètre vaut 1. Pour ces deux valeurs possibles du paramètre, il existe deux valeurs booléennes possibles du résultat. Il existe donc $2 \times 2 = 4$ fonctions booléennes à un paramètre, énumérées dans le tableau 2.2, par leur table de vérité.

- $fb_{0,1}(x)$, qui renvoie *faux* quel que soit x, est appelée *contradiction*.
- $fb_{1,1}(x)$, qui renvoie la valeur booléenne différente de x, est appelée *négation*.
- $fb_{2,1}(x)$, qui renvoie la valeur de x, est appelée *identité*.
- $fb_{3,1}(x)$, qui renvoie *vrai* quel que soit x, est appelée *tautologie*.

x	$fb_{0,1}(x)$ $= 0$
0	0
1	0

x	$fb_{1,1}(x)$ $= \neg x$
0	1
1	0

x	$fb_{2,1}(x)$ $= x$
0	0
1	1

x	$fb_{3,1}(x)$ $= 1$
0	1
1	1

TABLEAU 2.2 – Fonctions booléennes à un paramètre

Suivant le même schéma de construction, les fonctions booléennes à deux paramètres peuvent être énumérées. Ces fonctions notées $fb_{n,2}(x, y)$ seront associées à des tables de vérité listant les combinaisons de valeurs possibles. Puisqu'il y a deux valeurs possibles pour chacun des paramètres, il y a $2^2 = 4$ combinaisons d'arguments possibles. Puisqu'il y a 2 valeurs de résultats possibles pour chaque combinaison d'argument, il existe $2^{2^2} = 16$ fonctions distinctes.

De la même manière, avec trois paramètres, il existe $2^{2^3} = 256$ fonctions booléennes distinctes, et plus généralement, il existe extactement 2^{2^p} fonctions booléennes à p paramètres. Si le nombre de fonctions booléennes distinctes croit très rapidement en fonction du nombre de paramètres, il reste raisonnable d'énumérer les 16 fonctions à deux paramètres comme dans le tableau 2.3.

Au sein de cette énumération figurent les fonctions de l'algèbre de Boole : $\wedge$ et $\vee$.

$fb_{0,2}(x, y)$ renvoie toujours *faux* : c'est la fonction *contradiction* à deux paramètres.
$fb_{8,2}(x, y)$ est la fonction *et* notée $\wedge$.
$fb_{14,2}(x, y)$ est la fonction *ou* notée $\vee$.
$fb_{15,2}(x, y)$ renvoie toujours *vrai* : c'est la fonction *tautologie* à deux paramètres.

Parmi cette liste, la fonction $fb_{7,2}$ (notée par la suite $\uparrow$, et appelée `NAND` par l'industrie des semi-conducteurs) ainsi que la fonction $fb_{1,2}$ (notée par la suite $\downarrow$ et appelée `NOR`) sont dites universelles. Une fonction universelle est une fonction qui peut être utilisée, seule, pour exprimer toutes les autres fonctions booléennes possibles. Cette propriété établit donc que toutes les fonctions booléennes citées ci-dessus peuvent être construites utilisant uniquement des $\uparrow$ (ou encore, uniquement des $\downarrow$).

- La fonction *non*, notée $\neg x$ peut être construite comme suit.

  Par définition, on a : $x \uparrow x = \neg(x \wedge x)$. En appliquant l'idempotence, on obtient : $x \uparrow x = \neg x$. Donc, on a l'équation :

  $$\neg x = x \uparrow x \tag{2.1}$$

- La fonction *et*, notée $x \wedge y$ peut être construite comme suit.

  Par définition, on a : $\neg(x \wedge y) = x \uparrow y$, donc en appliquant la négation des deux côtés : $\neg\neg(x \wedge y) = \neg(x \uparrow y)$. En appliquant la double négation, on obtient : $x \wedge y = \neg(x \uparrow y)$. Or il a été précédement établi que : $\neg x = x \uparrow x$. Donc, on a l'équation :

  $$x \wedge y = (x \uparrow y) \uparrow (x \uparrow y) \tag{2.2}$$

x	y	$fb_{0,2}(x,y)$ $= 0$
0	0	0
0	1	0
1	0	0
1	1	0

x	y	$fb_{1,2}(x,y)$ $= x \downarrow y$ $= \neg(x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	0

x	y	$fb_{2,2}(x,y)$ $= \neg x \wedge y$
0	0	0
0	1	1
1	0	0
1	1	0

x	y	$fb_{3,2}(x,y)$ $= fb_{1,1}(x)$ $= \neg x$
0	0	1
0	1	1
1	0	0
1	1	0

x	y	$fb_{4,2}(x,y)$ $= x \wedge \neg y$
0	0	0
0	1	0
1	0	1
1	1	0

x	y	$fb_{5,2}(x,y)$ $= fb_{1,1}(y)$ $= \neg y$
0	0	1
0	1	0
1	0	1
1	1	0

x	y	$fb_{6,2}(x,y)$ $= x \veebar y$ $= (\neg x \vee y)$ $\wedge (x \vee \neg y)$
0	0	0
0	1	1
1	0	1
1	1	0

x	y	$fb_{7,2}(x,y)$ $= x \uparrow y$ $= \neg(x \wedge y)$
0	0	1
0	1	1
1	0	1
1	1	0

x	y	$fb_{8,2}(x,y)$ $= x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$fb_{9,2}(x,y)$ $= \neg x \veebar y$ $=$ $(\neg x \vee \neg y)$ $\wedge (x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	1

x	y	$fb_{10,2}(x,y)$ $= fb_{2,1}(y)$ $= y$
0	0	0
0	1	1
1	0	0
1	1	1

x	y	$fb_{11,2}(x,y)$ $= \neg x \vee y$
0	0	1
0	1	1
1	0	0
1	1	1

x	y	$fb_{12,2}(x,y)$ $= fb_{2,1}(x)$ $= x$
0	0	0
0	1	0
1	0	1
1	1	1

x	y	$fb_{13,2}(x,y)$ $= x \vee \neg y$
0	0	1
0	1	0
1	0	1
1	1	1

x	y	$fb_{14,2}(x,y)$ $= x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$fb_{15,2}(x,y)$ $= 1$
0	0	1
0	1	1
1	0	1
1	1	1

TABLEAU 2.3 – Fonctions booléennes à deux paramètres

- La fonction *ou*, notée $x \vee y$ peut être construite comme suit.

  En partant de la loi de De Morgan, on a : $\neg(x \vee y) = \neg x \wedge \neg y$. En appliquant la double négation, on obtient : $x \vee y = \neg(\neg x \wedge \neg y)$. Or $x \uparrow y = \neg(x \wedge y)$ et donc $x \vee y = \neg x \uparrow \neg y$. Or il a été établi précédemment que : $\neg x = x \uparrow x$. Donc, on a l'équation :

$$x \vee y = (x \uparrow x) \uparrow (y \uparrow y) \tag{2.3}$$

À l'aide des trois égalités (2.1), (2.2) et (2.3), il est donc possible de remplacer n'importe quelle expression booléenne composée de $\neg$, $\wedge$ et $\vee$ par une expression qui n'est composée que de $\uparrow$.

La manufacture des premiers circuits intégrés étant encore délicate, il était initialement plus aisé de produire en grand nombre des composants réalisant une seule et unique fonction logique. Ainsi l'ordinateur de navigation des missions Appolo (AGC) fut contruit exclusivement à partir de portes non-ou ($\downarrow$). Ces portes étaient réalisées par la gravure de huits transistors sur un circuit intégré. 4100 portes non-ou, soit 4100 circuits intégrés, furent interconnectées pour consistuer le microprocesseur de cet ordinateur embarqué ; un microprocesseur qui pesait 32 kg. Aujourd'hui les microprocesseurs mixent, au sein d'un même circuit intégré, toutes sortes de portes logiques constituées à partir de différents assemblages de transistors. La synthèse de circuits qui réalisent les fonctions logiques est confiée à des algorithmes qui automatisent ce processus de sélection des transistors et de leur placement sur le silicium.

2.3 Un problème de vérité

On considère le problème exposé dans la figure 2.1 sur le vol des bonbons.

2.3.1 Modélisation

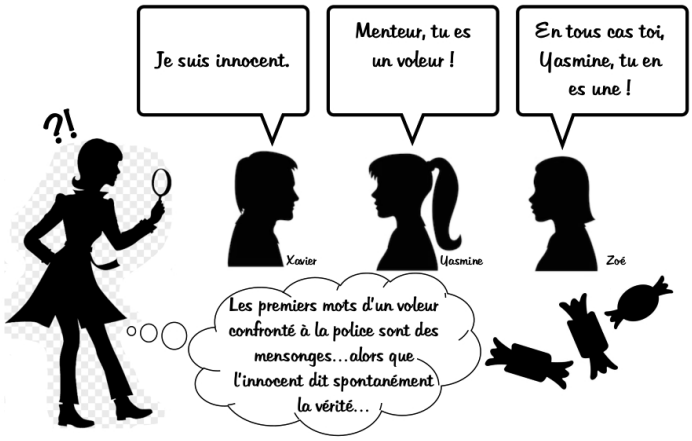
Cet énoncé peut être modéliser en utilisant l'algèbre de Boole.

- Soit x la valeur booléenne indiquant que Xavier est considéré innocent.
- Soit y la valeur booléenne indiquant que Yasmine est considérée innocente.
- Soit z la valeur booléenne indiquant que Zoé est considérée innocente.
- Soit p_x la proposition formulée par Xavier : $p_x = x$. La proposition p_x est vraie quand x est vrai.
- Soit p_y la proposition formulée par Yasmine : $p_y = \neg x$. La proposition p_y est vraie quand x est fausse.
- Soit p_z la proposition formulée par Zoé : $p_z = \neg y$. La proposition p_z est vraie quand y est fausse.

2.3.2 Résolution utilisant l'algèbre de Boole

Pour résoudre ce problème on peut utiliser un raisonnement par l'absurde, en prenant pour hypothèse que l'un des trois (Xavier, Yasmine ou Zoé) est innocent et en vérifiant que l'inspectrice n'y voit aucune contradiction.

Vol des bonbons



Un marchand s'est fait voler des bonbons dans sa bonbonnière. À l'aide des empreintes digitales présentes sur la boîte, l'inspectrice de police a pu identifier trois suspects qui ont été amenés au poste de police. Le marchand déclare qu'une seule personne a payé pour des bonbons, malheureusement il ne se souvient plus de son visage.

- Xavier, le premier suspect, déclare : « Je suis innocent. »
- Yasmine, la seconde suspecte, déclare : « Menteur, tu es un voleur ! »
- Zoé, la troisième suspecte, déclare : « En tout cas toi, Yasmine, tu en es une ! »

Selon son expérience, l'inspectrice considère que les premiers mots d'un voleur confronté à la police sont des mensonges alors que l'innocent dit spontanément la vérité.
Qui l'inspectrice considère-t-elle comme innocent ?

FIGURE 2.1 – Problème de vérité

Hypothèse 1 : Xavier, innocent, dit la vérité. Selon l'hypothèse 1, Xavier est innocent. Aux yeux de l'inspectrice, ceci implique qu'il dit la vérité et que les autres, qui ne sont pas innocents, mentent. On a :

$$x = 1,\ y = 0,\ z = 0 \quad \text{et} \quad h_1 = p_x \wedge \neg p_y \wedge \neg p_z$$
$$h_1 = x \wedge \neg\neg x \wedge \neg\neg y$$
en appliquant la double négation $$h_1 = x \wedge x \wedge y$$
en appliquant l'idempotence $$h_1 = x \wedge y$$
finalement $$h_1 = 1 \wedge 0$$
donc $$h_1 = 0.$$

Cette hypothèse soulève une contradiction aux yeux de l'inspectrice. L'hypothèse 1, qui suggère que Xavier est innocent, est fausse, si les voleurs mentent.

Hypothèse 2 : Yasmine, innocente, dit la vérité. Selon l'hyptohèse 2, Yasmine est innocente. Aux yeux de l'inspectrice, elle dit la vérité et les autres mentent. On a :

$$x = 0,\ y = 1,\ z = 0 \quad \text{et} \quad h_2 = \neg p_x \wedge p_y \wedge \neg p_z$$
$$h_2 = \neg x \wedge \neg x \wedge \neg\neg y$$
en appliquant la double négation $$h_2 = \neg x \wedge \neg x \wedge y$$
en appliquant l'idempotence $$h_2 = \neg x \wedge y$$
finalement $$h_2 = \neg 0 \wedge 1 = 1 \wedge 1$$
donc $$h_2 = 1.$$

Aux yeux de l'inspectrice, l'hypothèse 2, qui suggère que Yasmine est innocente, peut être considérée comme vraie.

Hypothèse 3 : Zoé, innocente, dit la vérité. Selon l'hypothèse 3, Zoé est innocente. Pour l'inspectrice, elle dit la vérité et les autres mentent. On a :

$$x = 0,\ y = 0,\ z = 1 \quad \text{et} \quad h_3 = \neg p_x \wedge \neg p_y \wedge p_z$$
$$h_3 = \neg x \wedge \neg\neg x \wedge \neg y$$
en appliquant la double négation $$h_3 = \neg x \wedge x \wedge \neg y$$
en appliquant la complémentarité $$h_3 = 0 \wedge \neg y$$
en appliquant l'élément absorbant $$h_3 = 0.$$

L'hypothèse 3, qui suggère que Zoé est innocente, est considérée comme fausse par l'inspectrice, sans même qu'il y ait besoin d'instancier les valeurs de x, y et z.

Conclusion : L'hypothèse 2 est la seule valide pour l'inspectrice. Selon cette hypothèse, Yasmine est innocente, alors que Xavier et Zoé sont les voleurs de bonbons.

2.3.3 Résolution utilisant la programmation fonctionnelle

Le langage de programmation `OCaml` est un langage de programmation dit « fonctionnel » dont les fondements seront abondemment étudiés dans cet ouvrage. Pour les langages de programmation fonctionnels, tout est *fonction* (au sens mathématique du terme). Il est donc naturel d'exprimer les propositions précédentes sous forme de fonctions et de les évaluer avec différents paramètres pour *calculer* la valeur de vérité de chaque hypothèse.

Comme le langage `Python` ou `Javascript`, le langage `OCaml` dispose d'un interpréteur en ligne qui lit puis évalue chaque « déclaration fonctionnelle » que l'on saisit. Pour utiliser cet interprète de commande il suffit d'exécuter le programme `ocaml` :

```
$ ocaml
OCaml version 4.14.0
Enter #help;; for help.
#
```

L'invite `#` du programme `ocaml` attend alors que soit saisie une déclaration pour l'évaluer. Une déclaration se termine par `;;`.

Selon la modélisation en algèbre de Boole faite précédemment, le problème du vol des bonbons dépend de trois valeurs booléennes, x, y et z. Ces trois valeurs booléennes sont utilisées par les propositions de Xavier, de Yasmine et de Zoé (respectivement p_x, p_y et p_z). Dans une approche fonctionnelle, ces trois propositions correspondent à trois fonctions qui prennent, chacune, en paramètres les trois valeurs booléennes.

Ainsi, la proposition de Xavier p_x est ici une fonction `px` qui prend trois paramètres, `x`, `y` et `z` et qui renvoie `x`. En `OCaml` on peut déclarer cette fonction comme suit :

```
# let px x y z = x ;;
```

L'interpréteur répond alors la ligne suivante :

```
val px : 'a -> 'b -> 'c -> 'a = <fun>
```

Ce faisant il indique simplement qu'il prend acte de l'existence d'un symbole `px` qui est une fonction (`= <fun>`). Cette fonction prend un premier paramètre du type `'a`, suivi d'un second paramètre du type `'b` et d'un troisième paramètre du type `'c`. Cette fonction renvoie un résultat du type `'a`. L'interpréteur `ocaml` n'impose pas un type pour ces paramètres. Ils peuvent être les mêmes ou tous différents. L'interprète a seulement déterminé que le type de retour serait nécessairement le même que le type du premier paramètre.

On déclare la proposition de Yasmine par :

```
# let py x y z = not x ;;
```

Ici la seule différence est la présence du `py` et du `not`. Il s'agit de la déclaration de la proposition p_y qui renvoie $\neg x$, exprimé dans le langage `OCaml` par `not x`. Pourtant la réponse de l'interprète est substentiellement différente :

```
val py : bool -> 'a -> 'b -> bool = <fun>
```

Le premier paramètre n'est plus d'un type indéterminé `'a`, mais du type `bool`. `OCaml` a automatiquement déterminé que le premier paramètre devait *nécessairement* être un booléen. De même, il a déterminé que le résultat de la fonction `py` serait un booléen. Les deux autres paramètres, inutilisés dans la proposition, sont de types indéterminés `'a` et `'b`.

On déclare la proposition de Zoé par :

```
# let pz x y z = not y ;;
```

L'interpréteur déclare, une fois encore, reconnaître l'existence d'une fonction associée au symbole `pz`.

```
val pz : 'a -> bool -> 'b -> bool = <fun>
```

Cette fonction `pz` doit nécessairement recevoir un booléen (type `bool`) en second paramètre et elle renvoie nécessairement un booléen.

Il est maintenant possible de déclarer les trois hypothèses sous la forme de trois fonctions (`h1`, `h2` et `h3`), paramètrées par `x`, `y` et `z`. Pour cela il faut savoir que le langage `OCaml` utilise l'opérateur `&&` pour représenter l'opération logique $\wedge$ (et) et qu'il utilise l'opérateur `||` pour représenter l'opération logique $\vee$ (ou).

```
# let h1 x y z = (px x y z) && not (py x y z) && not (pz x y z) ;;
```

L'interpréteur déclare alors reconnaître l'existence du symbole `h1` qui est une fonction.

```
val h1 : bool -> bool -> 'a -> bool = <fun>
```

Là encore, l'interpréteur `ocaml` a déterminé que le type des deux premiers paramètres ainsi que le type de retour de la fonction est nécessairement `bool`. En revanche, le type du troisième paramètre n'est pas contraint. De même on peut déclarer deux fonctions pour tester les hypothèses 2 et 3.

```
# let h2 x y z = not (px x y z) && (py x y z) && not (pz x y z) ;;
val h2 : bool -> bool -> 'a -> bool = <fun>
# let h3 x y z = not (px x y z) && not (py x y z) && (pz x y z) ;;
val h3 : bool -> bool -> 'a -> bool = <fun>
```

Il est alors possible de calculer la valeur logique des différentes hypothèses en donnant les valeurs de vérité associées à chacune d'elles, comme l'on évaluerait n'importe quelle fonction ou expression :

```
# h1 true false false ;;
```

L'interpréteur indique alors :

```
- : bool = false
```

L'évaluation de cette expression est donc du type `bool` et sa valeur est `false`. L'interpréteur `ocaml` vient de calculer que la première hypothèse est fausse. De même pour calculer la proposition logique associée à l'hypothèse 2, il suffit de déclarer :

```
# h2 false true false ;;
```

L'interpréteur répond :

```
- : bool = true
```

Ici, l'évaluation de l'hypothèse 2 a comme valeur de résultat le booléen `true`. Enfin, la troisième hypothèse s'évalue ainsi :

```
# h3 false false true ;;
```

Et l'interpréteur répond :

```
- : bool = false
```

L'évaluation de l'hypothèse 3 a comme valeur de résultat `false`. Comme dans la section précédente, l'inspectrice retient donc que seule l'hypothèse 2 est valide c'est-à-dire que c'est Yasmine qui est innocente, Xavier et Zoé sont coupables.

2.3.4 Résolution utilisant les portes logiques

Il est possible d'implanter les fonctions booléennes avec des circuits logiques. Il est aussi possible de calculer la valeur de vérité de chaque hypothèse du problème à l'aide d'un simple circuit. On utilise ici les circuits logiques *and*, *or* et *not* pour mettre en œuvre, dans un premier temps, les propositions logiques p_x, p_y et p_z. Ces propositions correspondent respectivement aux affirmations de Xavier, Yasmine et Zoé, en fonction de x, y et z qui sont les hypothèses de culpabilité de chacun. Sur cette base, il devient possible de réaliser le calcul des hypothèses h_1, h_2 et h_3.

Les symboles utilisés pour représenter les fonctions logiques sur le schéma d'un circuit sont donnés dans la figure 2.2.

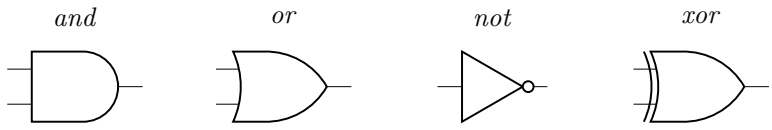
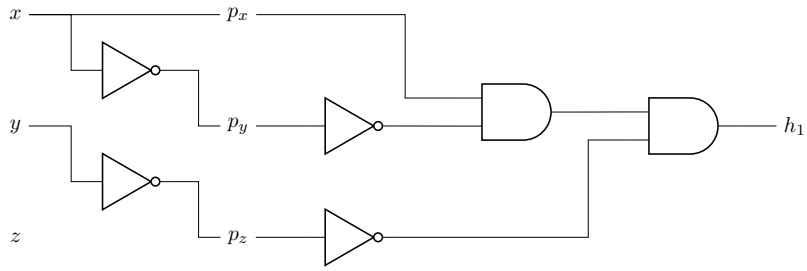
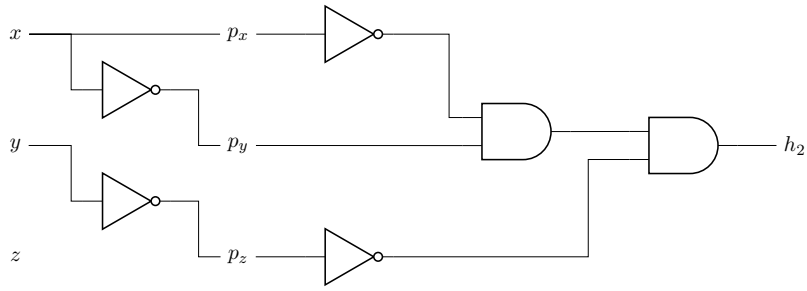


FIGURE 2.2 – Portes logiques

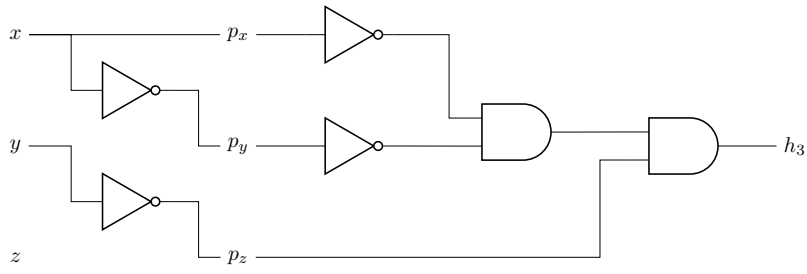
À l'aide de ces portes élémentaires, comme l'a démontré Shannon [65], il est possible de calculer n'importe quelle fonction booléenne élémentaire. Il est donc notamment possible de calculer h_1, comme suit :



En plaçant une tension sur x, mais pas sur y ni sur z on observe que la sortie h_1 ne produit pas de tension. L'hypothèse 1 n'est pas validée si Xavier est innocent, et que Yasmine et Zoé ne le sont pas. De même, il est possible de réaliser un circuit pour « calculer » la valeur logique de l'hypothèse 2 :



Sur ce second schéma, en plaçant une tension sur y, mais pas sur x ni sur z, on observe que la sortie h_2 présente une tension. L'hypothèse 2 est validée lorsque Yasmine est innocente, et que Xavier et Zoé ne le sont pas. Enfin, dans le même esprit il est possible de réaliser un circuit qui calcule la véracité de la dernière hypothèse.



En plaçant une tension sur z, mais pas sur x ni sur y on observe que la sortie h_3 ne produit pas de tension. L'hypothèse 3 n'est pas validée si Zoé est innocente, et que Xavier et Yasmine ne le sont pas. Comme dans les résolutions précédentes, l'inspectrice retient donc que seule l'hypothèse 2 est valide.

2.4 De l'algèbre de Boole au calcul binaire

Au-delà du calcul propositionnel, l'algèbre de Boole est aussi le fondement de la mise en œuvre de l'arithmétique binaire, telle que la pratique l'informatique contemporaine. Un booléen, vu du point de vue informatique, peut être distingué de la notion mathématique d'algèbre de Boole par son statut de *type de données*, c'est-à-dire d'objet susceptible d'être calculé, stocké et utilisé dans la dynamique d'un programme [2]. Certains langages de programmation disposent d'un type explicite pour représenter cette structure (`OCaml`, `Python`, `C++`, `Java`, etc.) et d'autres utilisent d'autres types pour la représenter (versions anciennes du `C`, shell Unix, etc.). Quoi qu'il en soit, la notion de booléen est toujours présente d'une façon ou d'une autre dans tous les langages de programmation.

Un nombre binaire de n bits est un nombre dont les n chiffres sont autant de booléens. Et sur ces nombres binaires, les opérations arithmétiques et logiques mettent en œuvre des fonctions booléennes. On considère l'exemple de l'addition binaire de deux nombres a et b, chacun représenté sur n bits.

La figure 2.3 présente l'addition binaire : $0011 + 0101$. La table de vérité 2.4 présente la fonction à opérer sur chaque bit.

```
    (0)(1)(1)    <- retenues
     0  0  1  1
+    0  1  0  1
--------------
     1  0  0  0
```

FIGURE 2.3 – Exemple d'addition binaire

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

TABLEAU 2.4 – Table de vérité de l'addition.

Cette fonction à deux paramètres est $f_{6,2}$, aussi connue sous le nom de *ou-exclusif*, notée $\veebar$. Cependant comme le montre l'exemple de l'addition précédente, il ne faut pas additionner deux variables binaires, mais 3, avec la retenue, pour calculer 1 bit du résultat.

Pour réaliser un circuit capable de faire une addition binaire, il faut réaliser un circuit logique qui prend en entrée le $i^{\text{ème}}$ bit (un booléen) du premier nombre noté a_i et le $i^{\text{ème}}$ bit (un autre booléen) du deuxième nombre noté b_i ainsi que la retenue, notée c_{i-1}, du résultat de l'addition des bits précédents de a et b.

Ce circuit logique doit produire deux chiffres booléens :

- r_i le bit (booléen) résultant de l'addition des bits, $r_i = a_i + b_i + c_{i-1}$;
- c_i le bit (booléen) qui donne la retenue de l'addition des chiffres

où $i \in [\![0, n-1]\!]$ et $c_{-1} = 0$ par convention.

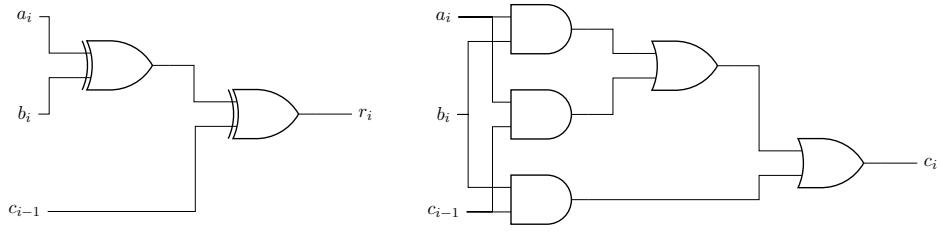
On peut construire la table de vérité de r_i et c_i.

a_i	b_i	c_{i-1}	$r_i(a_i, b_i, c_{i-1})$	$c_i(a_i, b_i, c_{i-1})$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

En observant ces tables, on peut proposer les fonctions booléennes suivantes :

- $r_i = a_i \veebar b_i \veebar c_{i-1}$;
- $c_i = (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$.

Ces fonctions peuvent être implantées par les circuits logiques suivants.



Puis ces circuits peuvent être branchés les uns aux autres, en cascade, pour former un circuit capable de réaliser des additions de n bits (n étant fixé à la réalisation du circuit). Pour $n = 4$, on a le circuit de la figure 2.4. La somme de a et b vaut le nombre constitué de 4 bits : $r_0 r_1 r_2 r_3$.

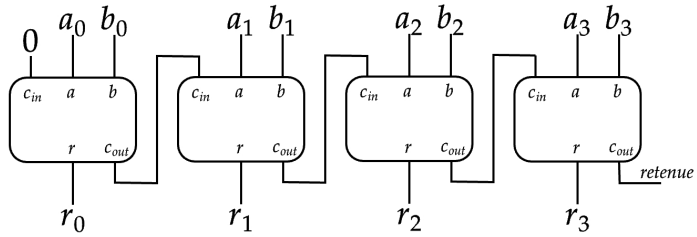


FIGURE 2.4 – Additionneur 4 bits

Dans le même esprit il est possible de réaliser un circuit pour faire une soustraction, une multiplication, ou toute autre opération arithmétique souhaitable. Au sein des microprocesseurs des ordinateurs actuels, ces circuits implémentent les opérations arithmétiques et logiques, dites élémentaires. Et lorsqu'on écrit des programmes, ce sont *in fine* ces circuits booléens qui réalisent les calculs demandés.

2.5 TD/TP - Booléens

Exercice 7 : Juge ⇝ *p.565*

Un juge veut libérer de la place dans ses prisons en donnant une chance de sortir aux prisonniers qui sont les plus à même de réfléchir sur leur forfait. Il présente un prisonnier devant deux portes et déclare « Voici deux portes. Chacune porte une indication. » Le prisonnier le constate en effet :

- sur la première porte, il est indiqué « Cette porte conduit à la liberté, l'autre à ta cellule. » ;
- sur la seconde porte, il est indiqué « L'une des portes conduit à la liberté, l'autre à ta cellule. »

Le prisonnier demande « Est-ce que c'est vrai ? » Alors le juge, qui veut privilégier les prisonniers ayant un jugement fiable, ajoute : « L'une des indications est vraie, l'autre est fausse. Sauras-tu reconnaître celle qui dit vrai ? Montre-le moi en choisissant la bonne porte. »

1) On modélise ce problème en utilisant l'algèbre de Boole. On prendra deux valeurs booléennes l_1 et l_2, qui indiquent respectivement que la porte 1 et la porte 2 conduisent à la liberté. Les indications sur les portes peuvent également être modélisés comme deux propositions logiques : p_1 et p_2. Écrire les expressions booléennes de ces deux propositions.

2) Pour modéliser l'affirmation du juge c'est-à-dire « l'une des indications est vraie, l'autre est fausse », on utilisera deux nouvelles propositions logiques p_3 et p_4. Écrire les expressions booléennes de ces deux propositions.

3) Simplifier la proposition p_3 en utilisant les lois de la logique propositionnelle.

4) Simplifier la proposition p_4 en utilisant les lois de la logique propositionnelle.

5) Construire la table de vérité correspondant aux propositions p_3 et p_4.

6) Déterminer la porte que le prisonnier doit choisir pour atteindre la liberté. Justifier.

7) Vérifier la table de vérité des propositions p_3 et p_4 en utilisant `OCaml` en mode interpréteur. Le code doit être écrit dans un fichier nommé `Nom_Booleens_Juge.ml` dont l'exécution est rappelée ci-dessous :

```
# #use "Nom_Booleens_Juge.ml";;
```

8) Créer le circuit correspondant aux propositions p_3 et p_4.

Exercice 8 : Jeu télévisé ⇝ *p.566*

La présentatrice d'un jeu télévisé présente deux boîtes à un participant et explique : « Dans chaque boîte vous trouvez, soit un gain de 1 000 €, soit une perte de gains accumulés jusque-là, mais rien d'autre. *A priori*, le contenu de chaque boîte est indépendant de ce que contient l'autre. Mais des indications sur les boîtes vous en diront davantage. »

- Sur la boîte bleue, il est indiqué : « La boîte bleue vous fait perdre vos gains. »
- Sur la boîte rouge, il est indiqué : « Au moins une des boîtes contient un gain de 1 000 €. »

La présentatrice précise : « Soit les deux indications sont correctes, soit elles sont toutes les deux fausses. » Puis elle propose au participant d'ouvrir la boîte de son choix, les deux, ou aucune.

1) Proposer une modélisation à l'aide de l'algèbre de Boole au problème posé.

2) En simplifiant les propositions à l'aide de l'algèbre de Boole, expliquer quelle devrait être la stratégie gagnante du joueur.

3) Faire la table de vérité correspondant aux propositions.

4) Calculer la valeur de vérité de la modélisation en utilisant `OCaml`.

5) Créer le circuit logique correspondant à la modélisation.

Exercice 9 : Jeu télévisé - suite ⇝ *p.567*

La présentatrice du jeu télévisé propose toujours une boîte rouge et une boîte bleue, portant chacune la même indication : « Les deux boîtes sont gagnantes. » Cependant, cette fois-ci les règles sont différentes.

- Pour la boîte bleue, l'indication qu'elle porte :
  - est fausse si la boîte contient un gain de 1 000 € ;
  - est vraie si la boîte contient une perte des gains accumulés jusque-là.
- À l'inverse, pour la boîte rouge, l'indication qu'elle porte :
  - est vraie si la boîte contient un gain de 1 000 € ;
  - est fausse si la boîte contient une perte des gains accumulés jusque-là.

Dire ce que doit jouer le participant. Justifier ce choix avec une méthode choisie après avoir modélisé le problème à l'aide de l'algèbre de Boole.

Exercice 10 : ★ Complément à deux ⇝ *p.568*

L'additionneur binaire permet l'addition de deux nombres binaires représentés sous la forme d'une série de booléens allant de a_0 à a_{n-1} et de b_0 à b_{n-1}. Il produit une série de booléens résultants notés de r_0 à r_{n-1} comme cela a été étudié précédemment. On s'intéresse ici à la synthèse d'un circuit qui réalise le complément à deux d'un nombre. Le complément à deux d'un nombre binaire est un codage astucieux de l'information de telle sorte que les additionneurs binaires fonctionnent aussi bien avec les nombres positifs qu'avec les nombres négatifs (voir la section 4.2.1 Représentation des nombres entiers, page 92). En ce qui concerne cet exercice, il faut juste savoir que selon cette représentation, le nombre opposé $-a$ d'un nombre a positif de n bits est obtenu en prenant l'opposé de chaque bit de a puis en ajoutant `1` au nombre résultant de cette inversion. C'est cette fonction booléenne qu'il s'agit de concevoir ici.

On note les bits du nombre binaire a codé sur n bits de a_0 à a_{n-1} et le nombre produit $r = -a$ décomposé lui aussi en n bits de r_0 à r_{n-1}. Le bit de poids fort de cette représentation en complément à deux (c'est-à-dire a_{n-1} et r_{n-1}) indique qu'un nombre est positif s'il vaut `0` et négatif s'il vaut `1`. On note aussi c_i la retenue qu'il convient de propager du $i^{\text{ième}}$ bit au $(i+1)^{\text{ième}}$ bit.

1) Donner la formule booléenne qui permet de calculer r_0 en fonction de a_0.

2) Donner la formule booléenne qui donne c_0 en fonction de a_0.

3) Donner la formule booléenne qui permet de calculer r_i en fonction de a_i et de c_{i-1}, pour $i \in [\![1, n-1]\!]$.

4) Donner la formule booléenne qui permet de calculer c_i en fonction de a_i et de c_{i-1}, pour $i \in [\![1, n-1]\!]$.

5) Donner le circuit logique associé à la formule booléenne de r_0.

6) Donner le circuit logique associé à la formule booléenne de c_0.

7) Donner le circuit logique associé à la formule booléenne de r_i, $\forall i \in [\![1, n-1]\!]$.

8) Donner le circuit logique associé à la formule booléenne de c_i, $\forall i \in [\![1, n-1]\!]$.

9) Démontrer qu'il n'y a qu'un nombre positif a composé de n bits dont l'opposé en complément à deux est lui aussi positif.

Indication : Sachant que $a_{n-1} = 0$, déterminer dans quel cas $r_{n-1} = 0$.

10) Démontrer qu'il n'y a qu'un nombre négatif a composé de n bits dont l'opposé en complement à deux est lui aussi négatif.

Indication : Sachant que $a_{n-1} = 1$, déterminer dans quel cas on a $r_{n-1} = 1$.

Chapitre 3
Programmation

Programmer consiste avant tout à concevoir une série d'opérations qui paramètrent le comportement d'une machine pour produire un résultat. Les premières machines capables de produire un résultat « mécaniquement » datent de l'Antiquité. Ainsi la machine d'Anticythère détermine la position de différents astres au cours du temps à partir d'un état initial. Au IX^e^ siècle, des automates comme l'horloge astronomique de Su Song ou les séquenceurs musicaux des frères Banu Musa enchaînent l'activation de différents mécanismes, tandis que la Pascaline de Blaise Pascal, au XVII^e^ siècle, réalise des calculs arithmétiques à partir de données initiales. Au XIX^e^ siècle, un mécanisme conçu par J.M. Jacquard constitue une avancée majeure, en permettant, par le biais de cartes perforées, de reconfigurer la trame d'un métier à tisser automatiquement au fur et à mesure de l'ouvrage. Le résultat dépend alors de l'enchaînement des configurations de la machine, définies par la carte.

Le premier véritable acte de programmation apparaît en 1843 sous la plume d'Ada Lovelace. Elle écrit divers programmes destinés à être exécutés par la machine analytique de C. Babbage. Pour A. Lovelace, concevoir un programme consiste à perforer des cartes comparables à celles de J.M. Jacquard. Ces perforations entraînent des actions mécaniques qui, plutôt que de lier des fils, opèrent des calculs numériques comme le faisait la Pascaline. Bien que les traitements soient numériques, A. Lovelace perçoit alors immédiatement qu'une machine programmable peut également traiter les symboles de toute nature.

> *« L'ancienne machine est de nature strictement arithmétique, et les résultats qu'elle peut obtenir se situent dans un domaine très clairement défini et restreint, tandis qu'il n'y a pas de ligne de démarcation finie qui limite les pouvoirs de la Machine Analytique. Ces pouvoirs sont coextensifs à notre connaissance des lois de l'analyse elle-même, et ne doivent être bornés que par notre compréhension de ces dernières. En effet, nous pouvons considérer la machine comme le représentant matériel et mécanique de l'analyse, et que nos capacités de travail actuelles dans ce domaine d'étude humaine seront plus efficacement en mesure de suivre le rythme de notre connaissance théorique de ses principes et lois, grâce au contrôle complet que la machine nous donne sur la manipulation exécutive des symboles algébriques et numériques. »*
>
> Ada Lovelace, *Sketch of the analytical engine invented by C. Babbage*, [55], page 696

Avec les premiers ordinateurs à tubes à vide dans les années 1940, tels que l'*ENIAC*, la programmation nécessite une configuration manuelle de câblages et commutateurs pour réaliser des opérations logiques et arithmétiques. Chaque programme est intégré physiquement à la machine. Des améliorations permettent cependant à l'*ENIAC* d'exécuter des « commandes stockées ». Cela marque un tournant : le programme devient un état évolutif de la mémoire interne, indépendant de l'activité matérielle. La programmation se transforme alors en interaction avec la machine. Ce changement conduit à l'essor des systèmes d'exploitation, chargés de gérer les programmes, comme d'autres gèrent les données, ainsi que des interpréteurs et compilateurs, qui traduisent les instructions humaines en code machine.

3.1 Architecture de von Neumann

John von Neumann, mathématicien et physicien américain, a élaboré en juin 1945 la première description d'un ordinateur dont le programme est stocké dans sa mémoire de travail (voir [74] et [59]). Son modèle d'un ordinateur, représenté sur la figure 3.1, est composé de trois unités logiques :

- un (ou des) microprocesseur(s) ;
- une mémoire de travail ;
- un (ou des) périphérique(s) d'entrée-sortie.

La mémoire de travail est utilisée pour conserver les informations (numériques) qui correspondent à la fois aux instructions des programmes et aux données consultées ou produites par leur exécution. Le microprocesseur est composé d'une unité de contrôle qui est chargée du séquençage des instructions et d'une unité arithmétique et logique qui effectue les traitements de base. Les dispositifs d'entrée-sortie permettent de communiquer avec le monde extérieur (connecteur usb, support graphique, carte réseau, *etc.*).

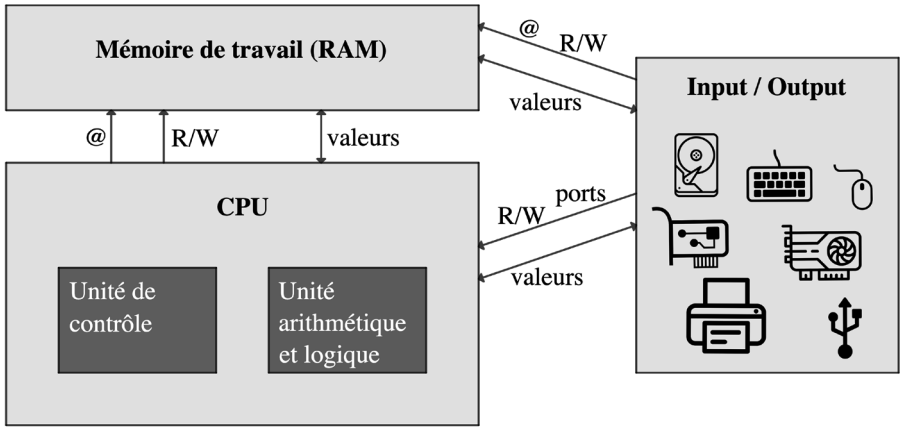


Figure 3.1 – Architecture de von Neumann

Dans le modèle d'architecture d'ordinateur de von Neumann, toujours pertinent en 2024, nonobstant les considérables progrès de la micro-électronique, le microprocesseur communique avec la mémoire *via* des adresses. Lorsque le microprocesseur requiert le contenu d'une information en mémoire, il fournit l'adresse de la case mémoire pour laquelle il sollicite l'écriture ou la lecture d'une valeur (une information numérique). Toutefois, les informaticiens ne se réfèrent plus directement aux adresses mémoire pour désigner les données à traiter. Ils se réfèrent plutôt à des « noms de variables » qui sont autant d'étiquettes associées à ces adresses. Par exemple, dans le cas d'un programme exprimé dans le langage `C`, c'est un autre programme appelé compilateur `C` qui exécute un algorithme pour déterminer, pour chaque nom

de variable, une adresse en mémoire qui sera utilisée pour mémoriser l'information associée. Les adresses ainsi retenues peuvent être absolues ou relatives à d'autres éléments du microprocesseur, et lorsque le système d'exploitation charge en mémoire le programme (dont l'utilisateur demande l'exécution) il arrive qu'il modifie les choix d'adresses que le compilateur avait établi. Le détail des traitements mises en œuvre par le compilateur et le système d'exploitation sort du cadre de cet ouvrage. Néanmoins, le langage `C` permet de manipuler les adresses effectives finalement associées par ces programmes aux variables.

3.2 Du langage source au langage machine

Alors qu'avec les larges modèles de langages, l'intelligence artificielle envahit notre quotidien, il est important de garder à l'esprit que les ordinateurs ne comprennent pas notre langage, que ce soit le français, l'anglais, que ce soit un paragraphe de texte ou simplement une succession de mots-clés. Une machine ne sait exécuter qu'une seule sorte de programmes, ceux exprimés dans le *langage machine*. À l'époque des premiers ordinateurs, les programmeurs tels qu'Ada Lovelace programmaient directement dans le langage de leur machine. Concevoir un programme s'avérait alors très long et fastidieux. Avec le développement de l'informatique, les programmes ont été saisis dans des fichiers informatiques, exprimés dans des langages moins hermétiques. Dans un premier temps, des mots-clés furent associés aux opérations des langages machine. Le langage, formé par ces mots-clés lisibles par un être humain, mais en bijection stricte avec les opérations élémentaires de la machine est appelé le *langage assembleur*. Un programme lui aussi nommé *assembleur* permet de transformer ce fichier appelé *fichier source* ou *code source* dans un autre fichier appelé *fichier exécutable* (ou simplement *exécutable*). C'est au sein de ce dernier, et non dans le fichier source, qu'est exprimé le programme dans le langage de la machine. Néanmoins, même ce langage, qui n'est qu'une représentation textuelle en bijection avec les opérations élémentaires reconnues par un microprocesseur, reste difficile à utiliser pour concevoir des programmes. Grace Hopper (1906 - 1992) défendait l'idée qu'un programme devrait pouvoir être écrit dans un langage proche de l'anglais plutôt que d'être calqué sur le langage de la machine, comme l'assembleur. En 1951, elle a réalisé le premier *compilateur*. Un *compilateur* est un programme qui traduit un programme source (généralement enregistré dans un fichier source), exprimé dans un langage intelligible, vers le langage de la machine, produisant ainsi un programme véritablement exécutable. Ce ne sont plus les programmes des informaticiens qui sont exécutés par les machines depuis bien longtemps. Ce sont leur traduction par un programme tiers, vers le langage de la machine.

3.2.1 Exemple introductif

On considère l'exemple d'un programme écrit en `C` et qui utilise une fonction nommée `addition` pour calculer et afficher la somme de deux entiers donnés en paramètres. Le fichier source donné dans le code 3.1 contient des mots-clés en anglais relativement intelligibles. Un compilateur du langage `C` permet de produire à partir de ce fichier source un fichier exécutable, nommé ici `addition`. La commande `objdump -d addition`, permet d'afficher le contenu du programme réellement exécutable sur la machine (ici un programme pour microprocesseur de la famille ARM). En pratique ce programme est simplement stocké dans le fichier sous la forme d'une série d'octets que le programme `objdump` présente sous la forme d'un texte.

```c
#include <stdio.h>

/* additionner deux entiers */
int addition(int nbr_1,int nbr_2){
  return nbr_1 + nbr_2;
}

int main(int argc,char **argv){
  printf("%d\n", addition(2,3));
  return 0;
}
```

Code 3.1 – [C] Exemple d'un programme

```
addition:       file format mach-o arm64

Disassembly of section __TEXT,__text:

0000000100003f60 <_addition>:
100003f60: ff 43 00 d1  sub     sp, sp, #16
100003f64: e0 0f 00 b9  str     w0, [sp, #12]
100003f68: e1 0b 00 b9  str     w1, [sp, #8]
100003f6c: e8 0f 40 b9  ldr     w8, [sp, #12]
100003f70: e9 0b 40 b9  ldr     w9, [sp, #8]
100003f74: 00 01 09 0b  add     w0, w8, w9
100003f78: ff 43 00 91  add     sp, sp, #16
100003f7c: c0 03 5f d6  ret

0000000100003f80 <_main>:
100003f80: ff 83 00 d1  sub     sp, sp, #32
100003f84: fd 7b 01 a9  stp     x29, x30, [sp, #16]
100003f88: fd 43 00 91  add     x29, sp, #16
100003f8c: bf c3 1f b8  stur    wzr, [x29, #-4]
100003f90: e0 0b 00 b9  str     w0, [sp, #8]
100003f94: e1 03 00 f9  str     x1, [sp]
100003f98: 40 00 80 52  mov     w0, #2
100003f9c: 61 00 80 52  mov     w1, #3
100003fa0: f0 ff ff 97  bl      0x100003f60 <_addition>
100003fa4: fd 7b 41 a9  ldp     x29, x30, [sp, #16]
100003fa8: ff 83 00 91  add     sp, sp, #32
100003fac: c0 03 5f d6  ret
```

Figure 3.2 – Contenu d'un programme

L'encadré à gauche, tracé sur le résultat de la commande `objdump`, est une représentation sous forme de paires de chiffres hexadécimaux des octets qui doivent être placés dans la mémoire de travail, aux adresses indiquées par la première colonne de nombres. Ce sont ces octets que la machine exécute. Il s'agit là d'une représentation « brute » de ce qu'il convient d'appeler le langage machine. L'encadré à droite est la représentation de ce langage machine en utilisant les mots-clés du langage assembleur. Tous les deux sont associés au fichier source du code 3.1. Bien que le langage machine soit l'unique langage que peut comprendre une machine, il reste laborieux à exploiter. Le langage assembleur est le langage de plus bas niveau qui représente le langage machine sous une forme lisible (si ce n'est intelligible) par un être humain. Il s'agit cependant d'une simple représentation sous forme de mnémoniques des opérations machines solicitées par le programme. Il n'est que très rarement utilisé pour programmer un ordinateur moderne, mais il est parfois utile pour analyser des éléments techniques d'un programme exécutable.

Pour développer un programme particulier, il est bien plus efficace d'utiliser un *langage de programmation* adapté à la nature du projet. Des langages de haut niveau permettent de s'abstraire des contingences matérielles et de focaliser son attention sur les difficultés intrinsèques au projet poursuivi. Toutefois quel que soit le langage utilisé pour cela, il est nécessairement traduit en langage machine pour qu'un ordinateur puisse l'exécuter. Cette traduction peut notamment consister en la production d'un fichier exécutable à partir d'un fichier source, c'est l'opération de compilation illustré précédemment. Mais il existe aussi d'autres moyens de transformer un programme source en un programme exécuté par la machine.

3.2.2 Langages de programmation

Un langage de programmation est un langage composé d'un alphabet, d'un ensemble de mots-clés, de règles de grammaire, et surtout d'une sémantique bien définie. Cependant pour qu'un tel langage puisse être utilisé pour produire des programmes, il doit avant tout disposer d'un ensemble d'outils logiciels, de programmes, permettant de le traduire vers les langages des machines. Ce sont ces outils autant que le langage choisi qui permettent de faire exécuter des algorithmes par des ordinateurs. Comme cela a été indiqué précédemment, la première personne à avoir programmé une machine est Ada Lovelace [11]. Si elle ne fut pas la première à concevoir des algorithmes, elle est considérée comme la première informaticienne car elle fut la première à exprimer un algorithme sous une forme qui puisse être exécuté par une machine. Elle définit notamment, en 1842, une séquence d'opérations calculant les nombres de Bernoulli dans le langage de la première machine destinée à être programmable. Malheureusement pour elle, la machine qui devait être capable de « donner vie » à son programme, la machine analytique de Charles Babbage, ne fut jamais achevée.

La *programmation* ou l'*implémentation* fait référence à la rédaction du code d'un programme dans un *fichier source*. Le terme *développement* fait plutôt référence à l'ensemble des activités liées à la création d'un programme : la spécification du logiciel depuis sa conception, son implémentation proprement dite ainsi que le suivi après son déploiement, jusqu'à son exécution.

3.2.3 Langage machine

Afin d'exécuter un programme, il convient donc aujourd'hui, en premier lieu, d'écrire un *fichier source* qui respecte une grammaire bien définie pour exprimer ce que l'on appelle le *code* dans un langage de programmation. Dans tous les cas, pour qu'un programme puisse fonctionner, il faut ensuite traduire ce fichier source en une série d'opérations machine. Ces opérations peuvent par exemple être écrites dans un *fichier exécutable*. Le système d'exploitation pourra alors charger ce fichier exécutable en mémoire puis en faire exécuter les opérations machine par le microprocesseur.

Deux stratégies sont envisagées suivant le type de langage utilisé et suivant l'objectif recherché : interprétation ou compilation qui sont résumés dans la figure 3.3.

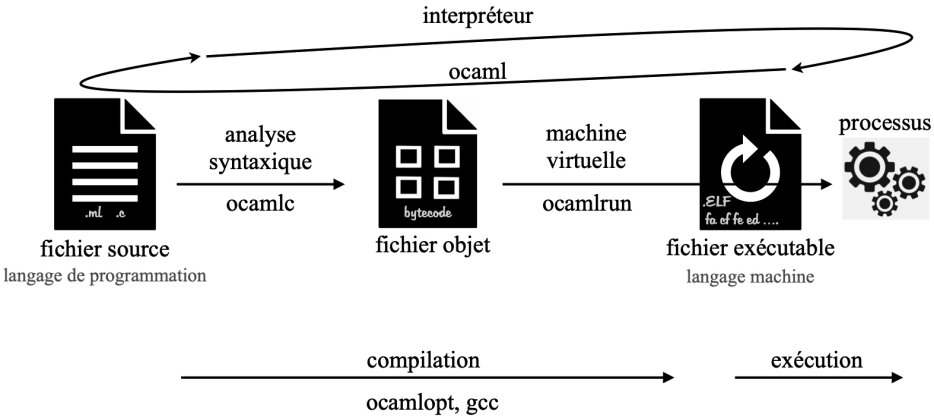


FIGURE 3.3 – Chemins du programme source à son exécution

Pour obtenir le fichier exécutable, il faut deux étapes de traitement :

- une analyse syntaxique[1] du fichier source (qui peut être stocké dans le fichier source) afin de construire un ensemble de structures de données (éventuellement écrits dans un ou plusieurs *fichiers objet*[2]) qui en représente les éléments sémantiques sous une forme exploitable par la machine ;
- une traduction de ces structures de données (éventuellement lues depuis ce(s) fichier(s) objet) dans le langage machine (possiblement stocké dans un *fichier exécutable*).

Ces deux étapes peuvent être réalisées séparément, d'une façon imbriquée ou encore en même temps. Elles peuvent être entièrement exécutées dans la mémoire de travail de la machine sans jamais être écrites dans un fichier, ou, au contraire écrites dans des fichiers distincts. Finalement, pour pouvoir lancer le programme, il faut exécuter le code machine ainsi formé.

3.2.4 Interprétation

Certains langages de programmation proposent un *interpréteur en ligne de commande* (*cli* pour *command-line interpreter*). Un tel interpréteur est un programme dont la tâche est d'effectuer l'analyse syntaxique, de traduire le résultat en séquences d'opérations machine et finalement de faire exécuter ces séquences au microprocesseur. Dans la boîte à outils des programmes proposés autour du langage `OCaml`, on trouve un tel interpréteur interactif. Il s'agit (notamment) du programme appelé `ocaml` qui peut être exécuté avec la commande du même nom.

1. Ce qui est ici nommé, par souci de concision, une analyse syntaxique se compose en fait en une analyse lexicale suivie d'une analyse grammaticale au programme de MPI.

2. Ce que cet ouvrage nomme fichier objet est parfois désigné par « *fichiers objet bytecode* » de l'anglais *bytecode object file* lorsqu'il s'agit du langage `OCaml`.

```
$ ocaml
OCaml version 4.14.0
Enter #help;; for help.
#
```

Le logiciel est alors en attente de la saisie d'une instruction sur le prompt. Il suffit de lui donner l'instruction en `OCaml` et de terminer sa ligne par `;;` pour lui indiquer qu'il peut démarrer le processus d'analyse, de traduction et d'exécution. Une fois cet ensemble d'opérations terminé, l'interpréteur affiche le résultat obtenu et attend la saisie de la séquence d'instructions suivante.

```
# let addition nbr_1 nbr_2 = nbr_1 + nbr_2
  let () =
    let result = addition 2 3 in
    Printf.printf "%d\n" result;;
5
val addition : int -> int -> int = <fun>
```

Code 3.2 – `[OCaml]` Exemple d'utilisation de l'interpréteur `ocaml`

Les langages dont les programmes sont généralement exécutés par un interpréteur sont qualifiés de *langages interprétés*. `Python`, `Javascript` et `bash`, parmi de nombreux autres en sont des exemples bien connus.

L'interpréteur peut être utilisé pour exécuter un code écrit dans un fichier source. Il peut donner l'illusion que la machine exécute directement le fichier source, mais ce qu'exécute vraiment la machine dans ce cas, c'est le programme d'interprétation. Pour ce dernier le code source n'est qu'une donnée. On prend l'exemple du code 3.2 écrit dans un fichier source nommé `addition.ml`. On peut utiliser directement ce fichier dans l'interpréteur `ocaml` :

```
# #use "addition.ml";;
val addition : int -> int -> int = <fun>
5
```

Il est aussi possible de lancer l'interprétation de ce fichier *via* l'interpréteur `ocaml` :

```
$ ocaml addition.ml
5
```

Dans les deux cas, c'est bien le programme `ocaml` qui est exécuté par la machine.

Terminologie - `ocaml`

`ocaml` est un interpréteur en ligne de commande, simplement appelé *interpréteur* dans cet ouvrage. Un tel programme produit, à partir d'un code source, une exécution des opérations sur le microprocesseur. L'interpréteur étant un programme exécutable sur le microprocesseur, il faut autant de programmes d'interprétation pour un langage que de familles de microprocesseurs ciblées.

3.2.5 Compilation

Un compilateur se distingue d'un interpréteur par le fait qu'il effectue l'analyse syntaxique et la traduction en langage machine pour la création d'un fichier exécutable. Ce fichier exécutable contient les instructions machines nécessaires à l'exécution du programme, ce qui permet au microprocesseur d'exécuter le programme directement sans passer par une étape d'analyse et de traduction intermédiaire. Une fois le fichier exécutable créé à partir du fichier source, il peut être exécuté plusieurs fois sans que la machine ait à réexécuter au préalable les traitements d'analyses syntaxique et de production de code machine. Cela se traduit par un gain de temps significatif lorsque les utilisateurs sollicitent ces logiciels, impactant favorablement les performances des programmes, notamment pour des applications fréquemment utilisées ou particulièrement longues et lourdes en temps de calcul sur la machine.

Parmi les programmes proposés pour servir le langage `OCaml`, `ocamlopt` joue précisément le rôle de compilateur. Son utilisation est illustrée ci-dessous :

```
$ ocamlopt addition.ml
$ ls
a.out  addition.cmi addition.cmx addition.ml  addition.o
$ ./a.out
5
```

Dans cet exemple, le fichier `a.out` est composé de code machine que l'on peut consulter avec des outils comme `objdump`. Ce fichier contient des instructions directement exécutables par le microprocesseur, illustrant le processus de compilation où le fichier source `addition.ml` est transformé en un fichier exécutable.

Terminologie - `ocamlopt`

`ocamlopt` est un compilateur natif qu'on appelle simplement *compilateur* dans cet ouvrage. Un tel programme produit à partir d'un code source un code directement exécutable par une famille de microprocesseurs défini à l'avance. Par défaut, le code produit par un programme de compilation est destiné à être exécuté sur la même famille de microprocesseurs que celui sur lequel est exécuté le compilateur. Lorsqu'un compilateur produit un code pour une machine différente de celle sur laquelle il est exécuté, on parle de compilation croisée (*cross-compilation*) mais dans ce cas il ne peut pas être directement exécuté sur la machine sur laquelle il est produit.

Pour le langage `C`, il existe un grand nombre de compilateurs. Certains sont proposés par les producteurs de microprocesseurs, d'autres sont vendus par des éditeurs de logiciels, et enfin, certains sont produits par des communautés de développeurs sous licence en sources ouvertes. C'est le cas du compilateur `gcc` qui est utilisé pour illustrer les propos de cet ouvrage.