Option informatique

Olivier Reynet

20-01-2025

TABLE DES MATIÈRES

I Introduction **3**

1 Algorithmes, types et structures **5**
A Des types abstraits de données et des structures de données 6
a Types simples, composés et instances 6
b Type abstrait de données et structure de données 6
c TAD Tableau 8
B $\mathbb{N}$: paradigme d'un ensemble inductif dont l'ordre est bien fondé 9
C De l'induction à l'induction structurelle 11
D Structures de données définies par induction 13
a Exemple TAD Liste 13
b Structure de liste inductive 14
E Induction structurelle 14
F Calculer des fonctions sur une structure inductive 15
G Pourquoi OCaml? 15

2 OCaml, des fonctions et des types **17**
A Pratiquer OCaml 17
B Vocabulaire utile 17
C Description d'OCaml 18
D Expressions et inférence de type 19
E Expressions locales et globales 20
F Fonctions 21
G Effets 22
H Types algébriques 24
I Listes 26
J Filtrage de motif 28
K Références et programmation impérative en OCaml 30
L Synthèse 32

II Logique **33**

3 De la logique avant toute chose **35**
A Logique et applications 35

B Constantes, variables propositionnelles et opérateurs logiques . . . 37
C Opérateurs logique et notations . . . 37
D Formules logiques : définition inductive et syntaxe . . . 39
E Sémantique et valuation des formules logiques . . . 42
F Lois de la logique . . . 45
a Éléments neutres et absorbants, idempotence . . . 45
b Commutativité, distributivité, associativité . . . 45
c Lois de De Morgan . . . 45
d Décomposition des opérateurs . . . 45
e Démonstrations . . . 46
G Principes et logique classique . . . 46
H Formes normales . . . 47
I Problème SAT . . . 49
J Algorithme de Quine . . . 49
a Principe . . . 49
b Règles de simplification . . . 51
K Exemple de démonstration par induction structurelle . . . 52

4 Déduction naturelle **53**
A Déduction naturelle . . . 53
B Règles d'introduction et d'élimination . . . 55
a Introduction et élimination de la conjonction . . . 55
b Introduction et élimination de l'implication . . . 55
c Introduction et élimination de la disjonction . . . 56
d Introduction et élimination de la négation . . . 57
C Synthèse des règles de la déduction naturelle . . . 57
D Correction de la déduction naturelle . . . 57
E Raisonnements utiles en logique classique . . . 60
a Raisonnement par l'absurde . . . 60
b Élimination de la double négation . . . 60
c Tiers exclu . . . 61
F Exemples de preuves . . . 61
a Syllogisme hypothétique . . . 61
b Modus tollendo tollens . . . 61
G Vers la logique du premier ordre --→ HORS PROGRAMME . . . 62
a Syllogismes . . . 62
b Règles du quantificateur existentiel . . . 63
c Règles du quantificateur universel . . . 63
H Correspondance Curry-Howard --→ HORS PROGRAMME . . . 63

III Structures de données **67**

5 Structures et types abstraits **69**
A Type abstrait de données et structure de données 70
B TAD Tableau 71
C TAD Liste 72
D TAD Dictonnaire 73
E Implémentations des tableaux 74
a Implémentation d'un tableau statique 74
b Implémentation d'un tableau dynamique 75
F Implémentations des listes 77
a Listes simplement chaînées 77
b Listes doublement chaînées 79
G Bilan des opérations sur les structures listes et tableaux 80
H Implémentation d'un TAD dictionnaire 80

6 Arbres binaires et ABR **83**
A Des arbres 83
B Arbres binaires 86
C Définition inductive des arbres binaires 87
D Démonstration par induction structurelle 89
E Définitions inductives de fonction sur les arbres 90
F Parcours en profondeur d'un arbre binaire 91
G Proprétés et manipulation des arbres binaires 93
H Arbre binaire de recherche (ABR) 93
I Opérations sur les arbres binaires de recherche 94
J Maintenir l'équilibre ⇢ HORS PROGRAMME 96

7 Arbres génériques et préfixes **97**
A Arbres génériques 97
a Calculs sur un arbre générique 98
b Transformer un arbre générique en arbre binaire 98
B Arbres préfixes 99
C Compression statistique 100
D Compression à dictionnaire 103

IV Programmation récursive **105**

V Exploration et graphes **107**

8 Retour sur trace **109**
A Exploration 109
B Principe du retrour sur trace 111

9 Des arbres aux tas **115**
A Tas binaires . . . 115
a Définition . . . 115
b Implémentation . . . 116
c Opérations . . . 117
B Tri par tas binaire . . . 118
C File de priorités implémentée par un tas . . . 119

10 Graphes avancés **121**
A Graphes connexes et acycliques . . . 121
B Arbres recouvrants . . . 124
a Algorithme de Prim . . . 127
b Algorithme de Kruskal . . . 129
c Unir et trouver --→ HORS PROGRAMME . . . 130
C Tri topologique d'un graphe orienté . . . 131
a Ordre dans un graphe orienté acyclique . . . 131
b Existence d'un tri topologique . . . 132
c Algorithmes de tri topologique . . . 133
D Composantes fortement connexes d'un graphe orienté et 2-SAT . . . 135
E Graphes bipartis et couplage maximum . . . 137
a Caractérisation des graphes bipartis . . . 137
b Couplage dans un graphe biparti . . . 138
c Chemin augmentant . . . 139
F Pour aller plus loin --→ HORS PROGRAMME . . . 143
a Affectation de ressources . . . 143
b Mariages stables ou appariement de ressources . . . 143

VI Langages et automates **145**

11 Introduction aux langages **147**
A Alphabets . . . 147
B Mots . . . 148
C Mots définis inductivement . . . 150
D Langages . . . 151
E Préfixes, suffixes, facteurs et sous-mots . . . 153
F Propriétés fondamentales . . . 155

12 Expressions régulières **157**
A Définition des expressions régulières . . . 158
B Définition des langages réguliers . . . 160
C Identités remarquables sur les expressions régulières . . . 161
D Arbre associé à une expression régulière . . . 162
E Expressions régulières dans les langages --→ HORS PROGRAMME . . . 162

13 Automates finis déterministes **165**
A Automate fini déterministe (AFD) . . . 166
B Représentations d'un automate . . . 167
C Acceptation d'un mot . . . 167
D Accessibilité et co-accessibilité . . . 168
E Complétion d'un AFD . . . 168
F Complémentaire d'un AFD . . . 170
G Produit de deux AFD - Automates produit . . . 171

14 Automates finis non déterministes **173**
A Automate fini non déterministe (AFND) . . . 173
B Représentation d'un AFND . . . 174
C Acceptation d'un mot . . . 174
D Déterminisé d'un AFND . . . 175
E ϵ-transitions . . . 178

15 Des expressions régulières aux automates **181**
A Théorème de Kleene . . . 181
B Algorithme de Thompson . . . 182
a Patron de conception d'un cas de base . . . 183
b Patron de conception de l'union . . . 183
c Patron de conception de la concaténation . . . 184
d Patron de conception de l'étoile de Kleene . . . 184
e Application . . . 184
f Élimination des transitions spontanées . . . 184
C Algorithme de Berry-Sethi et automate de Glushkov . . . 187
a Langages locaux . . . 187
b Expressions régulières linéaires . . . 190
c Automate locaux . . . 191
d Automate de Glushkov et algorithme de Berry-Sethi . . . 192
D Comparaison Thompson / Berry-Sethi . . . 195

16 Des automates aux expressions rationnelles **197**
A Automate généralisé . . . 197
B D'un automate généralisé à une expression régulière . . . 197

17 Au-delà des langages réguliers **201**
A Limites des expressions régulières . . . 201
B Caractériser un langage régulier . . . 202
C Les langages des puissances . . . 203

18 À savoir pour les concours **205**
A Algorithmes à savoir écrire en OCaml en moins de 5 minutes . . . 205
B Algorithmes classiques à savoir implémenter en 10 minutes . . . 205
C Algorithmes dont la connaissance aide à finir plus vite . . . 206

D Automates et langages réguliers 206
E Arbres et graphes 207
F Logique 207
G Techniques algorithmiques 207
H Techniques mathématiques 208

VII Annexes **209**

Bibliographie **211**
Articles 211
Livres 212
Sites web 212

LISTE DES ALGORITHMES

1 Algorithme Quine (SAT) 50
2 Construction d'un arbre préfixe associé à un code de Huffmann 101
3 Compresser à l'aide d'un arbre de Huffmann 102
4 Décompresser à l'aide d'un arbre de Huffmann 102
5 Algorithme de recherche par force brute, problème de satisfaction de contraintes 109
6 Algorithme d'exploration des solutions avec retour sur trace 111
7 Tri par tas, ascendant 119
8 Algorithme de Prim, arbre recouvrant 127
9 Algorithme de Kruskal, arbre recouvrant 130
10 Algorithme de Kahn pour le tri topologique 133
11 Tri topologique 134
12 Recherche d'un couplage de cardinal maximum 141
13 Algorithme de déterminisation d'un AFND 176

Première partie

Introduction

1

ALGORITHMES, TYPES ET STRUCTURES

Comme mentionné dans l'introduction de l'informatique commune, l'informatique est la science de construction de l'information par le calcul. Mais, s'il est facile de comprendre qu'on peut calculer sur les nombres, comment peut-on calculer de l'information en général ? Calculer sur les entiers est une chose, calculer une information en général en est une autre. La solution réside dans la création de structures de données concrètes qui rendent possible le calcul sur des concepts plus sophistiqués que les nombres.

Ces structures de données sont construites à partir de briques de bases, les types simples. Écrire un programme optimal en terme de complexité nécessite l'identification des structures de données utilisées : le choix d'une structure de données plutôt qu'une autre, par exemple choisir une entier long plutôt qu'un flottant ou une liste au lieu d'un tableau, peut rendre inefficace un algorithme selon le choix effectué.

Le concept de type abstrait de données permet d'écrire un algorithme en faisant abstraction de l'implémentation de la structure de données (cf. figure 1.1). Par exemple, un dictionnaire peut-être implémenté de différentes manières, soit en utilisant un arbre, soit en utilisant un tableau dynamique. Un même TAD peut être réalisé par plusieurs structures de données dont les performances en complexité temporelle ou mémoire ne sont pas nécessairement équivalentes.

L'étude des types abstraits et des structures de données associée à l'algorithme est donc fondamentale en informatique.

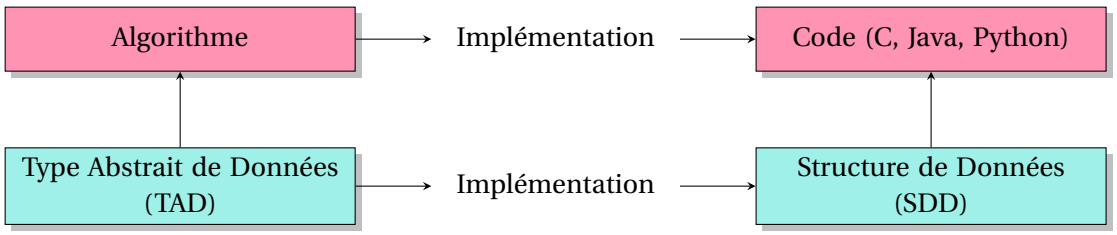


FIGURE 1.1 – Imbrication des concepts d'algorithme, de type abstrait de données, d'implémentation, de code et de structure de données

A Des types abstraits de données et des structures de données

a Types simples, composés et instances

Les types simples sont généralement les entiers, les flottants et les booléen. S'y ajoutent parfois les caractères qui sont représentés par des entiers.

Les types composés sont les listes, les tableaux, les arbres, les files, les piles. Ce sont des structures composites qui permettent et de manipuler l'information sous la forme d'ensembles ordonnés ou non.

■ **Définition 1 — Instance.** En informatique, une instance est un objet matérialisé dans la machine qui représente un exemplaire d'un certain type d'un langage dans la mémoire.

■ **Exemple 1 — Instance et types de base.** En python, la séquence :

- `a = 3` créé une instance d'un type `int` qui vaut 3,
- `f = 3.5` créé une instance du type `float` qui vaut 3.5,
- `b = True` créé une instance du type `bool` qui vaut `True`,

En OCaml, l'équivalent s'exprime :

- `let i = 3`,
- `let f = 3.5`
- `let b = true`

b Type abstrait de données et structure de données

■ **Définition 2 — Type abstrait de données (TAD).** Un type de données abstrait est une abstraction d'une structure de données qui ne se préoccupe pas de son implémentation sur une machine : sa structure interne est indiscernable, le type abstrait est vu de l'extérieur comme une boite noire.

Un TAD spécifie le quoi, c'est-à-dire le type de données contenues ainsi que les opérations possibles. Par contre, il ne spécifie pas comment dont les données sont stockées ni comment les opérations sont implémentées.

■ **Définition 3 — Structure de données.** Une structure de données est une mise en œuvre concrète d'un type abstrait, une implémentation d'un type abstrait sur dans un langage de programmation. On y décrit donc la manière avec laquelle sont codées les données et les opérations en machine.

R Un type abstrait de données est à une structure de donnée ce qu'un algorithme est à un programme. On spécifie un algorithme ou un type abstrait de données, mais on implémente un programme ou une structure de données.

■ **Exemple 2 — Un entier.** Un entier est un TAD qui :

(données) contient une suite de chiffres [a] éventuellement précédés par un signe − ou +,

(opérations) fournit les opérations $+, -, \times, //, \%$.

Selon le langage, ce TAD entier est implémenté en machine par un type concret différent :

- `int` en Python,
- `Integer` ou `int` en Java,
- `char`, `short`, `int`, `uint`, `long int` en C,
- `int` en OCaml.

a. peu importe la base pour l'instant...

■ **Exemple 3 — Un booléén.** De la même manière, on peut définir un TAD qui désigne un booléen. Un booléen est un TAD qui :

(données) se note Vrai ou Faux,

(opérations) fournit les opérations logiques conjonction, disjonction et négation...

Selon le langage, ce TAD booléen est implémenté en machine par un type concret différent :

- `bool` valant `True` ou `False` en Python,
- `boolean` valant `true` ou `false` en Java,
- `bool` valant `1` ou `0` en C,
- `bool` valant `true` ou `false` en OCaml.

Les exemples précédents de types abstrait de données étaient limités à des types simples. Mais il est possible de définir des types abstraits de données composés.

■ **Exemple 4 — Types abstraits de données composés.** Voici quelques types abstraits composés parmi les plus courants :

- liste,
- file,
- pile,
- arbre binaire,
- dictionnaire ou tableau associatif,
- ensemble,
- graphe.

c TAD Tableau

Le tronc commun informatique se concentre sur l'utilisation des structures **impératives**, c'est-à-dire essentiellement les tableaux statiques, dynamiques et les variables [1].

■ **Définition 4 — TAD tableau.** Un TAD tableau représente une structure finie indiçable par des entiers. Cela signifie qu'on peut accéder à la lecture ou à l'écriture de n'importe quel élément directement en utilisant un indice, par exemple `t[3]`.

(données) le plus souvent des nombres, en tout cas des types identiques : on appelle la donnée l'élément d'un tableau.

(opérations) on distingue deux opérations principales caractéristiques :

- l'accès à un élément via un indice entier via un opérateur de type `[]`,
- l'enregistrement de la valeur d'un élément d'après son indice.

Les implémentations du TAD tableau sont la plupart du temps des structures des données linéaires en mémoire : les données d'un tableau sont rangées dans des zones mémoires **continues**, les unes derrières les autres.

D'un point de vue de l'implémentation, on peut construire le TAD tableau de manière :

1. statique : la taille du tableau est fixée, on ne peut pas ajouter ou enlever d'éléments.
2. dynamique : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

R En Python, il n'existe pas de type tableau statique dans le cœur du langage. Cependant, la liste Python est implémentée par un tableau dynamique et permet donc de pallier ce manque. Néanmoins, pour le calcul numérique, il faut absolument privilégier l'usage des tableaux Numpy qui implémentent le TAD tableau statique.

On qualifie souvent les implémentations du TAD tableau d'**impératives** : il s'agit d'une zone précise de la mémoire que l'on peut **modifier** en effectuant un effet de bord. Les tableaux sont des types **muables**. Par exemple, on peut changer la valeur de la première case d'un tableau en Python par l'instruction `t[0] = 3`.

R Le caractère muable de ces structures impératives est un inconvénient lorsqu'on cherche à prouver la correction d'un programme ou que l'on travaille dans le contexte de la programmation concurrente. Fort heureusement, il est possible de construire des structures **immuables** [a] plus adaptées à ces objectifs : des structures de données inductives et immuables.

a. on dit aussi persistantes

1. références en OCaml

B ℕ : paradigme d'un ensemble inductif dont l'ordre est bien fondé

Le deux pierres angulaires aux fondements de l'informatique sont la théorie des ensembles et la logique. La fin du XIX^{e} siècle a marqué un tournant dans l'histoire des sciences et en particulier pour les mathématiques : la théorie des ensembles et ses paradoxes ont forcé les mathématiciens à mieux formaliser les raisonnements et les démonstrations, c'est-à-dire les types et la logique.

Les entiers naturels, les éléments de l'ensemble ℕ, constituent le fondement de l'informatique. Or, on ne peut pas construire cet ensemble des nombres entiers naturels, ensemble pourtant constitué des objets mathématiques les plus évidents : tout le monde comprend les concepts 0, 1, 2, construire l'ensemble $\{0,1,2,\ldots\}$ nécessite d'expliciter les «…», ce qui n'est pas simple a priori. Les premières définition axiomatiques de ℕ apparaissent à la fin du XIX^{e} siècle, sous l'impulsion de Dedekind et Peano.

■ **Définition 5 — Définition axiomatique de l'ensemble des entiers naturels.** On postule qu'il existe un ensemble dit des «entiers naturels» noté ℕ muni de :

1. l'élément zéro, noté 0 est un entier naturel.
2. une application «successeur» $s : \mathbb{N} \to \mathbb{N}$, vérifiant les axiomes de Peano suivants :
   (a) s est injective,
   (b) Zéro n'est le successeur d'aucun entier naturel,
   (c) (Axiome de récurrence) Soit A un sous-ensemble de ℕ tel que :
   - $0 \in A$,
   - $\forall n \in A, s(n) \in A$ (stabilité de l'ensemble),

   alors $A = \mathbb{N}$.

■ **Définition 6 — Addition.** L'opération addition sur les entiers naturels est définie par :

(B) $a + 0 = a$

(I) $a + s(b) = s(a + b)$

■ **Définition 7 — Multiplication.** L'opération multiplication sur les entiers naturels est définie par :

(B) $a \times 0 = 0$

(I) $a \times s(b) = a \times b + a$

R On peut vérifier que l'addition et la multiplication sont commutatives et que la multiplication est distributive par rapport à l'addition. On en déduit d'une relation d'ordre $\leqslant$ sur ℕ compatible avec ces opérations :

$$\forall (m,n) \in \mathbb{N}^2, m \leq n \iff \exists k \in \mathbb{N}, n = m + k$$

R En notant $1 = s(0)$, on peut formellement se débarrasser de la fonction s en la remplaçant par $a \to a+1$. On simplifie alors les démonstrations.

Théorème 1 Toute partie non vide de N admet un plus petit élément.

Démonstration. Par l'absurde. Soit A une partie non vide de $\mathbb{N}$ n'admettant pas de plus petit élément. Considérons l'ensemble B défini par $B = \{n \in \mathbb{N}, \forall k \leq n, k \notin A\}$.

- Zéro est dans B, car dans le cas contraire il serait dans A et constituerait un plus petit élément de A, ce qui contredit notre hypothèse.
- Soit n, un élément quelconque de B. On cherche à montrer que $n+1$ est dans B. On procède par l'absurde. Supposons que $n+1$ appartient à A. Comme $A \in \mathbb{N}$, A contient l'ensemble de ses successeurs et il existe un $k \in \mathbb{N}$ tel que $k \leqslant n$, c'est-à-dire $n+1$ est le successeur de k dans A. Mais on a nécessairement $k = n+1$, sinon k appartiendrait à B, par définition de B. On a donc :

  $$\forall k \leqslant n, k \notin A \text{ et } n+1 \in A$$

  ce qui signifie $\forall i \in A, i \leqslant n+1$, c'est-à-dire tous les éléments de A sont plus grands que $n+1$. Donc A possède un plus petit élément, ce qui est absurde. Donc $n+1$ appartient à B.

L'ensemble B vérifie donc le principe de récurrence et on a $B = \mathbb{N}$. On en déduit que $A = \emptyset$, ce qui est absurde puisqu'on a supposé que A n'était pas vide. ■

Théorème 2 — Principe d'induction. Soit $\mathcal{P}_n$ une propriété fonction d'un entier naturel n. Lorsque les deux conditions suivantes sont vérifiées :

(B) $\mathcal{P}_0$ est vraie,

(I) $\mathcal{P}_n$ implique $\mathcal{P}_{n+1}$ pour tout n,

alors pour tout entier naturel n, $\mathcal{P}_n$ est vraie.

Démonstration. Par l'absurde. Soit $\mathcal{P}_n$ une propriété fonction d'un entier naturel n vérifiant les conditions (B) et (I). Supposons qu'il existe des entiers k pour lesquels la proposition $\mathcal{P}_k$ soit fausse et notons l'ensemble de ces entiers X : $X = \{k \in \mathbb{N}, \mathcal{P}_k \text{ est fausse}\}$.

Si X est vide, alors $\mathcal{P}_n$ est vraie pour pour tout entier naturel.

Si X est non vide, il admet un plus petit élément v d'après le théorème 1.

- soit $v = 0$ et alors $\mathcal{P}_0$ est fausse, ce qui contredit la propriété (B) et notre hypothèse. C'est absurde.
- soit $v > 0$, $\mathcal{P}_v$ est fausse et $\mathcal{P}_{v-1}$ est vraie. Donc (I) n'est pas vérifiée par $\mathcal{P}_n$, ce qui contredit notre hypothèse. C'est absurde.

On en conclut que $\mathcal{P}_n$, à partir du moment où elle vérifie les propriétés (B) et (I), est vraie pour tout entier naturel n. ■

R L'élément qui cristallise l'intérêt des informaticiens pour le principe d'induction, c'est qu'il est **constructif** : il permet, à partir d'une information simple (cas de base B) et de règles d'induction simples (I), de calculer un ensemble d'information.

■ **Définition 8 — Ordre bien fondé.** Soit $(E, \leqslant)$ un ensemble muni d'une relation d'ordre. On dit que l'ordre sur E est bien fondé s'il n'existe pas de suite infinie strictement décroissante d'éléments de E.

■ **Définition 9 — Ensemble bien ordonné.** Soit $(E, \leqslant)$ un ensemble muni d'une relation d'ordre, c'est-à-dire un ensemble ordonné. On dit que l'ordre sur E est bien ordonné s'il est également bien fondé.

R Un ensemble bien ordonné est nécessairement doté d'une relation d'ordre totale.

Théorème 3 — Caractérisation des ordres bien fondés. Soit Soit $(E, \leqslant)$ un ensemble. L'ordre de E est bien fondé si et seulement si toute partie non vide F de E admet un élément minimal.

R On déduit des théorèmes 3 et 1 que l'ordre $\leqslant$ associé à l'ensemble $\mathbb{N}$ des entiers naturels est bien fondé.

R $(\mathbb{N}, \leqslant)$ est un ensemble bien ordonné mais $(\mathbb{Z}, \leqslant)$ ne l'est pas.

C De l'induction à l'induction structurelle

Cette section étend le principe d'induction à tout ensemble construit de manière inductive.

■ **Définition 10 — Définition inductive d'un ensemble.** Soit E un ensemble. Une définition inductive d'une partie X de E consiste à se donner :

(B) un sous ensemble non vide $\mathcal{B}$ de E appelé ensemble de base,

(I) et d'un ensemble de règles $\mathcal{R}$: chaque règle $r_i \in \mathcal{R}$ est une fonction r_i de $E^{n_i} \to E$ telle que $\forall x_1, \cdots x_{n_i} \in X \Rightarrow r_i\left(x_1, \cdots, x_{n_i}\right) \in X$ pour $n_i > 1$.

On dit que X est alors défini inductivement. On appelle les r_i les constructeurs de X.

Théorème 4 — Théorème du point fixe. À une définition inductive d'un ensemble correspond un plus petit ensemble qui vérifie les propriétés suivantes :

1. il contient $\mathcal{B}$, c'est-à-dire $\mathcal{B} \subset X$,
2. il est stable pour les règles de $\mathcal{R}$: pour chaque règle $r_i \in \mathcal{R}$, pour tout $x_1, \cdots, x_{n_i} \in X$, on a $r_i\left(x_1, \cdots, x_{n_i}\right) \in X$.

Démonstration. Soit E un ensemble défini inductivement comme en 10 à l'aide d'un ensemble de base $\mathcal{B}$ et des règles $\mathcal{R}$.

Soit $\mathcal{P}$ l'ensemble des parties de E vérifiant (B) et (I).

$\mathcal{P}$ est non vide car il contient au moins l'ensemble E qui vérifie les conditions (B) et (I) : $E \in \mathcal{P}$.

Considérons alors X l'ensemble défini comme l'intersection de tous les éléments de $\mathcal{P}$:

$$X = \bigcap_{Y \in \mathcal{P}} Y$$

Par définition, $\mathcal{B}$ est inclus dans chaque $Y \in \mathcal{P}$. $\mathcal{B}$ est donc inclus dans X et X vérifie la condition (B).

On peut montrer que X vérifie également la condition (I). Soit une règle $r_i \in \mathcal{R}$ et des $x_1, \cdots, x_{n_i} \in X$. On a $x_1, \cdots, x_{n_i} \in Y$ pour chaque $Y \in \mathcal{P}$. Pour chaque tel Y, puisque Y est stable par la règle r_i, on doit avoir $r\left(x_1, \cdots, x_{n_i}\right) \in Y$. Puisque cela est vrai pour tout $Y \in \mathcal{P}$, on a aussi $r\left(x_1, \cdots, x_{n_i}\right) \in X$. Donc X est stable par la règle r_i.

Enfin, X est le plus petit ensemble qui vérifie les conditions (B) et (I), car, de par sa définition, il est **inclus dans tout autre ensemble** vérifiant les conditions (B) et (I). ■

Une structure définie inductivement selon 10 possède donc un plus petit élément contenu dans toutes les constructions possibles. On peut montrer que cette structure possède une ordre bien fondé.

R Puisqu'on sait qu'il existe, on considère alors **le plus petit de ces ensembles** défini inductivement qui contient $\mathcal{B}$ et qui est stable par les règles de construction $\mathcal{R}$.

En effet, si on définit l'ensemble des entiers pairs P par 0 et la règle $\forall n \in P, n+2 \in P$, alors on constate que $\mathbb{N}$ vérifie bien ces deux propriétés. Néanmoins, ce n'est pas le plus petit des ensembles caractérisés par cette définition. L'ensemble des nombres pairs est donc le plus petit de ces ensembles définis inductivement.

■ **Définition 11 — Principe d'induction structurelle.** Soit E un ensemble défini **inductivement**, de termes de base $\mathcal{B}$ et de règles de construction $\mathcal{R}$

Soit $\mathcal{P}$ une propriété sur les termes de E.

Si $\mathcal{P}$ est satisfaite pour chaque terme de base de $\mathcal{B}$ (cas de base) et pour chaque constructeur de $\mathcal{R}$ (pas d'induction), alors $\mathcal{P}$ est satisfaite pour tous les termes de E.

Plus formellement,

$$\left.\begin{array}{ll} \forall b \in \mathcal{B}, & \mathcal{P}(b) \\ \forall r \in \mathcal{R}, \forall x_1, x_2, \ldots x_n \in E, & \mathcal{P}(r(x_1, x_2, \ldots, x_n)) \end{array}\right\} \Longrightarrow \forall x \in E, \mathcal{P}(x) \qquad (1.1)$$

Théorème 5 — Fonction définie inductivement. Soit $X \subset E$ un ensemble défini inductivement de façon non ambigüe à partir de l'ensemble de base $\mathcal{B}$ et des règles $\mathcal{R}$. Soit Y un ensemble.

Pour qu'une application f de X dans Y soit parfaitement définie, il suffit de se donner :

(B) la valeur de $f(x)$ pour chacun des éléments $x \in B$,

(I) pour chaque règle $r_i \in \mathcal{R}$, la valeur de $f(x)$ pour $x = r_i\left(x_1, \cdots, x_{n_i}\right)$ en fonction de la valeur $x_1, \ldots, x_{n_i}, f(x_1), \ldots$ et $f\left(x_{n_i}\right)$.

Démonstration. On cherche à montrer que la fonction f ainsi définie associe bien une unique valeur dans Y à chaque $x \in X$. On procède par induction.

- Cas de base : la valeur associée à chaque $x \in \mathcal{B}$ est bien unique d'après (B).
- Pas d'induction : supposons que cela est vrai pour $x_1, \cdots, x_{n_i}$. On cherche à montrer que cela est vrai en $x = r_i\left(x_1, \cdots, x_{n_i}\right)$, c'est-à-dire que les $f(x_1), \ldots, f(x_{n_i})$ sont définis de manière unique. Comme x ne peut être obtenu que par la règle r_i à partir de $x_1, \ldots, x_{n_i}$, $f(x)$ est donc parfaitement définie par la contrainte pour la règle (I).

■

Pour un ensemble X défini inductivement, une fonction est donc parfaitement définie par le théorème 5 sur X.

■ **Définition 12 — Factorielle, définie inductivement.** Comme on a définit l'ensemble des entiers naturels $\mathbb{N}$ inductivement (cf. définition 5), la fonction factorielle peut être définie inductivement par :

Fact : $\mathbb{N} \to \mathbb{N}$

(B) $\text{Fact}(0) = 1$,

(I) $\text{Fact}(s(n)) = s(n) \times \text{Fact}(n)$, ce qui peut également s'écrire : $\text{Fact}(n+1) = (n+1) \times \text{Fact}(n)$.

D Structures de données définies par induction

La capacité de construire des calculs sur les entiers naturels de manière inductive est fascinante : pour un mathématicien, cet ensemble bien ordonné engendre une infinité de preuves et permet de définir des fonctions sur $\mathbb{N}$; pour un informaticien, cet ensemble est le modèle qui permet la construction d'autres ensembles. Définir une structure de données de manière inductive, c'est pouvoir **construire et calculer** sur les éléments de cet ensemble, c'est-à-dire la raison d'être de l'informatique. Ces **structures inductives** irriguent toutes les parties du programme officiel de l'option informatique et la plupart des développement de ce cours : les listes, les arbres, formules logiques, et les expressions régulières.

a Exemple TAD Liste

■ **Définition 13 — TAD liste.** Un TAD liste représente **une séquence finie d'éléments d'un même type** qui possède un **rang** dans la séquence. Les données sont traitées séquentiellement, dans l'ordre du rang.

Un TAD liste est **dynamique**, c'est à dire qu'on peut ajouter ou enlever des éléments.

La longueur d'une liste est le nombre d'éléments qu'elle contient. On dit qu'une liste est vide si elle ne contient aucun élément, sa longueur vaut alors zéro. Le début de la liste est

désigné par le terme tête de liste (**head**), le dernier élément de la liste par la fin de la liste (**tail**).

(données) de type simple ou composé

(opérations) on peut trouver [a] :

- un constructeur de liste vide,
- un opérateur de test de liste vide,
- un opérateur pour ajouter en tête de liste,
- un opérateur pour ajouter en fin de liste,
- un opérateur pour déterminer et/ou retirer la tête de la liste,
- un opérateur pour déterminer et/ou retirer la queue de la liste (tout sauf la tête),
- un opérateur pour accéder au ième élément.
- un opérateur pour accéder au dernier élément de la liste.

a. Toutes les implémentations ne proposent pas nécessairement toutes ces opérations !

Le TAD liste peut être implémenté par un tableau de manière impérative [2] mais il peut également l'être par une structure définie inductivement.

b Structure de liste inductive

■ **Définition 14 — Liste construite de manière inductive.** Une liste d'éléments est soit :

($\mathcal{B}$) Vide

($\mathcal{C}$) une liste construite en AJOUTANT EN TÊTE un élément à une liste

R Les éléments d'une structure définie de manière inductive sont appelés **termes**.

R Un constructeur possède une certaine **arité** : il permet de construire un terme à partir de un ou plusieurs termes. Dans l'exemple de la liste, AJOUT EN TÊTE est un constructeur d'arité 1 car il ne prend qu'un seul paramètre.

R Cette définition inductive engendre un ordre dans les termes. **Cet ordre est partiel et bien fondé est appelé ordre structurel.**

E Induction structurelle

Il est possible de démontrer des propriétés des structures inductives grâce à l'**induction structurelle**.

2. comme c'est le cas en Python

■ **Définition 15 — Principe d'induction structurelle.** Soit $\mathcal{I}$ un ensemble inductif de termes de base $\mathcal{B}$ et de constructeurs $\mathcal{C}$. Soit $\mathcal{P}$ une propriété sur les termes de $\mathcal{I}$.

Si $\mathcal{P}$ est satisfaite pour chaque terme de base de $\mathcal{B}$ et pour chaque constructeur de $\mathcal{I}$, alors $\mathcal{P}$ est satisfaite pour tous les termes de $\mathcal{I}$.

Plus formellement,

$$\left.\begin{array}{ll} \forall b \in \mathcal{B}, & \mathcal{P}(b) \\ \forall c \in \mathcal{C}, \forall t_1, t_2, \dots t_n \in \mathcal{I}, & \mathcal{P}(c(t_1, t_2, \dots, t_n)) \end{array}\right\} \Longrightarrow \forall t \in \mathcal{I}, \mathcal{P}(t) \tag{1.2}$$

F Calculer des fonctions sur une structure inductive

On peut facilement définir des fonctions sur les ensembles inductifs en s'appuyant sur leur définition. Cela permet de faire des calcul sur l'information qu'ils représentent.

■ **Exemple 5 — Longueur d'une liste.** On peut définir la longueur d'une liste en s'appuyant sur la définition inductive : la longueur d'une liste vide est nulle et la longueur d'une liste à laquelle on ajoute en tête un élément est la longueur de cette liste plus un.

On obtient alors une fonction LONGUEUR définie récursivement :

(Cas de base) LONGUEUR(Vide) = 0

(Règle de construction) LONGUEUR(AJOUTER EN TÊTE(L,a)) = 1 + LONGUEUR(L)

G Pourquoi OCaml ?

Comme on peut le voir ci-dessous et comme on le verra dans les chapitres suivants, le langage OCaml est particulièrement adapté aux structures inductives pour plusieurs raisons :

1. les types en OCaml sont nativement récursifs,
2. les types OCaml peuvent être des types somme $|$ ou produit $*$. On les appelle des types algébriques.
3. OCaml procure la syntaxe dite du filtrage de motifs,
4. OCaml permet d'écrire des fonctions récursives.

Combiner ces quatre fonctionnalités permet de traduire directement la plupart des concepts mathématiques exprimés sous la forme d'un ensemble inductif. Les **fonctions** dont les paramètres peuvent être des types algébriques concrétisent les calculs sur les termes des ensembles inductifs.

Ces fonctionnalités de OCaml[3] expliquent en grande partie le choix du langage OCaml en CPGE pour l'option informatique.

3. Tous les langages ne dispose pas de ces fonctionnalités : Python ne dispose pas de types récursifs et algébriques et son paradigme est impératif, pas fonctionnel.

Vocabulary 1 — Pattern matching ⇜⇝ Filtrage de motifs

```
type ingredient =
        | Sel | Poivre | Beurre | Sucre | Farine
        | Fondre of ingredient
        | Melanger of ingredient*ingredient;;

let rec masse = function
        | Sel -> 30
        | Poivre -> 10
        | Beurre -> 250
        | Sucre -> 250
        | Farine  -> 10
        | Fondre i ->  masse i
        | Melanger (i1,i2) -> (masse i1) + (masse i2);;

let kouignamann = Melanger(Beurre, Sucre);;
let m = masse(kouignamann);;
```

2

OCaml, des fonctions et des types

A Pratiquer OCaml

Le plus simple Pour utiliser OCaml, il suffit de l'utiliser en ligne via les bacs à sable des sites OCaml ou TryOcaml.

Pour travailler localement Sur votre machine, le plus simple est d'utiliser l'interprète interactif OCaml. On peut facilement l'installer sur n'importe quel système d'exploitation en suivant ces instructions.

Pour travailler avec un éditeur de texte ăgrâce au mode Tuareg. Les commandes `M-x tuareg-mode` et `M-x run-ocaml` permettent d'activer ce mode. Le résumé (sic!) des commandes est accessible en ligne. Pour les puristes de la ligne de commande, ce mode est également disponible sous Vim.

Utiliser un IDE Enfin, il est possible d'utiliser OCaml avec la plupart des environnement de développement : Eclipse, Visual Studio ou IntelliJ (Jet Brains).

B Vocabulaire utile

■ **Définition 16 — Modèle de calcul.** Un modèle de calcul (MOC) est la description d'une manière de calculer une fonction mathématique étant donnée une entrée. Il existe des modèles de calcul séquentiels [a], fonctionnels [b] et concurrents [c].

a. les automates
b. le lambda calcul
c. les circuits logiques, les réseaux de Petri ou Synchronous Data Flow (cf. simulink)

■ **Définition 17 — Paradigme de programmation.** Un paradigme de programmation est un ensemble de formes et de figures qui constitue un modèle propre à un langage.

■ **Définition 18 — Paradigme impératif.** Le paradigme impératif s'attache à décrire des sé-

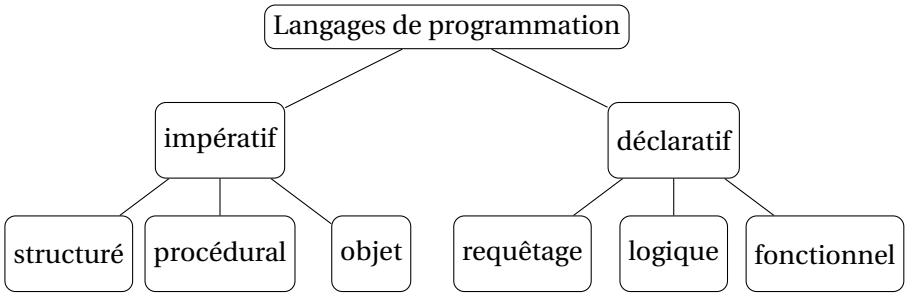


FIGURE 2.1 – Paradigmes des langages de programmation

quences d'instructions (ordres) qui agissent sur un état interne de la machine (contexte). L'impératif explicite le *comment procéder* pour exécuter un programme. Cette programmation se rapproche de la logique électronique des processeurs.

■ **Définition 19 — Paradigme procédural.** Ce paradigme est une déclinaison de l'impératif et propose de regrouper des éléments réutilisables de code dans des routines. Ces routines sont appelées procédures (si elles ne renvoient rien) ou fonctions (si elles renvoient un résultat).

■ **Définition 20 — Paradigme objet.** Ce paradigme est une déclinaison de l'impératif et propose de décrire un programme comme l'interaction entre des objets à définir. Une classe est un type d'objet qui possède des attributs et des comportements. Ces caractéristiques sont encapsulées et peuvent être masquées à l'utilisateur d'un objet : cela permet de protéger l'intégrité de l'objet et de garantir une cohérence dans la manipulation des données.

■ **Définition 21 — Paradigme déclaratif.** Le paradigme déclaratif est une syntaxe qui s'attache à décrire le *quoi*, c'est-à-dire *ce que le programme doit faire*, non pas comment il doit le faire. Un langage déclaratif ne dépend pas de l'état interne d'une machine (contexte). Cette programmation se rapproche de la logique mathématique et délègue au compilateur la délicate question du *comment procéder*.

■ **Définition 22 — Paradigme fonctionnel.** Le paradigme fonctionnel est une déclinaison du déclaratif qui considère qu'un programme n'est qu'un calcul et qu'un calcul est le résultat d'une fonction. Le mot fonction est ici à prendre au sens mathématique du terme (lambda calcul) : une fonction appelée avec les mêmes paramètres produit le même résultat en toute circonstance.

C Description d'OCaml

Le langage OCaml est un langage qui s'appuie sur le modèle de calcul Categorical Abstract Machine (CAM) qui lui confère une syntaxe fonctionnelle très proche du langage mathéma-

tique. OCaml est un langage interprété et compilé dont le typage est fort et statique. Les paradigmes de programmation d'OCaml sont les paradigmes fonctionnel, impératif et orienté objet.

Les points forts d'OCaml sont :

le typage fort et implicite OCaml est **fortement typé** : toute expression possède un type. Le typage est implicite : l'utilisateur n'a pas à le préciser car le compilateur OCaml utilise un algorithme d'**inférence de type** pour déterminer le type d'une expression. Le typage est statique et vérifié à la compilation : le couplage d'un typage fort et statique permet d'augmenter la performance du code et de rendre plus robuste le code face aux erreurs.

des structures de données muables et immuables OCaml propose des tableaux (Array), des structure de données mutables (tableaux, dictionnaires) mais aussi des structures immuables (listes). De nombreuses bibliothèques sont disponibles : files, tas, arbres…

un Garbage Collector un algorithme de gestion automatisée de la mémoire permet à OCaml de nettoyer les espaces mémoires qui ne sont plus utilisés par le programme en cours d'exécution. C'est important car les structures immuables engendrent en permanence la création et la destruction d'objets en mémoire.

la curryfication des fonctions OCaml permet de manipuler les fonctions comme des objets. Il permet également d'utiliser l'application partielle des fonctions (curryfication).

Dans la suite de ce chapitre, on pourra tester en même temps les expressions et les évaluer sur machine ou en ligne avec le bac à sable OCaml.

D Expressions et inférence de type

En tant que langage fonctionnel (et donc déclaratif), OCaml considère le calcul comme l'évaluation de fonctions mathématiques : OCaml traite donc des expressions qu'il évalue lorsqu'on fait suivre l'expression par `;;`.

Les types simples disponibles en OCaml sont

- `int` les entiers,
- `float` les flottants,
- `string` les chaînes de caractères,
- `bool` les booléens.

Voici un exemple d'évaluation d'un expression :

```
>>> 3 + 2*5 ;;
- : int = 13
```

Cet exemple montre bien qu'OCaml a inféré le type du résultat : il a écrit `int = 13`, c'est-à-dire que le résultat de l'évaluation de l'expression est un entier. Il faut noter qu'OCaml ne fait pas de transtypage implicite : pour lui, `3` est un entier et le restera, tout comme `3.5` est un flottant et le restera également. C'est pourquoi les opérateurs `+`,`*` et `/` qui additionne, multiplie ou divise les entiers ne sont pas les mêmes que ceux qui opèrent sur les flottants `+.`, `*.` ou `/.`.

D'ailleurs, l'inférence de type ne s'y trompe pas :

```
>>> 3.0 + 2.0*.5.0 ;;
Error : This expression has type float but an expression was expected of type int
```

Dans cet exemple, on a oublié d'utiliser l'opérateur `+.` pour additionner des flottants. Le compilateur s'en aperçoit car pour lui l'opérateur `+` n'accepte que deux opérandes entières et son résultat est un entier. Or, les opérandes fournies sont des flottants. Il détecte donc l'incohérence, la signale et ne peut pas continuer l'exécution.

En OCaml, un chou est un chou et restera un chou. Parole de léonard.

E Expressions locales et globales

Le mot clef `let` permet de définir une variable **globale**. Cette définition est une déclaration-initialisation. Une variable est toujours initialisée : cela permet au mécanisme d'inférence de type de fonctionner. À droite du symbole `=` doit se trouver une expression (ci-dessous `21 * 2`). À gauche du symbole `=` doit se trouver un identifiant (ci-dessous `x`).

```
>>> let x = 21 * 2;;
val x : int = 42
>>> x / 2;;
- : int = 21
```

Les mots clefs `let ... in` permettent de définir des variables **locales**. La portée de la variable ainsi définie est l'expression qui suit le mot clef `in`.

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> let budget = let depenses = 3500 in recettes - depenses;;
val budget : int = 500
>>> depenses * 2;;
Error
: Unbound value depenses
>>> recettes
- : int = 4000
```

OCaml définit une expression conditionnelle à l'aide de l'opérateur ternaire `if t then e1 else e2`. À la différence de son homologue en programmation impérative, cette expression **renvoie** un résultat. C'est pourquoi elle exige que les expressions `e1` et `e2` soient du même type.

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> let budget = let depenses = 3500 in recettes - depenses;;
val budget : int = 500
>>> let beneficiaire = if budget > 0 then true else false;;
val beneficiaire : bool = true
>>> let result = if beneficiaire then "Excellent !" else "Va falloir aller à la banque !"
    ;;
val result : string = "Excellent !"
```

Il faut bien noter que le symbole = pour déclarer une expression est utilisé conjointement à `let`. Il ne s'agit pas d'une affectation. D'ailleurs, un dernier exemple nous montre que l'opérateur = utilisé seul est un test d'égalité :

```
>>> let recettes = 4000;;
val recettes : int = 4000
>>> recettes = recettes + 400;;
- : bool = false
```

Enfin, on ne peut pas modifier le contenu d'une variable :

```
>>> let a = 3;;
     val a : int = 3
>>> a = 4 + 4;;
     - : bool = false
>>> a = a + 4;;
     - : bool = false
>>> let a = a + 3;;
     val a : int = 6
```

Pour modifier une variable, il faut donc la redéfinir ou utiliser le mécanisme de références (cf. section K).

O En OCaml, les variables sont donc immuables... ce qui est déconcertant au premier abord, car le propre d'une variable n'est-il pas de varier? La réponse est qu'en OCaml, comme en mathématiques, les variables sont en fait des expressions. Si on prend l'équation $y = 2x + 3$, l'idée de modifier la variable x n'existe pas vraiment en tant que telle : x va pouvoir varier dans le sens où l'on peut l'initialiser à différentes valeurs et que l'équation reste valable.

F Fonctions

Une fonction OCaml est une fonction au sens mathématique du terme et une expression paramétrée ou non. C'est pourquoi OCaml permet de la déclarer via le mot clef `let`.

```
>>> let perimeter r = 2. *. 3.1415926 *. r;;
val perimeter : float -> float = <fun>
>>> perimeter 1.;;
- : float = 6.2831852
```

On observe que le résultat est bien une fonction marquée par le mot-clef `fun`. Lors de l'évaluation de la déclaration de la fonction, OCaml nous délivre la signature associée à la fonction : `perimeter : float -> float = <fun>`. Celle-ci signifie que la fonction `perimeter` prend un paramètre de type `float` et renvoie un paramètre de type `float`.

Si la fonction est récursive, il faut le mentionner pour qu'OCaml en tienne compte :

```
>>> let rec explode n = if n = 0 then "Boum !" else  ". " ^ (explode (n - 1));;
val explode : int -> string = <fun>
>>> explode 3;;
- : string = ". . . Boum !"
```

O En ce qui concerne les fonctions OCaml :

- Les fonctions en OCaml renvoient la dernière valeur calculée. Il n'existe par de mot-clef `return` comme en Python.
- Les paramètres formels des fonctions ne sont pas délimités par des parenthèses.
- Les types des paramètres d'entrée et de sortie sont inférés automatiquement.

■ **Définition 23 — Curryfication d'une fonction ou application partielle.** La curryfication est l'opération de transformation une fonction à plusieurs arguments en une fonction à un argument qui retourne la fonction prenant le reste des arguments.

Enfin, OCaml permet également de curryfier les fonctions : l'application partielle d'une fonction est une fonction.

```
>>> let ajoute a b = a + b;;
val ajoute : int -> int -> int = <fun>
>>> let ajoute_deux = ajoute 2;;
val ajoute_deux : int -> int = <fun>
>>> ajoute_deux 4;;
- : int = 6
```

R La version curryfiée de la fonction est souvent un atout pratique en programmation fonctionnelle. On parle aussi d'**application partielle**. Dans l'exemple ci-dessus, l'expression `ajoute 2` est l'application partielle de la fonction `ajoute`.

Une fonction à n variables s'interprète donc soit :

1. comme une fonction à n variables,
2. une famille de fonctions à $n-1$ variables paramétrées par la première,
3. une famille de fonctions à $n-2$ variables paramétrées par les deux premières...

R Deux syntaxes sont possibles pour définir une fonction `f` de plusieurs variables :

1. la version ne permettant pas la curryfication : `let f (x,y)= ... ;;`
2. la version permettant la curryfication `let f x y = ... ;;`

G Effets

Vous aurez remarqué que cette introduction à OCaml n'a pas commencé par le traditionnel programme `"Hello Word!"` ! et il y a une bonne raison à cela : les langages fonctionnels n'aiment pas les effets de bords.

Vocabulary 2 — Side effect ⇜⇝ Effet secondaire mal traduit en français par effet de bord.

■ **Définition 24 — Effet (de bord) d'une expression.** Un effet de bord d'une expression est une action de celle-ci qui modifie l'état d'une variable en dehors de l'environnement local à la fonction : l'effet est donc quelque chose d'observable en dehors du fonctionnement standard d'une fonction, c'est-à-dire en dehors de la valeur retournée par la fonction.

■ **Exemple 6 — Effets de bord.** Quelques exemples classiques d'effets de bord :

- la modification d'une variable définie en dehors de la fonction.
- la modification d'une variable muable passée en paramètre à la fonction.
- l'appel d'une fonction qui produit des effets,
- toute entrée/sortie : l'impression d'une chaîne de caractère sur la console d'un écran d'ordinateur, la réception d'un message sur un socket réseau, la lecture ou l'écriture dans un fichier, l'interaction avec un système d'exploitation.

■ **Définition 25 — Expression pure et impure.** On dit qu'une expression est pure si elle n'engendre aucun effet de bord. Dans le cas contraire, on la dit impure.

R Dans l'esprit des concepteurs d'OCaml et des langages fonctionnels en général et pour des raisons de cohérence, **une fonction doit toujours renvoyer exactement le même résultat si elle est invoquée avec les mêmes paramètres en entrée**[a]. Or, si une fonction possède des effets de bords, il est possible que ce ne soit pas le cas.

a. Dans le cadre de la programmation des systèmes concurrents (microprocesseurs à architectures multicœurs), cette vision fonctionnelle est très importante car elle permet d'éviter de nombreux problèmes de synchronisation mémoire.

■ **Exemple 7 — Fonction impure en Python.** Dans l'exemple ci-dessous, la fonction `setn` est appelée deux fois avec le même paramètre 3 mais produit deux résultats différents. C'est donc une fonction impure.

```
x = 0

def setn(n):
    global x
    n = n + x
    return n

n = setn(3)
assert n == 3
x = 4
n = setn(3)
assert n == 3
```

Le programme `hello` ci-dessous est un exemple de fonction impure en OCaml. Écrire un message sur la console, c'est agir sur l'environnement d'exécution en dehors de la fonction. Le

risque est, par exemple, que la console ne réponde pas aux ordres du système d'exploitation. Dans ce cas, le programme ne s'exécute pas correctement à cause de l'effet de bord.

```
>>> let hello name = print_string ("Hello " ^ name ^"\n");;
val hello : string -> unit = <fun>
>>> hello "Olivier";;
Hello Olivier
- : unit = ()
```

La signature d'`hello` indique que la fonction renvoie un type `unit`.

■ **Définition 26 — Type unit.** Le type `unit = ()` représente le rien, le vide. Il est utilisé pour bien signifier qu'une fonction ne prend pas de paramètre ou ne renvoie rien. **Elle peut par contre avoir un effet.**

H Types algébriques

En plus des types simples, OCaml propose des mécanismes pour construire d'autres types éventuellement récursifs à partir des types simples :

- les types algébriques qui sont des composés
  - de types sommes qui sont des alternatives ou des énumérations,
  - et de types produits, des produits cartésiens de types,
- les record, ou enregistrements, qui permettent d'enregistrer une collection de types dans un même objet,
- les type optionnels qui par nature n'existent pas nécessairement.

■ **Définition 27 — Types sommes ou énumérations.** Un type somme est une alternative de types. Il est défini par l'alternance de ses constructeurs.

■ **Exemple 8 — Types sommes en OCaml.** Le type utilise l'alternative `|` et des constructeurs qui commencent par une majuscule.

```
>>> type chess_piece = Pawn | Knight | Bishop | Rook | Queen | King;;
type chess_piece = Pawn | Knight | Bishop | Rook | Queen | King
>>> let p = Pawn;;
val p : chess_piece = Pawn
```

■ **Définition 28 — Types produits.** Un type produit est un produit cartésien de types. Il engendre des tuples. Généralement, on ne nomme pas un type produit.

■ **Exemple 9 — Types produits en OCaml.** Le produit de type est réalisé par l'opérateur `*`.

```
>>> type point3d = float * float * float;;
type point3d = float * float * float
>>> let zero = 0.0, 0.0, 0.0;;
```

```
val zero : float * float * float = (0., 0., 0.)
```

■ **Définition 29 — Types enregistrements.** Un enregistrement est une collection de types nommés et enregistrés dans une même structure.

■ **Exemple 10 — Types enregistrements en OCaml.** Dans le domaine des services réseaux, on a souvent besoin d'identifier un service par son socket réseau qui est la combinaison de son adresse IP ou nom d'hôte, d'un numéro de port et d'un protocole. Cela peut se faire via un type enregistrement. Attention à la syntaxe, aux points virgules, deux points et symbole égal.

```
>>> type service_info =
.. {    service_name : string;
    ..     port         : int;
    ..     protocol     : string;
    .. };;
type service_info = { service_name : string; port : int; protocol : string; }
>>> let http_service = {service_name = "www"; port = 80; protocol = "http" };;
val http_service : service_info =
{service_name = "www"; port = 80; protocol = "http"}
>>> http_service.port;;
- : int = 80
```

■ **Définition 30 — Types optionnels.** Un type optionnel est un type qui peut être typé et posséder une valeur ou ne pas être typé ni posséder de valeur.

■ **Exemple 11 — Types options en OCaml.** En OCaml, les mots-clefs pour le type `option` sont `None` et `Some`.

```
>>> type zip_code = None | Some of int;;
type zip_code = None | Some of int
>>> let brest_code = Some 29200;;
val brest_code : zip_code = Some 29200
>>> let lost_city_code = None;;
val lost_city_code : zip_code = None
```

■ **Définition 31 — Types algébriques.** Un type algébrique est un type éventuellement récursif qui est une alternative de types éventuellement produit.

■ **Exemple 12 — Types algébriques.** Dans cet exemple, on construit un jeu de carte. Une première fonction permet d'obtenir la liste des figures d'une couleur donnée. Une autre la liste des cartes numéros pour une couleur donnée (à la belote!). Le mot-clef `of` permet de préciser un type de donnée associé au constructeur.

```
>>> type couleur = Coeur | Carreaux | Pique | Trefle;;
type couleur = Coeur | Carreau | Pique | Trefle
>>> type carte = As of couleur | Roi of couleur | Dame of couleur | Valet of couleur
    | Numero of int * couleur;;
type carte =
As of couleur
| Roi of couleur
| Dame of couleur
| Valet of couleur
| Numero of int * couleur
>>> let figures_de c = [As c; Roi c; Dame c; Valet c];;
val figures_de : couleur -> carte list = <fun>
>>> figures_de Pique;;
- : carte list = [As Pique; Roi Pique; Dame Pique; Valet Pique]
>>> let numero_de c = [10,c;9,c;8,c;7,c];;
val numero_de : 'a -> (int * 'a) list = <fun>
>>> numero_de Coeur;;
- : (int * couleur) list = [(10, Coeur); (9, Coeur); (8, Coeur); (7, Coeur)]
```

I Listes

Le type liste OCaml est un type récursif immuable.

R **Immuable** signifie qu'on ne peut pas le modifier. Pour faire évoluer une liste, c'est-à-dire pour ajouter ou retirer des éléments, il est donc nécessaire de créer une autre liste.

O Une liste en OCaml ne contient qu'un seul type de données.

■ **Définition 32 — Définition inductive des listes.** Une liste est soit une liste vide soit un élément suivi d'une liste.

L'ensemble des listes $\mathcal{L}$ à valeur dans $\mathcal{E}$ est donc définie par :

Base la liste vide `[]` est une liste

Constructeur : : $\forall L \in \mathcal{L}, \forall e \in \mathcal{E}, e :: l \in \mathcal{L}$

Ce qui en OCaml donne :

```
type 'a list =
  | []
  | (::) of 'a * 'a list     (* :: est l'opérateur ajouter en tête *)
```

L'expression `'a` désigne un type quelconque. Donc on peut construire une liste de n'importe quel type d'objet.

O Quelques remarques sur les opérations sur les listes en OCaml :

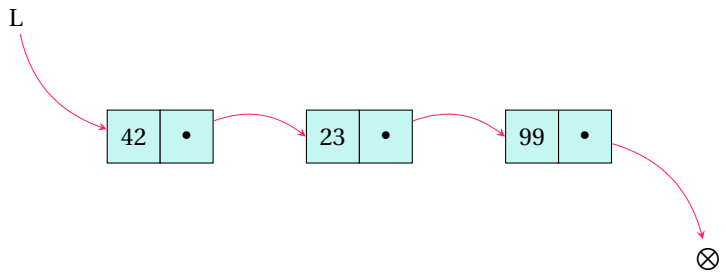


FIGURE 2.2 – Représentation d'une liste d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 2.1 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

- pour supprimer un élément d'une liste, il faut en construire une autre.
- pour ajouter un élément à une liste, il faut en construire une autre.

Ces opérations ont donc un coût comme l'illustre le tableau 5.3.

■ **Exemple 13 — Liste en OCaml.** On peut construire une liste OCaml directement en initialisant les valeurs des éléments :

```
>>> let l = [1;2;3;4;5];;
val l : int list = [1; 2; 3; 4; 5]
```

Le constructeur : : permet d'ajouter un élément en tête de liste. Par exemple :

```
>>> let l = 42::21::14::7::l;;
val l : int list = [42; 21; 14; 7; 1; 2; 3; 4; 5]
>>> let wrong = 0.2::l;;
Line 1, characters 17—18:
|1 | let wrong = 0.2::l;;
                     ^
Error: This expression has type int list but an expression was expected of type float
    list. Type int is not compatible with type float
```

Le type d'objet contenu dans une liste est nécessairement toujours le même : dans l'exemple ci-dessus, le type de la liste est `int list`, une liste d'entiers.

On ne peut accéder directement [a] qu'au premier élément. L'accès aux autres éléments nécessite de balayer la liste. Généralement, comme c'est une structure inductive, on procède par déconstruction : on prend la tête de liste que l'on traire et on garde le reste de la liste.

```
>>>let head = List.hd l;;
val head : int = 42
>>> let tail = List.tl l;;
val tail : int list = [21; 14; 7; 1; 2; 3; 4; 5]
```

---

a. en temps constant O(1)

J Filtrage de motif

Le filtrage de motif est une technique puissante pour gérer les types algébriques et les ensembles inductifs.

■ **Définition 33 — Filtrage de motif.** Le filtrage de motif est l'action de tester une expression pour détecter sa constitution dans le but de faire un calcule en fonction du motif détecté.

```
let f x =
    match x with
       | motif_1  > expression 1 (* traitement de ce cas *)
       | motif_2  > expression 2 (* traitement de ce cas *)
       .
       .
       | motif_n  >  expression n (* traitement de ce cas *);;
```

L'expression `motif` est une **constante** ou un **constructeur** de types. L'ensemble des motifs décrits doit être **exhaustif**.

Vocabulary 3 — Pattern matching ⇜⇝ Filtrage de motif

■ **Exemple 14 — Filtrage simple et exhaustif.** Considérons la fonction qui teste la parité d'un nombre entier. On l'écrirait, dans un langage impératif, avec une structure conditionnelle `if then else`. Il est possible de l'écrire ainsi en OCaml, mais on peut également utiliser le filtrage de motif comme suit :

```
let is_even n =
    match n mod 2 with
        | 0 -> true
        | _ -> false ;;
```

Ce filtrage est bien exhaustif : le second cas `_` englobe tous les cas possibles autre que 0.

■ **Exemple 15 — Filtrage de motif en OCaml.** Le code ci-dessous définit un type algébrique `shape` puis une fonction qui calcule l'aire en fonction du type `shape` passé en paramètre. L'aire du triangle est calculée avec la formule de Héron.

```
type shape = Circle of float | Square of float | Triangle of (float * float * float);;

let area s =
match s with
    | Circle r -> Float.pi *. (r ** 2.0)
    | Square s -> s ** 2.0
    | Triangle (a, b, c) -> let s = (a +. b +. c) /. 2.0 in sqrt (s *. (s -. a) *. (s
        -. b) *. (s -. c));;

area (Triangle (3.0, 4.0, 5.0));;
```

■ **Exemple 16 — Filtrage de motif et liste en OCaml.** La fonction ci-dessous est récursive et calcule la somme des éléments d'une liste. Elle déconstruit au fur et à mesure la liste pour additionner ses éléments.

```
let rec sum l =
    match l with
        | [] -> 0
        | head :: tail -> head + sum tail;;
let l = 42::21::14::7::l;;
sum l;;
```

Le motif `[]` représente une liste vide. Si ce motif est détecté, alors la fonction renvoie 0, car la somme des éléments d'une liste vide vaut 0.

Sinon, il y a au moins un élément en tête de liste `head` suivi par une sous-liste `tail` éventuellement vide. Alors on renvoie la valeur `head` ajoutée à la somme du reste de la liste.

Dans cette exemple emblématique, la liste `l` n'est pas nécessairement détruite en mémoire, mais on la déconstruit d'un élément à chaque appel récursif en créant une autre liste `tail` qui est une partie de `l`. L'opération est de complexité linéaire.

■ **Exemple 17 — Filtrage de motif conditionnel.** Il est possible de distinguer des motifs identiques en construction mais différents dans les valeurs. On désigne cette opération par filtrage conditionnel.

```
let fcond n m b =
    match (n,b) with
        | (_,false) -> 0
        | (k, true) when k = m -> 1
        | (k, true) -> 2;;
```

■ **Exemple 18 — Filtrage non exhaustif.** Il n'est pas rare d'oublier de lister un motif possible. Dans ce cas, OCaml le signale lors de l'évaluation. Par exemple, le code suivant ne filtre pas tous les motifs de manière exhaustive :

```
let fcond n m b =
    match (n,b) with
        | (k, true) when k = m -> 1
        | (k, true) -> 2;;
```

C'est pourquoi l'avertissement `partial_match` suivante est émis par l'interprète OCaml :

```
Warning 8 [partial_match]: this pattern_matching is not exhaustive.
      |Here is an example of a case that is not matched:
      |(_, false)
```

O Les objets immuables comme les listes nécessitent donc un espace mémoire adapté : il faut être capable de créer des listes à volonté. Le Garbage Collector, en cours d'exécution du programme, collecte les objets (comme les listes) non utilisés puis libère l'espace mémoire associé. Il garantit ainsi un exécution correcte, sans encombrement mémoire dû à la création multiple d'objets par les fonctions et les constructeurs.

K Références et programmation impérative en OCaml

OCaml est un langage mutliparadigme et on peut programmer de manière impérative avec effet. On peut regretter une certaine lourdeur de la syntaxe par rapport à Python mais les exigences des deux langages n'ont rien à voir non plus : Python est un langage laxiste [1], OCaml ne l'est pas [2].

O Le mécanisme pour pouvoir modifier une variable est nommé **référence**. La référence en OCaml est ce qui permet de manipuler une variable comme on l'entend dans les langages impératifs, c'est-à-dire une variable **muable**.

1. ce qui est pratique pour le prototypage d'applications et le calcul numérique
2. ce qui est important pour le développement d'applications fiables, robustes et performantes.

Pour référencer une variable, il suffit de le déclarer avec le mot-clef `ref` suivi de la valeur d'initialisation. Pour utiliser la valeur d'une référence, il faut la précéder d'un point d'exclamation !. Enfin, pour affecter une nouvelle valeur à une variable, il faut utiliser l'opérateur `:=`.

■ **Exemple 19 — Déclaration, initialisation et utilisation d'une référence en OCaml.** Lorsqu'on affecte une nouvelle valeur à une référence en OCaml, l'opération renvoie `unit`. **Une affectation est un effet de bord**.

```
>>> let x = ref 4;;
val x : int ref = {contents = 4}
>>> let y = 3 + !x;
val y : int = 7
>>> x := 0;;
- : unit = ()
>>> x
- : int ref = {contents = 0}
>>> !x
-: int = 0
>>> y
- : int = 7
```

OCaml propose également les structures itératives `for` et `while`. Les tableaux (Array) sont des structures impératives, tout comme les dictionnaires.

■ **Exemple 20 — Boucle et tableau en OCaml.** Dans cet exemple, un tableau initialisé à zéro est créé. Puis ses valeurs sont modifiées en utilisant une boucle `for`. On notera que l'accès à un élément su tableau se fait par `.(i)` et l'affectation de la nouvelle valeur par `<-`.

```
>>> a
- : int array = [|0; 1; 2; 3; 4|]
>>> let a = Array.make 5 0;
val a : int array = [|0; 0; 0; 0; 0|]
>>> for i = 0 to (Array.length a - 1) do a.(i) <- a.(i) + i done;;
- : unit = ()
>>> a
- : int array = [|0; 1; 2; 3; 4|]
```

O En OCaml, chaque type possède ses propres opérateurs et fonctions. Le compilateur peut générer facilement des messages compréhensibles. L'approche est rigoureuse.

O Même si les boucles existent en OCaml, si on veut s'en tenir au paradigme fonctionnel du langage, il est préférable de les éviter car celles-ci s'accompagnent le plus souvent de références et d'effets de bords. Dans ce cadre, on lui préfère une approche récursive.

L Synthèse

O En OCaml tout est expression à évaluer. Les types revêtent une importance capitale. On distingue :

- les types simples (`int`, `float`, `bool`, `char`, `string`)
- les types algébriques :
  - les types sommes `Pique | Trefle`
  - les types produits `int*char` et les enregistrrements `ey : int, value : floatkey : int , value: float`.

Ces types sont inférés automatiquement par l'interprète OCaml. Ils peuvent être récursifs. Les variables `let x ...` sont immuables. Le mécanisme des références permet de contourner cette limitation.

O En ce qui concerne les fonctions OCaml :

- Les fonctions en OCaml renvoient la dernière valeur calculée. Il n'existe par de mot-clef `return` comme en Python.
- Les paramètres formels des fonctions ne sont pas délimités par des parenthèses.
- Les types des paramètres d'entrée et de sortie sont inférés automatiquement.
- La signature de la fonction est systématiquement calculée par OCaml.

O Le filtrage de motif est un outil central qui couplé aux types algébriques et aux ensembles inductifs permet l'écriture de codes puissants, expressifs et lisibles.

Deuxième partie

Logique

3

DE LA LOGIQUE AVANT TOUTE CHOSE

À la fin de ce chapitre, je sais :

- ☞ formuler des propositions logiques à partir du langage naturel
- ☞ utiliser les connecteurs logiques pour relier des variables
- ☞ établir une table de vérité
- ☞ utiliser les lois de Morgan, le tiers exclu et la décomposition de l'implication
- ☞ mettre une formule propositionnelle sous une forme normale
- ☞ étudier la satisfaisabilité d'une formule propositionnelle

A Logique et applications

Quelque soit le sujet, les êtres humains ont tendance à chercher s'assurer qu'ils ont raison ou que les autres ont tord, ne serait-ce que pour des raisons d'ego [1] ou de commerce [2]. Dans le cadre de la science et de l'ingénierie, il est fondamental de pouvoir apporter la preuve que le raisonnement tenu est correct, car c'est ainsi que la science progresse collectivement et ainsi que l'ingénierie garantit le bon fonctionnement de l'objet produit. Aujourd'hui,tous les domaines de l'industrie sont dépendants de la logique. On peut citer par exemple : la planification (logistique, organisation des tâches), la satisfaction de contraintes multiples, l'élaboration de diagnostiques, la vérification formelle de modèles de systèmes complexes ou l'élaboration et vérification des circuits électroniques.

Une démarche scientifique exige un raisonnement formalisé : c'est l'objet de la logique et du calcul propositionnel qui est présenté dans ce chapitre. Ce formalisme est né au milieu du XIX^{e} siècle grâce au travaux de Boole et a été finalisé au cours de la première moitié du XX^{e} siècle par Frege, Russel et Gödel.

1. D'abord, j'ai toujours raison!
2. Mon collègue est-il en train de m'arnaquer?

■ **Définition 34 — Logique.** Du grec *logos*, la raison. La logique en tant que discipline scientifique fournit les outils nécessaires à la construction d'un raisonnement : elle permet de manipuler des concepts et d'enchaîner les déductions. Les vérités logiques ne concernent aucun domaine de connaissance en particulier : elles sont valides en amont de toute vérité scientifique particulière empirique [a].

a. c'est-à-dire issue de l'expérience du monde réel

■ **Exemple 21 — Structure d'un raisonnement.** Considérons l'énoncé suivant :

> Si un professeur est un super-héros et que je suis un professeur, alors je suis un super-héros.

Cet énoncé est vrai et inspire une observation fondamentale : la notion de vérité que l'on peut lui associer ne repose que sur sa structure, sa forme, c'est-à-dire Si ... et ..., alors .... On pourrait en effet remplacer le terme *professeur* par *élève*, le terme *super-héros* par *star* et le terme *je* par *tu* et malgré tout, l'énoncé serait vrai.

> Si un élève est une star et que tu es un élève, alors tu es une star.

On peut donc **construire** des énoncés logiques formels à l'aide de connecteurs : Si, et, ou, alors, ne ... pas,donc.... Par exemple, on peut combiner :

p_1 le ciel est bleu

et

p_2 je me promène au bord de l'océan

ainsi :

p_3 Si le ciel est bleu, alors je me promène au bord de l'océan.

Formellement, on peut modéliser cet énoncé logique ainsi : $p_3 = (p_1 \longrightarrow p_2)$.

■ **Définition 35 — Proposition simple ou atomique.** Une proposition simple est une expression vraie ou fausse.

La logique des propositions est limitée : elle ne peut pas prendre en compte des énoncés quantifiés [3] comme dans l'exemple 22.

■ **Exemple 22 — Prédicat et raisonnement quantifié** ⇢ **Hors Programme** . Considérons l'énoncé suivant :

> Si tous les professeurs sont des super-héros et que je suis un professeur, alors je suis un super-héros.

La logique des propositions ne peut pas modéliser cet énoncé à cause du quantificateur universel *tous les*. En effet, la valeur de vérité de cette proposition dépend de ce quantificateur. Que se passe-t-il si un professeur n'est pas un super-héros ? Ce n'est plus la même

3. On parle alors de logique du premier ordre, logique des prédicats ⇢ HORS PROGRAMME

proposition. Une proposition simple ne contient donc pas de quantificateur. Pour modéliser ce type d'énoncé, on utilise les prédicats, ceux que vous manipulez en mathématiques : supposons que S est le prédicat unaire *super-héros*, P le prédicat unaire *professeur*. On peut écrire le prédicat suivant :

$$(\forall x, P(x) \longrightarrow S(x)) \wedge P(x) \longrightarrow S(x)$$

La suite de ce chapitre expose donc la logique des propositions. Elle débute par la définition des briques de bases que sont les constantes, les variables propositionnelles et les opérateurs logiques. Puis elle énonce la syntaxe de la logique propositionnelle en donnant la définition inductive des formules logiques. La définition de la valuation d'une formule permet de donner un sens à la logique propositionnelle et de définir une sémantique. Enfin, le chapitre aborde les problèmes de satisfaisabilité d'une formule logique.

B Constantes, variables propositionnelles et opérateurs logiques

Dans cette section est détaillé les éléments de l'ensemble qui forme la base des formules logiques.

■ **Définition 36 — Constante universelle $\top$.** La constante $\top$ désigne le vrai. On peut la voir comme un opérateur d'arité nulle.

■ **Définition 37 — Constante vide $\bot$.** La constante $\bot$ désigne la contradiction. On peut la voir comme un opérateur d'arité nulle.

Vocabulary 4 — Top et Bottom ⇜⇝ En anglais, la constante universelle $\top$ se dit *Top* et la constante vide $\bot$ Bottom.

■ **Définition 38 — Variable propositionnelle.** Une variable propositionnelle est une proposition atomique (cf. définition 35).

C Opérateurs logique et notations

L'étude des structures des propositions permet de définir opérateurs qui relient les formules logiques : la négation *non*, le *et*, le *ou*, l'*implication* et l'*équivalence*.

■ **Définition 39 — Opérateur.** Un opérateur est une fonction qui réalise une opération primitive dans un langage.

■ **Exemple 23 — Opérateurs.** Les symboles mathématiques $+, -, \times$ sont des opérateurs pour l'arithmétique des entiers naturels : $2+3\times5$. On utilise également ces symboles dans le cas de l'arithmétique des polynômes : $P+Q$. Cependant, il faut bien observer que ce ne sont pas les mêmes opérateurs.

De la même manière, en informatique, les symboles `+`, `-`, `*` représentent les opérateurs arithmétiques sur les entiers en Python. Même si on utilise les mêmes symboles pour faire les opérations de base sur les flottants, il faut bien remarquer que le langage offre ici ce qu'on appelle du sucre syntaxique, une facilité. Car ce sont en fait des opérateurs différents. D'ailleurs, en OCaml, les opérateurs arithmétiques sur les entiers `+`, `-`, `*` ne sont pas les opérateurs sur les flottants `+.`, `-.`, `*.`.

■ **Définition 40 — Arité d'un opérateur.** L'arité d'un opérateur est le nombre de paramètres que prend sa fonction sous-jacente. On distingue les opérateurs unaires (une seule opérande) et les opérateurs binaires (deux opérandes).

■ **Exemple 24 — Arités des opérateurs arithmétiques.** On dit que l'addition ou la multiplication sont des opérateurs binaires car ils prennent deux opérandes en entrée.

En informatique, on distingue l'opérateur binaire `-` de l'opérateur unaire `-`. Le premier réalise la soustraction des deux opérandes `a - b`, le second change le signe de l'opérande `-a`.

■ **Définition 41 — Notation infixe des opérateurs (connecteurs).** Dans le cadre d'une notation infixe des opérateurs binaires, l'opérateur est placé entre ses opérandes.

■ **Définition 42 — Notation préfixe des opérateurs (constructeurs).** Dans le cadre d'une notation préfixe des opérateurs binaires, l'opérateur est placé devant ses opérandes.

■ **Définition 43 — Négation (NON).** L'opérateur négation consiste à calculer la valeur opposée de son opérande.

- Connecteur : $\neg$
- Constructeur : `not`
- Notation informatique fréquente : not ou !

■ **Définition 44 — Conjonction (ET).** L'opérateur conjonction a pour résultat vrai si ses deux opérandes sont vraies.

- Connecteur : $\wedge$
- Constructeur : `and`
- Notation informatique fréquente : `&&` ou `and`

■ **Définition 45 — Disjonction (OU).** L'opérateur disjonction a pour résultat vrai si une des deux opérandes est vraies.

- Connecteur : $\vee$
- Constructeur : `or`
- Notation informatique fréquente : `||` ou `or`

R Les opérateurs négation, conjonction et disjonction sont les opérateurs premiers : ils permettent d'exprimer tous les autres.

■ **Définition 46 — Implication matérielle ($\Longrightarrow$).** L'opérateur implication logique a pour résultat faux seulement si la seconde opérande est fausse.

- Connecteur : $\Longrightarrow$
- Constructeur : `imp`

R L'implication matérielle est le seul opérateur de base en logique minimale.

■ **Définition 47 — Équivalence matérielle ($\Longleftrightarrow$).** L'opérateur équivalence matérielle est équivalent à une implication dans les deux sens.

- Connecteur : $\Longleftrightarrow$
- Constructeur : `eq`

■ **Exemple 25 — Notation infixe et préfixe d'une même formule logique.** Voici une même formule logique décrite à l'aide de variables propositionnelles et d'opérateurs infixes ou préfixes.

- avec les connecteurs : $(c \Longrightarrow b) \vee (\neg c \wedge b)$
- avec les constructeurs : or(imp(c,b),et(not(c), b))

Même s'il est possible de programmer des connecteurs, les informaticiens préfèrent les constructeurs pour plusieurs raisons :

- ils sont plus faciles à implémenter,
- ils donnent à l'ensemble des formules une structure arborescente facile à analyser.

R Les opérateurs possèdent un ordre de priorité, de la plus forte à la plus faible :

$$\neg > \wedge > \vee$$

Cela signifie qu'en cas d'ambiguïté sur une formule sans parenthèses, cet ordre permet trancher l'interprétation : on effectue d'abord la négation puis la conjonction et enfin la disjonction.

D Formules logiques : définition inductive et syntaxe

On considère un ensemble des variables propositionnelles $\mathcal{V}$ utilisé pour écrire un ensemble de formules $\mathcal{F}$ en logique propositionnelle.

■ **Définition 48 — Ensemble des formules propositionnelles $\mathcal{F}$ (défini inductivement).** L'ensemble $\mathcal{F}$ des formules propositionnelles sur $\mathcal{V}$ est défini inductivement comme suit :

$$\bot \in \mathcal{F} \quad \text{(Base)} \tag{3.1}$$
$$\top \in \mathcal{F} \quad \text{(Base)} \tag{3.2}$$
$$\forall x \in \mathcal{V}, x \in \mathcal{F} \quad \text{(Base)} \tag{3.3}$$
$$\forall \phi \in \mathcal{F}, \text{not}(\phi) \in \mathcal{F} \quad \text{(Constructeur négation)} \tag{3.4}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \text{and}(\phi, \psi) \in \mathcal{F} \quad \text{(Constructeur conjonction)} \tag{3.5}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, \text{or}(\phi, \psi) \in \mathcal{F} \quad \text{(Constructeur disjonction)} \tag{3.6}$$

Cela signifie qu'une formule logique est soit :

- une constante universelle ou vide,
- une variable propositionnelle,
- une négation d'une formule logique,
- une conjonction ou une disjonction de formules logiques.

R L'intérêt principal de la définition inductive est qu'elle permet de **construire des formules logiques qui sont des objets informatiques avec lesquels on peut calculer**.

■ **Définition 49 — Formule atomique.** Une formule ϕ est atomique si ϕ est $\bot$, $\top$ ou une variable propositionnelle.

R Une formule logique de $\mathcal{F}$ peut être représentée par un arbre comme l'illustre la figure 3.1 : cette forme est très importante pour les compilateurs et les outils d'analyse de code en général. On la désigne par le terme arbre syntaxique. En parcourant un tel arbre, on peut calculer une formule logique. C'est pourquoi, la structure d'arbre est au programme et sera détaillée dans les prochains chapitres.

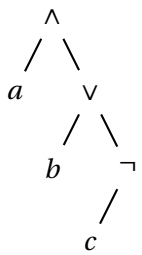


FIGURE 3.1 – Arbre représentant la formule logique $a \wedge (b \vee \neg c)$

■ **Définition 50 — Ensemble des sous-formules.** Soit ϕ une formule logique et $\mathcal{V}$ l'ensemble des ses variables propositionnelles. On définit l'ensemble des sous-formules de ϕ par la fonction sf :

$$\forall \phi \in \{\bot, \top, \} \cup \mathcal{V}, sf(\phi) = \phi \quad \text{(Base)} \tag{3.7}$$

$$\forall \phi \in \mathcal{F}, \exists \psi \in \mathcal{F}, \phi = \neg\psi \Longrightarrow sf(\phi) = \{\phi\} \cup sf(\psi) \quad \text{(Constructeur négation)} \tag{3.8}$$

$$\forall \phi \in \mathcal{F}, \exists \psi, \xi \in \mathcal{F}, \phi = \psi \wedge \xi \Longrightarrow sf(\phi) = \{\phi\} \cup sf(\psi) \cup sf(\xi) \quad \text{(Constructeur conjonction)} \tag{3.9}$$

$$\forall \phi \in \mathcal{F}, \exists \psi, \xi \in \mathcal{F}, \phi = \psi \vee \xi \Longrightarrow sf(\phi) = \{\phi\} \cup sf(\psi) \cup sf(\xi) \quad \text{(Constructeur disjonction)} \tag{3.10}$$

Une constante ou une variable propositionnelle est une sous-formule. Toutes les formules qui permettent de construire une formule logique sont des sous-formules.

■ **Définition 51 — Taille d'une formule.** La taille d'une formule logique est le nombre d'opérateurs qui construisent la formule. On la note $|\phi|$ et on la définit inductivement par :

$$|\bot| = 0 \quad \text{(Base)} \tag{3.11}$$

$$|\top| = 0 \quad \text{(Base)} \tag{3.12}$$

$$\forall x \in \mathcal{V}, |x| = 0 \quad \text{(Base)} \tag{3.13}$$

$$|\neg\phi| = 1 + |\phi| \quad \text{(Constructeur négation)} \tag{3.14}$$

$$|\phi \diamond \psi| = 1 + |\phi| + |\psi| \quad \text{(Constructeur conjonction ou disjonction } \diamond) \tag{3.15}$$

■ **Définition 52 — Hauteur d'une formule.** La hauteur d'une formule logique est la hauteur de son arbre syntaxique. On la note $h(\phi)$ et elle est définie inductivement :

$$h(\bot) = 0 \quad \text{(Base)} \tag{3.16}$$

$$h(\top) = 0 \quad \text{(Base)} \tag{3.17}$$

$$\forall x \in \mathcal{V}, h(x) = 0 \quad \text{(Base)} \tag{3.18}$$

$$h(\neg\phi) = 1 + h(\phi) \quad \text{(Constructeur négation)} \tag{3.19}$$

$$h(\phi \diamond \psi) = 1 + \max(h(\phi, h(\psi)) \quad \text{(Constructeur conjonction ou disjonction } \diamond) \tag{3.20}$$

E Sémantique et valuation des formules logiques

■ **Définition 53 — Ensemble des valeurs de vérité.** L'ensemble des valeurs de vérité est un ensemble à deux éléments que l'on peut noter de différentes manières :

$$\mathbb{B} = \{0,1\} = \{F,V\} = \{\text{Faux},\text{Vrai}\} = \{\text{F},\text{T}\} \tag{3.21}$$

■ **Définition 54 — Valuation ou interprétation.** Une valuation de $\mathcal{V}$ est une distribution des valeurs de vérité sur l'ensemble des variables propositionnelles $\mathcal{V}$, soit une fonction $v : \mathcal{V} \longrightarrow \mathbb{B}$.

■ **Définition 55 — Évaluation d'une formule logique (définie inductivement).** Soit v une valuation de $\mathcal{V}$: $v : \mathcal{V} \longrightarrow \mathbb{B}$. L'évaluation d'une formule logique d'après v est notée $[\![\phi]\!]_v$. Elle est définie inductivement par :

$$[\![\bot]\!]_v = \text{F} \qquad \text{(Base)} \tag{3.22}$$
$$[\![\top]\!]_v = \text{V} \qquad \text{(Base)} \tag{3.23}$$
$$\forall x \in \mathcal{V}, [\![x]\!]_v = v(x) \qquad \text{(Base)} \tag{3.24}$$
$$\forall \phi \in \mathcal{F}, [\![\neg\phi]\!]_v = \neg[\![\phi]\!]_v \qquad \text{(Constructeur négation)} \tag{3.25}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \wedge \psi]\!]_v = [\![\phi]\!]_v \wedge [\![\psi]\!]_v \qquad \text{(Constructeur conjonction)} \tag{3.26}$$
$$\forall \phi \in \mathcal{F}, \forall \psi \in \mathcal{F}, [\![\phi \vee \psi]\!]_v = [\![\phi]\!]_v \vee [\![\psi]\!]_v \qquad \text{(Constructeur disjonction)} \tag{3.27}$$
$$\tag{3.28}$$

Pour être capable d'évaluer une formule logique, il faut donc pouvoir évaluer des négations, des conjonctions et des disjonctions. Les tables de vérités des opérateurs premiers sont données sur les tableaux 3.1, 3.2, 3.3, 3.4 et 3.5.

a	$\neg a$
F	V
V	F

TABLE 3.1 – Table de vérité de l'opérateur négation

a	b	$a \wedge b$
F	F	F
F	V	F
V	F	F
V	V	V

TABLE 3.2 – Table de vérité de l'opérateur conjonction

a	b	$a \vee b$
F	F	F
F	V	V
V	F	V
V	V	V

TABLE 3.3 – Table de vérité de l'opérateur disjonction

a	b	$a \Longrightarrow b$
F	F	V
F	V	V
V	F	F
V	V	V

TABLE 3.4 – Table de vérité de l'opérateur implication matérielle

a	b	$a \Longleftrightarrow b$
F	F	V
F	V	F
V	F	F
V	V	V

TABLE 3.5 – Table de vérité de l'opérateur équivalence matérielle

■ **Définition 56 — Modèle.** Un modèle pour une formule logique ϕ est une valuation v telle que :

$$[\![\phi]\!]_v = \mathrm{V} \tag{3.29}$$

■ **Définition 57 — Conséquence sémantique $\models$.** Soit ϕ et ψ deux formules de $\mathcal{F}$. On dit que ψ est une conséquence sémantique de ϕ si tout modèle de ϕ est un modèle de ψ. On note alors : $\phi \models \psi$

Une formule ψ peut également être la conséquence sémantique d'un ensemble de formules Γ. On note alors : $\Gamma \models \psi$.

■ **Définition 58 — Équivalence sémantique $\equiv$.** Deux formules logiques ϕ et ψ sont équivalentes sémantiquement si quelle que soit la valuation choisie, l'évaluation des deux formules produit le même résultat. On note cette équivalence $\phi \equiv \psi$.

Plus formellement,

$$\phi \equiv \psi \iff \forall v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = [\![\psi]\!]_v \tag{3.30}$$

■ **Définition 59 — Tautologie.** Une formule ϕ toujours vraie quel que soit le modèle d'interprétation est une tautologie. On la note $\top$ ou $\models \phi$.

R Deux formules sont donc équivalentes d'un point de vue sémantique si et seulement si leur équivalence matérielle est une tautologie : $\models (\phi \iff \psi)$.

■ **Définition 60 — Antilogie.** Une formule a toujours fausse quel que soit le modèle d'interprétation est une antilogie. On la note $\bot$.

■ **Définition 61 — Formule satisfaisable.** Soit ϕ une formule logique de $\mathcal{F}$. S'il existe une valuation v de $\mathcal{V}$ qui satisfait ϕ, alors ϕ est dite satisfaisable.

Plus formellement :

$$\phi \text{ est une formule satisfaisable} \iff \exists v : \mathcal{V} \longrightarrow \mathbb{B}, [\![\phi]\!]_v = \mathrm{V} \tag{3.31}$$

F Lois de la logique

Les lois de la logique sont des équivalences sémantiques.

a Éléments neutres et absorbants, idempotence

$$a \wedge \top \equiv a \tag{3.32}$$
$$a \wedge \bot \equiv \bot \tag{3.33}$$
$$a \vee \top \equiv \top \tag{3.34}$$
$$a \vee \bot \equiv a \tag{3.35}$$
$$a \wedge a \equiv a \tag{3.36}$$
$$a \vee a \equiv a \tag{3.37}$$

b Commutativité, distributivité, associativité

$$a \wedge b \equiv b \wedge a \tag{3.38}$$
$$a \vee b \equiv b \vee a \tag{3.39}$$
$$a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c) \tag{3.40}$$
$$a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c) \tag{3.41}$$
$$a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c \tag{3.42}$$
$$a \vee (b \vee c) \equiv (a \vee b) \vee c \tag{3.43}$$

c Lois de De Morgan

$$\neg(a \wedge b) \equiv \neg a \vee \neg b \tag{3.44}$$
$$\neg(a \vee b) \equiv \neg a \wedge \neg b \tag{3.45}$$

d Décomposition des opérateurs

$a \Longrightarrow b \equiv \neg a \vee b$	Implication et opérateurs premiers	(3.46)
$a \Longrightarrow b \equiv \neg b \Longrightarrow \neg a$	Contraposition	(3.47)
$(a \wedge b) \Longrightarrow c \equiv a \Longrightarrow (b \Longrightarrow c)$	Curryfication	(3.48)

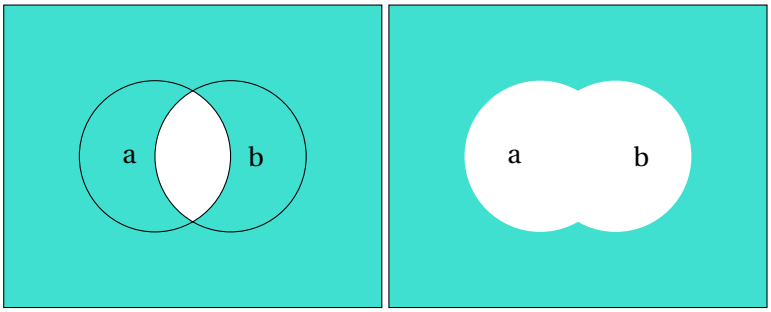


FIGURE 3.2 – Illustration des lois de De Morgan

e Démonstrations

Pour démontrer les lois précédentes, on peut produire les tables de vérités correspondantes puis utiliser les lois déjà prouvées pour démontrer les autres. Ces démonstrations constituent d'excellents exercices.

G Principes et logique classique

$a \vee \neg a \equiv \top$	Principe du tiers-exclus	(3.49)
$a \wedge \neg a \equiv \bot$	Principe de non-contradiction	(3.50)
$\neg\neg a \equiv a$	Double négation	(3.51)
$\bot \Longrightarrow a$	Principe d'explosion	(3.52)
		(3.53)

R Il existe plusieurs logiques qui se différencient principalement par la manière de gérer les déductions de l'absurde :

- La logique minimale n'utilise qu'un seul connecteur : l'implication. Elle a pour caractéristique de ne rien déduire de $\bot$. Cela signifie qu'elle n'inclut ni le principe du tiers-exclus, ni le principe d'explosion.
- La logique classique utilise les opérateurs définis dans ce cours. Elle a pour caractéristique d'inclure les principes ci-dessus et permet donc de conduire des raisonnements par l'absurde. La critique faite à ce type de raisonnement, c'est qu'il permet d'accepter l'existence d'un concept sans pouvoir le construire explicitement.
- La logique intuitionniste est constructive : la notion de preuve constructive remplace la notion de vérité. Constuire un concept, c'est exhiber sa preuve d'existence. Si un concept ne peut pas être établi par une preuve constructive, cela signifie qu'il n'existe pas. La

logique intuitionniste distingue le *être vrai* du *ne pas être faux*. C'est pourquoi elle n'inclut ni le principe du tiers exclus, ni le raisonnement par l'absurde, ni la double négation. Elle inclut par contre le principe d'explosion.

H Formes normales

■ **Définition 62 — Littéral.** Un littéral est une variable propositionnelle ou sa négation.

■ **Définition 63 — Clause conjonctive.** Une clause conjonctive est une conjonction de littéraux.

■ **Définition 64 — Forme normale disjonctive(FND).** Une forme normale disjonctive d'une formule logique est une disjonction de clauses conjonctives.

Théorème 6 — Toute formule logique est équivalente à une forme normale disjonctive.

Démonstration. En fait, il suffit d'écrire que cette formule logique est la disjonction de toutes ses valuations vraies. Plus formellement :

$$\phi \equiv \bigvee_{\substack{\nu, \\ [\![\phi]\!]_\nu = \mathrm{V}}} \bigwedge_{x \in \mathcal{V}} x \tag{3.54}$$

■

■ **Exemple 26 — Lien entre la table de vérité et la forme disjonctive complète.** On considère la formule logique $\phi = (a \vee b) \wedge ((c \implies b) \vee a)$. Sa table de vérité contient l'ensemble possible de ses valuations vraies et fausses :

a	b	c	$a \vee b$	$c \implies b$	$(c \implies b) \vee a$	ϕ
F	F	F	F	F	F	F
F	F	V	F	F	F	F
F	V	F	V	V	V	V
F	V	V	V	V	V	V
V	F	F	V	F	V	V
V	F	V	V	F	V	V
V	V	F	V	V	V	V
V	V	V	V	V	V	V

Par conséquence, en prenant la disjonction de toutes les modèles, on obtient la FND :

$$\phi \equiv (a \wedge b \wedge c) \vee (a \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (\bar{a} \wedge b \wedge c) \vee (\bar{a} \wedge b \wedge \bar{c})$$

■ **Définition 65 — Clause disjonctive.** Une clause disjonctive est une disjonction de littéraux.

■ **Définition 66 — Forme normale conjonctive (FNC).** Une forme normale conjonctive d'une formule logique est une conjonction de clauses disjonctives.

Théorème 7 — Toute formule logique est équivalente à une forme normale conjonctive.

Démonstration. Soit ϕ une formule logique. On considère sa négation $\neg\phi$. D'après la question précédente, on peut mettre $\neg\phi$ sous une forme normale disjonctive, c'est à dire

$$\neg\phi \equiv c_1 \vee c_2 \ldots \vee c_n \tag{3.55}$$

où les $c_i = l_1 \wedge l_2 \wedge \ldots \wedge l_m$ sont des conjonctions de littéraux. En appliquant la loi de Morgan, on trouve que :

$$\begin{aligned}
\neg\neg\phi &\equiv (\neg c_1) \wedge (\neg c_2) \wedge \ldots \wedge (\neg c_n) && (3.56)\\
&\equiv (l_1 \vee l_2 \ldots \vee l_m) \wedge (\neg c_2) \wedge \ldots \wedge (\neg c_n) && (3.57)\\
&\equiv d_1 \wedge d_2 \wedge \ldots \wedge d_n && (3.58)\\
&\equiv \phi && (3.59)
\end{aligned}$$

où les d_i sont des disjonctions. Donc ϕ peut s'écrire sous une forme normale conjonctive. ■

■ **Exemple 27 — FNC équivalente.** Soit la formule logique $\phi = (a \vee \neg b) \wedge \neg(c \wedge \neg(d \wedge e))$. Une forme normale équivalente est $(a \vee \neg b) \wedge (\neg c \vee d) \wedge (\neg c \vee e)$. Celle-ci est obtenue en utilisant les lois logiques pour changer la forme de la formule.

■ **Exemple 28 — Table de vérité et forme normale conjonctive.** On considère la formule logique $\phi = (a \vee b) \wedge ((c \Longrightarrow b) \vee a)$. Sa table de vérité est toujours :

a	b	c	$a \vee b$	$c \Longrightarrow b$	$(c \Longrightarrow b) \vee a$	ϕ
F	F	F	F	F	F	F
F	F	V	F	F	F	F
F	V	F	V	V	V	V
F	V	V	V	V	V	V
V	F	F	V	F	V	V
V	F	V	V	F	V	V
V	V	F	V	V	V	V
V	V	V	V	V	V	V

Pour trouver la forme normale conjonctive, il faut sélectionner les lignes des contre-modèles de la formule. Puis, pour chaque littéral d'un contre-modèle, prendre la négation de sa valeur et construire une clause disjonctive. Enfin, prendre la conjonction de ces clauses. Par conséquence, on a donc la FNC :

$$\phi \equiv (a \vee b \vee \neg c) \wedge (a \vee b \vee c)$$

En utilisant la distributivité, on obtient :

$$\phi \equiv (a \vee b) \vee (\neg c \vee c) \equiv a \vee b$$

I Problème SAT

■ **Définition 67 — Problème de décision.** Un problème de décision est un problème dont la réponse est binaire : soit le on peut le décider, soit on ne peut pas.

■ **Définition 68 — Problème SAT.** Le problème de satisfaisabilité booléenne (SAT) est un problème de décision lié à une formule de logique propositionnelle et dont l'objectif est de déterminer s'il existe une valuation qui rend la formule vraie.

On note SAT(ψ) = V si ψ est satisfaisable, et F sinon.

■ **Exemple 29 — Exemples de problème SAT.** Pour définir la date d'une réunion, on considère les contraintes suivantes :

- Johann est obligé d'assister à ses cours lundi, mercredi ou jeudi.
- Cécile ne peut pas se libérer mercredi,
- Annaïg est prise le vendredi
- Prosper n'est là ni le mardi ni le jeudi.

Est-il possible de trouver un jour pour fixer la réunion ? Il s'agit d'un problème de satisfaisabilité de la formule logique :

$$(\neg L \vee Ma \vee \neg Me \vee \neg J \vee V) \wedge (L \vee Ma \vee \neg Me \vee J \vee V) \wedge (L \vee Ma \vee Me \vee J \vee \neg V) \wedge (L \vee \neg Ma \vee Me \vee \neg J \vee V)$$

La méthode de résolution d'un problème SAT par la force brute (cf. figure 3.3) est de complexité exponentielle : on explore toutes les valuations possibles. Si la formule possède n variables, alors la complexité est en $\Theta(2^n)$. L'algorithme de Quine (cf. algorithme 1) permet d'éviter de tester des valuations qui ne sont pas solution.

J Algorithme de Quine

a Principe

L'algorithme de Quine (cf algorithme 1) explore l'ensemble des valeurs possibles pour chaque variable propositionnelle. Cette exploration se fait de manière arborescente [4].

La racine de l'arbre d'exploration est la formule. À chaque niveau de l'arbre, l'hypothèse est faite qu'une variable est vraie ou fausse et l'algorithme simplifie la formule en conséquence en remplaçant la variable par sa valuation. Des règles de simplifications permettent de propager

4. L'algorithme de Quine, c'est le retour sur trace ou backtracking appliqué à au problème SAT, cf. le cours de deuxième année, chapitre 8).

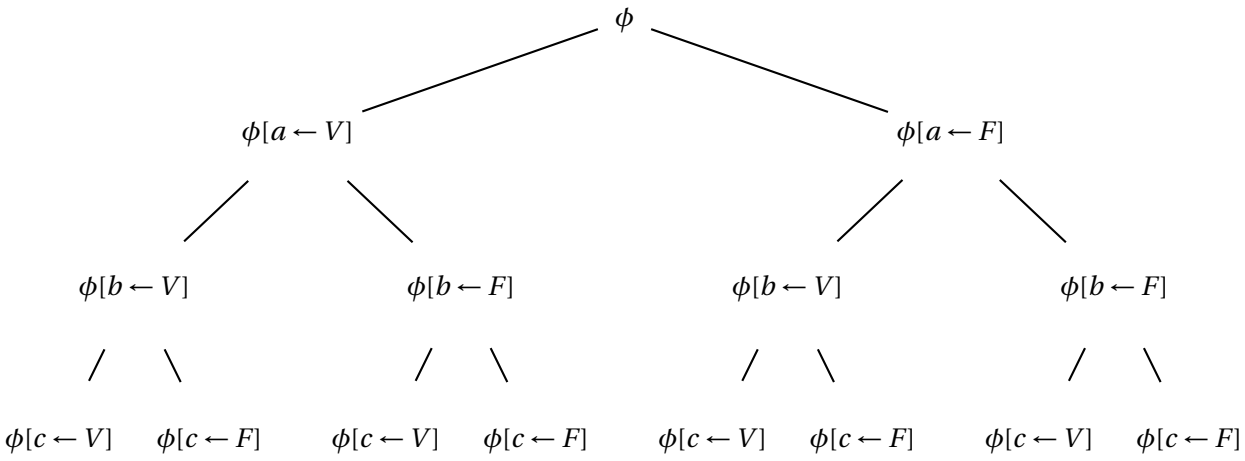


FIGURE 3.3 – Exemple d'arbre d'exploration de toutes les valuations possibles pour une formule logique simple $\phi = (a \wedge b) \vee c$. Selon la valuation des feuilles de l'arbre, on conclue sur la satisfaisabilité de la formule. Si au moins une feuille est évaluée V, alors la formule est satisfaisable.

la valeur et de conclure sur la possibilité de satisfaire la formule sous cette hypothèse ou non. Si c'est le cas, on étiquette la feuille de l'arbre avec un V et l'algorithme renvoie V.

Si la formule simplifiée est non satisfaisable, alors on ne poursuit pas l'exploration des solutions dans cette branche de l'arbre : on la marque F et on effectue un retour sur trace pour continuer l'exploration des autres branches.

Lorsque l'exploration est terminée, si les feuilles sont toutes marquées F, alors la formule n'est pas satisfaisable.

Algorithme 1 Algorithme Quine (SAT)

```
1 : Fonction QUINE_SAT(f)                          ▷ f est une formule logique
2 :     SIMPLIFIER(f)
3 :     si f ≡ ⊤ alors
4 :         renvoyer Vrai
5 :     sinon si f ≡ ⊥ alors
6 :         renvoyer Faux
7 :     sinon
8 :         Choisir une variable x parmi les variables propositionnelles restantes de f
9 :         renvoyer QUINE(f[x ← Vrai]) ou QUINE(f[x ← Faux])
```

■ **Exemple 30 — Quine appliqué.** On considère la formule $\phi = (a \wedge b) \vee c$. La figure 3.4 détaille les étapes de l'algorithme de Quine sur l'arbre d'exploration. La branche de gauche a été raccourcie par les simplifications.

On en déduit la FND équivalente :

$$\phi \equiv c \vee (a \wedge b \wedge \bar{c}) \equiv c \vee (a \wedge b)$$

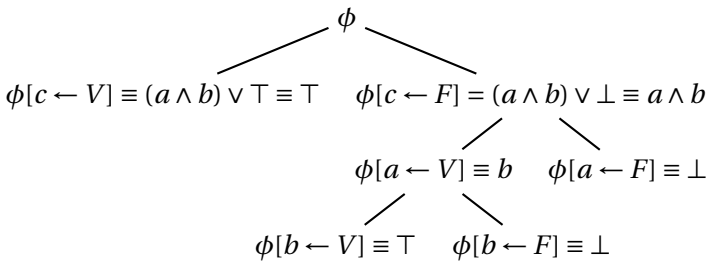


FIGURE 3.4 – Exemple d'arbre d'exploration structurant l'algorithme de Quine.

R Dans le pire des cas, l'algorithme de Quine nécessite d'explorer toutes les branches de l'arbre. Sa complexité est donc exponentielle en fonction du nombre de variables de la formule.

b Règles de simplification

Chaque nœud de l'arbre d'exploration est une formule créée en **remplaçant** une variable par $\top$ ou $\bot$. Il est alors possible de simplifier cette formule,c'est-à-dire de réduire sa taille, c'est-à-dire son nombre d'opérateurs, en observant les équivalences suivantes :

$$\neg\bot \equiv \top \quad \text{smart not} \tag{3.60}$$
$$\neg\top \equiv \bot \quad \text{smart not} \tag{3.61}$$
$$(\top \vee a) \equiv \top \quad \text{smart or} \tag{3.62}$$
$$(\bot \vee a) \equiv a \quad \text{smart or} \tag{3.63}$$
$$(\top \wedge a) \equiv a \quad \text{smart and} \tag{3.64}$$
$$(\bot \wedge a) \equiv \bot \quad \text{smart and} \tag{3.65}$$
$$(\top \Longrightarrow a) \equiv a \quad \text{smart imp} \tag{3.66}$$
$$(a \Longrightarrow \top) \equiv \top \quad \text{smart imp} \tag{3.67}$$
$$(\bot \Longrightarrow a) \equiv \top \quad \text{smart imp} \tag{3.68}$$
$$(a \Longrightarrow \bot) \equiv \neg a \quad \text{smart imp} \tag{3.69}$$
$$(\top \Longleftrightarrow a) \equiv a \quad \text{smart eq} \tag{3.70}$$
$$(\bot \Longleftrightarrow a) \equiv \neg a \quad \text{smart eq} \tag{3.71}$$

On nomme ces opérations de simplification des constructeurs intelligents (*smart constructors*). De plus, si la formule est simplifiée en $\top$ alors l'algorithme de Quine renvoie Vrai. Sinon, il continue l'exploration de l'arbre.

K Exemple de démonstration par induction structurelle

On se propose d'illustrer les démonstration de type induction structurelle sur l'exemple suivant : on se donne les formules logiques $\mathcal{F}$ définies inductivement comme en 48. On définit une transformation τ des formules logiques de la manière suivante :

- tous les opérateurs de type conjonction sont remplacés par des disjonctions et inversement, tous les opérateurs de type disjonctions sont remplacés par des conjonctions,
- les constantes ou les variables propositionnelles sont remplacées par leur négation.

On cherche à démontrer la propriété $\mathcal{P} : \forall \phi \in \mathcal{F}, \tau(\phi) \equiv \neg\phi$.

Démonstration. On procède par induction structurelle sur l'ensemble des formules logiques définis inductivement 48.

(Cas de base : $\bot$) Comme $\bot$ est une constante, d'après la définition de τ on a immédiatement $\tau(\bot) \equiv \top \equiv \neg\bot$.

(Cas de base : $\top$) de la même manière, $\tau(\top) \equiv \bot \equiv \neg\top$.

(Cas de base : $x \in \mathcal{V}$) de la même manière, $\forall x \in \mathcal{V}, \tau(x) \equiv \neg x$.

(Constructeur : not) Soit $\phi = \neg\phi_1$, ϕ_1 étant une formule vérifiant la propriété $\mathcal{P}$. Alors, $\tau(\phi_1) \equiv \neg\phi_1$. Comme la transformation τ ne modifie pas l'opérateur $\neg$, on peut écrire : $\tau(\phi) \equiv \tau(\neg\phi_1) \equiv \neg\tau(\phi_1) \equiv \neg(\neg\phi_1) \equiv \neg\phi$. La propriété $\mathcal{P}$ est donc vérifiée sur une formule construit par not.

(Constructeur : and) Soit ϕ_1 et ϕ_2 deux formules vérifiant la propriété $\mathcal{P}$. On construit alors $\tau(\phi_1 \wedge \phi_2) \equiv \tau(\phi_1) \vee \tau(\phi_2) \equiv \neg\phi_1 \vee \neg\phi_2 \equiv \neg(\phi_1 \wedge \phi_2)$. La propriété $\mathcal{P}$ est donc vérifiée sur une formule construit par and.

(Constructeur : or) Soit ϕ_1 et ϕ_2 deux formules vérifiant la propriété $\mathcal{P}$. On construit alors $\tau(\phi 1 \vee \phi_2) \equiv \tau(\phi_1) \wedge \tau(\phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2 \equiv \neg(\phi_1 \vee \phi_2)$. La propriété $\mathcal{P}$ est donc vérifiée sur une formule construit par or

(Conclusion) La propriété $\mathcal{P}$ est vérifiée pour tous les éléments de bases de formules logiques. Par ailleurs, toute formule logique construite à l'aide d'un constructeur vérifie également la propriété. La propriété $\mathcal{P}$ est donc vérifiée pour toutes les formules logiques.

■

4

DÉDUCTION NATURELLE

À la fin de ce chapitre, je sais :

- ☞ lire un séquent
- ☞ décrire les règles d'introduction et d'élimination
- ☞ justifier les principaux raisonnements de la logique classique
- ☞ construire un arbre de preuve démontrant une formule simple

A Déduction naturelle

Explorer l'arbre syntaxique d'une formule logique s'avère être une tâche dont la complexité est exponentielle dans le pire des cas, $O(2^n)$ si la formule comporte n variables. Vérifier qu'une formule logique est une tautologie est faisable mais pas toujours en temps humain. On cherche donc un moyen de prouver qu'une formule logique est vraie non pas en testant toutes les valuations possibles mais en construisant une preuve, c'est-à-dire une suite d'opérations purement logiques.

La déduction naturelle a été développée dans ce but et est un des premiers système de preuve ayant été mis en œuvre. Il s'agit d'établir des règles d'inférence élémentaires permettant de prouver des formules d'une logique donnée, élémentaires à analyser et à implémenter, dans le but d'obtenir un programme d'assistant de preuve, voire de preuve automatique.

■ **Définition 69 — Séquent ou jugement.** Soit $\mathcal{F}$ l'ensemble des formules logiques, Γ une partie de $\mathcal{F}$ (les hypothèses) et a une formule logique (la conclusion). Un séquent est une relation binaire entre l'ensemble $\mathcal{P}(\mathcal{F})$ et $\mathcal{F}$. On la note ainsi :

$$\Gamma \vdash a \tag{4.1}$$

Elle signifie que l'on peut déduire a en utilisant uniquement les hypothèses Γ : de Γ on peut conclure a.

■ **Exemple 31 — Séquent simple.** Voici un exemple de séquent valide :

$$\forall x \in \mathbb{R}, x^2 - 10x + 21 = 0 \vdash x = 3 \vee x = 7 \tag{4.2}$$

Voici un exemple de séquent non valide, car la conclusion n'est pas vérifiée pour ces hypothèses :

$$\forall x \in \mathbb{R}_+, x^2 - 4x - 21 = 0 \vdash x = -3 \tag{4.3}$$

Par contre, ce dernier est valide :

$$\forall x \in \mathbb{R}_+, x^2 - 4x - 21 = 0 \vdash x = 7 \tag{4.4}$$

■ **Définition 70 — Déduction naturelle.** La déduction naturelle est un système de déduction qui permet de déterminer si des séquents sont **prouvables** ou non. Elle met en valeur le raisonnement «naturel» d'une preuve mathématiques et s'appuie sur une ensemble de **règles** qu'il s'agit de définir afin de pouvoir construire les preuves comme des emboitements de règles d'inférence.

La déduction naturelle organise une démonstration sous la forme d'un **arbre** dont la racine est le séquent à démontrer. Les nœuds de l'arbre se déduisent les uns des autres pas à pas, de manière quasi-évidente via l'introduction ou l'élimination de règles d'inférence élémentaires : la conclusion d'une branche devient une hypothèse du niveau inférieur. Les feuilles sont des axiomes, des introductions de constantes logiques ou des hypothèses, dans tous les cas, des règles sans conditions.

■ **Définition 71 — Règle d'inférence ou règle de déduction.** Une règle d'inférence en déduction naturelle est un ensemble de séquents, les hypothèses (H) ou prémisses, suivi d'un autre séquent conclusion (C). On la représente généralement sous la forme de Gentzen :

$$\frac{\Gamma \vdash H_1 \qquad \Gamma \vdash H_2 \quad \dots \quad \Gamma \vdash H_n}{\Gamma \vdash C}$$

■ **Définition 72 — Axiome.** Un axiome est une règle d'inférence pour laquelle l'ensemble des hypothèses est vide.

$$\frac{}{\Gamma \vdash a}\ \text{ax}$$

■ **Définition 73 — Arbres de preuve ou arbres de dérivation (définition inductive).** L'ensemble des arbres de preuve $\mathcal{A}$ d'un séquent s par déduction naturelle est soit :

(une feuille) l'application d'un axiome dont la conclusion est s,

(un nœud) l'application d'une règle d'inférence (R) dont la conclusion est s et dont les prémisses dérivent d'éléments de $\mathcal{A}$ par des règles d'inférence [a].

a. Ces prémisses sont les conclusions d'arbres de preuve

R Une règle d'inférence forme un constructeur de l'ensemble inductif des arbres de preuve.

La déduction naturelle comporte une dizaine de règles d'inférences qui permettent de construire un arbre de preuve. On distingue les règles qui introduisent une conséquence de plusieurs séquents des règles qui éliminent des séquents en réduisant les conséquences.

B Règles d'introduction et d'élimination

a Introduction et élimination de la conjonction

Lorsqu'on connaît une preuve de la formule a et une preuve de la formule b, alors on peut construire une preuve de la formule $a \wedge b$.

$$\frac{\Gamma \vdash a \qquad \Gamma \vdash b}{\Gamma \vdash a \wedge b} \wedge_i$$

On dit qu'on a **introduit** la conjonction et on note cette règle $\wedge_i$. De même, si on connaît une preuve de $a \wedge b$, alors on peut construire une preuve de a ou de b en **éliminant** la conjonction.

$$\frac{\Gamma \vdash a \wedge b}{\Gamma \vdash a} \wedge_e$$

■ **Exemple 32 — L'opérateur $\wedge$ est commutatif .** On peut montrer que la conjonction est commutative :

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash a \wedge b}\,\text{ax}}{\Gamma \vdash b}\wedge_e \qquad \dfrac{\dfrac{}{\Gamma \vdash a \wedge b}\,\text{ax}}{\Gamma \vdash a}\wedge_e}{\Gamma \vdash b \wedge a}\wedge_i$$

b Introduction et élimination de l'implication

Pour introduire l'implication, on suppose que b peut être déduit de a, alors il est possible de déduire l'implication $a \to b$ en se passant de l'hypothèse a.

$$\frac{\Gamma, a \vdash b}{\Gamma \vdash a \to b} \to_i$$

Pour déduire une formule d'une implication, on suppose qu'on peut justifier a et l'implication. On dispose alors d'une preuve de b :

$$\frac{\Gamma \vdash a \to b \qquad \Gamma \vdash a}{\Gamma \vdash b} \to_e$$

R Dans l'antiquité, cette règle de l'élimination de l'implication $\rightarrow_e$ était nommé *modus ponens*. On la désigne aussi parfois sous le nom de *détachement*. L'implication et le fait de poser a permettent de poser (ou détacher) b.

■ **Exemple 33 — Preuve de $\vdash p \rightarrow p$.** On donne ci-dessous la preuve que l'implication matérielle est réflexive.

$$\dfrac{\dfrac{}{p \vdash p}\,\text{ax}}{\vdash p \rightarrow p}\,\rightarrow_i$$

■ **Exemple 34 — Preuve de $p \wedge q \vdash p \rightarrow q$.** Pour construire cet arbre de preuve, on utilise l'introduction de l'implication et l'élimination de la conjonction.

$$\dfrac{\dfrac{\dfrac{}{p \wedge q, p \vdash p \wedge q}\,\text{ax}}{p \wedge q, p \vdash q}\,\wedge_e}{p \wedge q \vdash p \rightarrow q}\,\rightarrow_i$$

c Introduction et élimination de la disjonction

De la même manière, on introduit et on élimine la disjonction. Lorsqu'on connaît une preuve de la formule a et une preuve de la formule b, alors on peut construire une preuve de la formule $a \vee b$. On peut écrire soit

$$\dfrac{\Gamma \vdash a}{\Gamma \vdash a \vee b}\,\vee_i$$

soit

$$\dfrac{\Gamma \vdash b}{\Gamma \vdash a \vee b}\,\vee_i$$

puisque la disjonction n'exige nullement que les deux soient vraies pour être vraie.

La déduction d'une disjonction est possible s'il existe une formule commune que l'on peut déduire des deux formules de la disjonction.

$$\dfrac{\Gamma \vdash a \vee b \qquad \Gamma, a \vdash c \qquad \Gamma, b \vdash c}{\Gamma \vdash c}\,\vee_e$$

■ **Exemple 35 — Preuve que la disjonction est commutative.** On construit la preuve du séquent $p \vee q \vdash q \vee p$.

$$\dfrac{\dfrac{}{p \vee q \vdash p \vee q}\,\text{ax} \qquad \dfrac{\dfrac{}{p \vee q, p \vdash p}\,\text{ax}}{p \vee q, p \vdash q \vee p}\,\vee_i \qquad \dfrac{\dfrac{}{p \vee q, q \vdash q}\,\text{ax}}{p \vee q, q \vdash q \vee p}\,\vee_i}{p \vee q \vdash q \vee p}\,\vee_e$$

d Introduction et élimination de la négation

Le fait que la négation d'une formule soit vraie lorsque cette formule est fausse nous permet de justifier la négation en montrant que la formule conduit à la contradiction.

$$\dfrac{\Gamma, a \vdash \bot}{\Gamma \vdash \neg a}\,\neg_i$$

Symétriquement, l'élimination d'une négation conduit à une contradiction.

$$\dfrac{\Gamma \vdash \neg a \qquad \Gamma \vdash a}{\Gamma \vdash \bot}\,\neg_e$$

R La contradiction peut donc être engendrée par une proposition et son contraire. C'est le seul moyen d'introduire $\bot$ dans un séquent. On pourrait donc noter l'élimination de la négation $\bot_i$. Il est également possible d'éliminer la contradiction en utilisant le **principe d'explosion**.

$$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash a}\,\bot_e$$

Ce principe peut-être démontré si on accepte le raisonnement par l'absurde [a]. Pour la logique intuitionniste, c'est donc un axiome.

a. Les savants de l'antiquité avaient trouvé ce principe. Mais cela a été prouvé au XII^{e} siècle par Guillaume de Soisson : la démonstration s'appuie sur l'hypothèse qu'on peut prouver une formule et son contraire. Elle utilise un syllogisme disjonctif pour introduire $\bot$ et conclue par un raisonnement par l'absurde.

C Synthèse des règles de la déduction naturelle

Le tableau 4.1 rassemble les règles de construction de la déduction naturelle. Elles permettent de construire des arbres de preuves. Il s'agit de comprendre ces règles avant de les apprendre en les utilisant sur des démonstrations simples.

D Correction de la déduction naturelle

La sémantique des formules logiques et la déduction naturelle constituent deux points de vue sur ce que pourrait être la *vérité* en logique des propositions. La sémantique s'appuie sur des valuations tandis que la déduction cherche à construire le raisonnement qui prouve la formule. En fait, si une proposition est prouvable sous une certaine hypothèse, alors cette proposition est une conséquence sémantique de cette hypothèse et réciproquement.

Formule	Introduction	Élimination
$\top$	$\dfrac{}{\Gamma \vdash \top}\ \top_i$	
$\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash a}\ \bot_e$ (Principe d'explosion)
$a \in \Gamma$ (Axiome)	$\dfrac{}{\Gamma \vdash a}\ \text{ax}$	
Conjonction	$\dfrac{\Gamma \vdash a \qquad \Gamma \vdash b}{\Gamma \vdash a \wedge b}\ \wedge_i$	$\dfrac{\Gamma \vdash a \wedge b}{\Gamma \vdash a}\ \wedge_e$
Disjonction	$\dfrac{\Gamma \vdash a}{\Gamma \vdash a \vee b}\ \vee_i$ et / ou $\dfrac{\Gamma \vdash b}{\Gamma \vdash a \vee b}\ \vee_i$	$\dfrac{\Gamma \vdash a \vee b \qquad \Gamma, a \vdash c \qquad \Gamma, b \vdash c}{\Gamma \vdash c}\ \vee_e$
Implication	$\dfrac{\Gamma, a \vdash b}{\Gamma \vdash a \to b}\ \to_i$	$\dfrac{\Gamma \vdash a \to b \qquad \Gamma \vdash a}{\Gamma \vdash b}\ \to_e$ (Modus ponendo ponens)
Négation	$\dfrac{\Gamma, a \vdash \bot}{\Gamma \vdash \neg a}\ \neg_i$	$\dfrac{\Gamma \vdash \neg a \qquad \Gamma \vdash a}{\Gamma \vdash \bot}\ \neg_e$ (Introduction de la contradiction $\bot_i$)

TABLE 4.1 – Ensemble des règles de la déduction naturelle

Théorème 8 — Équivalence entre prouvabilité et conséquence sémantique. En logique des propositions, toute conséquence sémantique est prouvable et toute formule logique prouvable est une conséquence sémantique.

Formulé autrement, si tout modèle de Γ est un modèle de $a \in \mathcal{F}$ alors on peut déduire a de Γ et réciproquement.

Plus formellement :

$$\Gamma \vDash a \Longleftrightarrow \Gamma \vdash a \tag{4.5}$$

Démonstration. On procède en montrant les implications dans les deux directions.

($\Longleftarrow$) Dans ce sens, la démonstration s'appuie sur la définition des règles d'inférence : pour chaque règle, on montre que si $\Gamma \vdash a$ alors on a $\Gamma \vDash a$.

($\top_i$) Si $\Gamma \vdash \top$, comme $\top$ est une constante et la constante associée au vrai, tout modèle de Γ la satisfait. On a donc $\Gamma \vDash \top$.

($\bot_e$) Supposons que $\Gamma \vdash \bot$. $\bot$ est une constante mais aucun modèle de Γ ne la satisfait, par définition. D'après la définition de la conséquence sémantique, a est une conséquence sémantique de Γ si tout modèle de Γ est un modèle de a. Comme l'ensemble des modèles de $\bot$ est l'ensemble vide, alors il n'existe pas de modèle de Γ qui ne soit pas un modèle de $\bot$. Donc, $\Gamma \vDash \bot$.

($a \in \Gamma$) **Axiome** Soit $a \in \Gamma$. Un modèle de Γ est donc un modèle de a. Donc, $\Gamma \vDash a$.

($\wedge_i$) On s'appuie sur le cas précédent et on suppose donc que $\Gamma \vDash a_1$ et $\Gamma \vDash a_2$. Tout modèle de Γ est donc à la fois un modèle de a_1 **et** un modèle de a_2. Ce qui signifie que $\Gamma \vDash a_1 \wedge a_2$.

($\wedge_e$) On s'appuie sur le cas précédent et on suppose donc que $\Gamma \vDash a_1 \wedge a_2$, Γ est un modèle de a_1 **et** un modèle de a_2. Qui peut le plus peut le moins, Γ est donc un modèle de a_1.

($\vee_i$) On s'appuie sur le cas précédent (axiome) et on suppose donc que $\Gamma \vDash a_1$. Tout modèle de Γ est donc un modèle de a_1. Même s'il n'est pas un modèle de a_2, d'après la définition de la disjonction, ce modèle est un modèle de $a_1 \vee a_2$. Ce qui signifie que $\Gamma \vDash a_1 \vee a_2$.

($\vee_e$) On suppose donc que $\Gamma \vDash a_1 \vee a_2$, Γ, $\Gamma, a_1 \vDash c$ et $\Gamma, a_2 \vDash c$. Soit un modèle de Γ et a_1, alors ce modèle est un modèle de c. On procède de même avec un modèle de Γ et a_2. Donc, un modèle de $\Gamma, a_1 \vee a_2$ est un modèle de c, d'après la définition de la disjonction. Ce qui signifie que $\Gamma, a_1 \vee a_2 \vDash c$.

($\rightarrow_i$) D'après l'hypothèse et les cas précédents, on a $\Gamma, a_1 \vDash a_2$ et on cherche à montrer que $\Gamma \vDash a_1 \rightarrow a_2$. Plusieurs cas sont possibles :

- Si a_1 est vraie, alors pour tout modèle de Γ, a_2 est vraie.
- Si a_1 est fausse, $a_1 \rightarrow a_2$ est toujours vraie, car ex falso quod libet.

Donc l'implication $a_1 \rightarrow a_2$ est vraie (d'après sa table de vérité) pour tout modèle de Γ. Ce qui signifie que $\Gamma \vDash a_1 \rightarrow a_2$.

($\rightarrow_e$) D'après l'hypothèse et les cas précédents, on a $\Gamma \vDash a_1 \rightarrow a_2$ et $\Gamma \vDash a_1$. Prenons un modèle de Γ. D'après notre hypothèse, ce modèle est à la fois un modèle de a_1 et un modèle de l'implication $a_1 \rightarrow a_2$. D'après la table de vérité de l'implication, comme celle-ci est vraie et que a_1 est vraie, on a nécessairement a_2 vraie. Donc, $\Gamma \vDash a_2$.

($\neg_i$) D'après l'hypothèse et les cas précédents, on a $\Gamma, a \vDash \bot$. Soit un modèle de Γ et de a. D'après l'hypothèse, de ce modèle, on ne peut qu'engendrer que le faux. Or, c'est un modèle de a. Donc on ne peut pas en déduire a. Par contre, on peut en déduire $\neg a$. Donc de ce modèle de Γ, on peut déduire $\neg a$.

($\neg_e$) D'après l'hypothèse et les cas précédents, on a $\Gamma \vDash \neg a$ et $\Gamma \vDash a$. Tout modèle de Γ est à la fois un modèle de a et de $\neg a$. Si l'on part du principe du tiers exclu, cela est impossible. L'ensemble des modèles vérifiant ces conditions est vide. On a donc $\Gamma \vDash \bot$.

($\Longrightarrow$) Cette démonstration est l'objet du théorème de complétude de Gödel[5]. On l'admet dans ce cours.

■

R On peut donc conclure que, dans le cadre de la déduction naturelle, :

- tout ce qui est prouvable est vrai (correction),
- tout ce qui est vrai est prouvable (complétude).

R Si des stratégies de preuve existent en déduction naturelle pour chercher un arbre de preuve automatiquement, elles ne sont en général pas très simples à exécuter et font souvent appel au retour sur trace, ce qui nuit à leur efficacité. D'autres systèmes ont été développés depuis (calcul des séquents, calcul des constructions) qui dépassent ces limitations.

E Raisonnements utiles en logique classique

a Raisonnement par l'absurde

Le raisonnement par l'absurde s'énonce simplement :

$$\frac{\Gamma, \neg a \vdash \bot}{\Gamma \vdash a}\ \text{raa}$$

R En latin, le raisonnement se dit *reduction ad absurdum*, d'où l'acronyme raa.

■ **Exemple 36 — Preuve du principe d'explosion.** Il est possible de prouver le principe d'explosion, c'est-à-dire :

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash a}\ \bot_e$$

La preuve s'appuie sur le raisonnement par l'absurde. On introduit habilement la négation de c dans les hypothèses du séquent en procédant par affaiblissement [a] que l'on note aff.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \bot}\ \text{ax}}{\Gamma, \neg a \vdash \bot}\ \text{aff}}{\Gamma \vdash a}\ \text{raa}$$

a. qui peut le plus peut le moins

b Élimination de la double négation

La double négation en logique classique peut être éliminée puisqu'on admet le principe du tiers exclu.

$$\frac{\Gamma \vdash \neg\neg a}{\Gamma \vdash a}\ \neg\neg_e$$

■ **Exemple 37 — Preuve du raisonnement par l'absurde par la double négation.** Il est possible de prouver le principe d'explosion en utilisant l'élimination de la double négation :

$$\dfrac{\dfrac{\dfrac{}{\Gamma, \neg c \vdash \bot}\,\text{ax}}{\Gamma \vdash \neg\neg c}\,\neg_i}{\Gamma \vdash c}\,\neg\neg_e$$

c Tiers exclu

Le principe du tiers exclu s'énonce simplement :

$$\dfrac{}{\Gamma \vdash a \vee \neg a}\,\text{te}$$

■ **Exemple 38 — Preuve du raisonnement par l'absurde par le tiers exclu.** Il est possible de prouver le raisonnement par l'absurde en utilisant le tiers exclu, le principe d'explosion (élimination de $\bot$) et l'élimination de la disjonction :

$$\dfrac{\dfrac{}{\Gamma \vdash a \vee \neg a}\,\text{te} \quad \dfrac{}{\Gamma, a \vdash a}\,\text{ax} \quad \dfrac{\dfrac{}{\Gamma, \neg a \vdash \bot}\,\text{ax}}{\Gamma, \neg a \vdash a}\,\bot_e}{\Gamma \vdash a}\,\vee_e$$

F Exemples de preuves

a Syllogisme hypothétique

Le syllogisme hypothétique s'exprime sous la forme du séquent $p \to q, q \to r \vdash p \to r$.

$$\dfrac{\dfrac{\dfrac{}{p \to q, q \to r, p \vdash q \to r}\,\text{ax} \quad \dfrac{\dfrac{}{p \to q, q \to r, p \vdash p \to q}\,\text{ax} \quad \dfrac{}{p \to q, q \to r, p \vdash p}\,\text{ax}}{p \to q, q \to r, p \vdash q}\,\to_e}{p \to q, q \to r, p \vdash r}\,\to_e}{p \to q, q \to r \vdash p \to r}\,\to_i$$

b Modus tollendo tollens

Du latin *en niant, je nie*, cette figure s'exprime sous la forme du séquent $p \to q, \neg q \vdash \neg p$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p \to q, \neg q, p \vdash p \to q}\,\text{ax} \quad \dfrac{}{p \to q, \neg q, p \vdash p}\,\text{ax}}{p \to q, \neg q, p \vdash q}\,\to_e \quad \dfrac{}{p \to q, \neg q, p \vdash \neg q}\,\text{ax}}{p \to q, \neg q, p \vdash \bot}\,\neg_e}{p \to q, \neg q \vdash \neg p}\,\neg_i$$

G Vers la logique du premier ordre --→ HORS PROGRAMME

a Syllogismes

■ **Définition 74 — Mnémonique.** Au féminin, une mnémonique est un ensemble des procédés qui facilitent les opérations de mémorisation. Dans le cadre de la logique, il s'agit donc d'astuces pour mémoriser des formules logiques. Dans le cadre de l'informatique, on peut utiliser ce mot au masculin; il désigne alors une instruction en langage d'assemblage (de type chaîne de caractères) correspondant à une instruction du langage machine (de type entier codé en binaire), par exemple : `ADD R1, R2`.

Au moyen âge, les philosophes et les logiciens ont développé des mnémoniques pour identifier et mémoriser facilement certaines figures de la logique et de la rhétorique. Ils choisissaient des mots dont les voyelles représentaient des affirmations (A) ou des réfutations (E) **universelles** [1], des affirmations(I) ou des réfutations (O) **particulières** [2]. Les syllogismes du moyen-âge et de l'antiquité exprime donc des prédicats de la logique d'ordre 1.

■ **Exemple 39 — barbara.** Le syllogisme barbara est un syllogisme de type AAA. Il représente une figure du type TOUT M EST P, OR TOUT S EST M, DONC TOUT S EST P.

■ **Exemple 40 — celarent.** Le syllogisme celarent est un syllogisme de type EAE. Il représente une figure du type AUCUN M N'EST P, OR TOUT Q EST M, DONC AUCUN Q N'EST P

La logique du premier ordre introduit la notion de prédicat, de fonction, de variable liée ou libre ainsi que deux quantificateurs. Lorsqu'une formule logique F dépend par une certaine variable propositionnelle x, on note $F(x)$.

■ **Définition 75 — Variable liée.** Dans une formule logique du premier ordre, une variable est liée si le nom par lequel on la désigne ne modifie pas la formule. C'est pourquoi elle est aussi désigné par le terme variable muette.

Par exemple, $\forall x.F(x) \wedge y$ possède la même signification que $\forall t.F(t) \wedge y$. x et t sont des variables liées.

Par contre, $\forall x.F(x) \wedge y$ et $\forall x.F(x) \wedge z$ ne possèdent pas la même signification : l'une est une propriété sur y et l'autre sur z. y et z sont des variables libres.

(R) Une même variable peut apparaître liée et libre dans une même formule comme c'est le cas pour celle-ci : $(\forall x F(x)) \wedge G(x)$

■ **Définition 76 — De la liberté dans les formules.** Une variable est libre dans une formule

1. c'est-à-dire avec le quantificateur universel $\forall$ qui naîtra bien plus tard : *Tous les hommes sont mortels* ou bien *Aucun homme n'est mortel.*
2. c'est-à-dire avec le quantificateur existentiel $\exists$: *Il existe au moins un homme mortel* ou bien *Aucun homme n'est mortel.*

si elle possède au moins une occurrence libre dans cette formule.

Une variable est liée dans une formule si toutes les occurrences de la variable dans la formule sont liées.

b Règles du quantificateur existentiel

Soit une instance d'une formule $F(x)$. Si au moins une des valeurs possibles de x fait que la formule $F(x)$ est vraie, alors on introduit le quantificateur existentiel et on note : $\exists x.F(x)$.

Le quantificateur peut être introduit en déduction naturelle par la règle suivante, t étant une valeur pour laquelle F est satisfaite :

$$\frac{\Gamma \vdash F[x \leftarrow t]}{\Gamma \vdash \exists x.F(x)} \exists_i$$

De même, si la variable x n'est libre dans aucune formule, on peut éliminer le quantificateur existentiel par la règle :

$$\frac{\Gamma \vdash \exists x.F(x) \qquad \Gamma, F \vdash \phi \qquad x \text{ n'est une variable libre ni de } \Gamma \text{ ni de } \phi}{\Gamma \vdash \phi} \exists_e$$

c Règles du quantificateur universel

Le quantificateur universel traduit l'idée que F peut être déduite indépendamment de x. Il est introduit en déduction naturelle par la règle suivante :

$$\frac{\Gamma \vdash F \qquad x \text{ n'est pas une variable libre de } \Gamma}{\Gamma \vdash \forall x.F(x)} \forall_i$$

De même, on peut éliminer le quantificateur universel par la règle en rompant la généralisation :

$$\frac{\Gamma \vdash \forall x.F(x)}{\Gamma \vdash F[x \leftarrow t]} \forall_e$$

■ **Exemple 41 — Preuve simple en logique du premier ordre.** On cherche à montrer que $\forall x F(x) \vdash \exists x F(x)$.

$$\dfrac{\dfrac{\dfrac{}{\forall x.F(x) \vdash \forall x.F(x)}\,\text{ax}}{\forall x.F(x) \vdash F[x \leftarrow t]}\,\forall_e}{\forall x.F(x) \vdash \exists x.F(x)}\,\exists_i$$

H Correspondance Curry-Howard --→ HORS PROGRAMME

■ **Définition 77 — Expression bien typée.** Dans une expression bien typée, les types des fonctions et des opérateurs utilisés coincident avec le type des paramètres des fonctions.

Par exemple, en OCaml, `2 + 3` est bien typée car l'opérateur `+` sait opérer sur deux entiers.

Par contre, `2.0 + 3.0` n'est pas bien typée.

■ **Définition 78 — Jugement de typage.** Si, pour un environnement de variables Γ donné, l'expression e est bien typée et a le type τ, alors on note

$$\Gamma \vdash e : \tau$$

■ **Exemple 42 — Jugements de typage en OCaml.** Voici quelques exemples simples de jugement de typage en OCaml.

- $\vdash$ `21 : int`
- $\vdash$ `true : bool`
- $\vdash$ `(+)2 3 : int`
- $\vdash$ `(+): int -> int -> int`

Si on dispose des jugements :

- $\vdash$ `f : int -> int -> int`
- $\vdash$ `a : int`
- $\vdash$ `b : int`

alors on peut appliquer `a` et `b` à `f` et écrire $\vdash$ `f a b : int`.

De la même manière que pour les formules logiques, un contexte de variables,c'est-à-dire une environnement définissant un typage des variables, doit donc être précisé :

`a : int, b : int, f : int -> int -> int` $\vdash$ `f a b : int`

Ainsi, pourvu que le type défini des expressions utilisées soit respecté, alors on peut déduire le type d'une autre expression.

La définition d'un programme bien typé peut se faire de manière inductive et la dérivation de typage se fait de manière similaire au calcul des séquents. On peut donc vérifier formellement le typage d'un programme.

Les règles de l'axiome, de l'élimination de l'implication ou de l'introduction de l'implication logique possèdent leur correspondant dans la vérification de types.

$$\frac{}{\Gamma \vdash e : \tau}\ \text{ax}$$

$$\frac{\dfrac{}{\Gamma, \texttt{a : s} \vdash \texttt{e : t}}\ \text{ax}}{\Gamma \vdash \texttt{fun a -> e : s -> t}}\ \text{fun}_i$$

$$\frac{\dfrac{}{\Gamma \vdash \texttt{f : s -> t}}\ \text{ax} \qquad \dfrac{}{\Gamma \vdash \texttt{e : s}}\ \text{ax}}{\Gamma \vdash \texttt{f e : t}}\ \text{fun}_e$$

En observant ces dérivations, on est frappé par le fait qu'on pourrait établir une correspondance entre :

- les types et les formules logiques,

- les programmes et les preuves.

C'est la correspondance de Curry-Howard qui est au cœur des logiciels d'assistant de preuve comme Coq qui permettent de vérifier une démonstration et de démonstration automatique. Les applications sont mathématiques, électroniques (conception des circuits) et informatique (vérification d'assertions relatives à des programmes).

R La correspondance Curry-Howard permet de mettre en lumière le lien étroit entre les formules mathématiques et les types, les preuves et les programmes. C'est un argument fort **contre** la brevetabilité du logiciel, car breveter un logiciel, c'est breveter une formule mathématique. Doit-on breveter les formules mathématiques ? Où est-ce un bien commun ?

Troisième partie

Structures de données

5

STRUCTURES ET TYPES ABSTRAITS

À la fin de ce chapitre, je sais :

- ☞ expliquer la notion de type abstrait de données
- ☞ distinguer les différentes structures de données au programme
- ☞ choisir une structure de données adaptée à un algorithme

Écrire un programme optimal en terme de complexité nécessite l'identification des structures de données utilisées très tôt dans le développement : le choix d'une structure de données plutôt qu'une autre, par exemple choisir une entier long plutôt qu'un flottant ou une liste au lieu d'un tableau, peut rendre inefficace un algorithme selon le choix effectué. Le génie logiciel s'appuie donc à la fois sur :

des types simples comme les (`int, float, bool, char`) sont les éléments de base de l'informaticien, éléments qui représentent une information simple, **atomique**.

et types composés comme les listes, les tableaux, les arbres, les files, les piles. Ce sont des structures composites qui permettent et de manipuler l'information sous la forme d'ensembles ordonnés ou non.

Ce chapitre a pour but d'approfondir la définition des structures de données afin de permettre un choix éclairé, c'est-à-dire adapté à un algorithme. C'est pourquoi on définit d'abord ce qu'est un type abstrait de données en illustrant ce concept sur les tableaux, les dictionnaires et les listes. Puis le lien avec les implémentations possibles de ces types en structures de données met en évidence la diversité des solutions disponibles.

A Type abstrait de données et structure de données

■ **Exemple 43 — Analogie introductive : de la fonction mécanique à l'architecture physique.** Pour réaliser une fonction mécanique, il est courant de disposer de plusieurs solutions concrètes. Par exemple, si l'on considère un vélo, comment convertir le mouvement de rotation du pédalier en mouvement de rotation des roues ? La fonction abstraite recherchée, c'est-à-dire le **quoi**, ce que l'on veut pouvoir faire, est un convertisseur de rotation en rotation. Cette **abstraction** peut être réalisée par un système classique (roues dentées et chaîne métallique), par une courroie polyamide et carbone ou par un cardan. Trois réalisations possibles au moins pour une même abstraction.

Tout comme en conception mécanique on distingue l'abstraction à réaliser de sa réalisation, de la même manière, en informatique, on distingue un type de données que l'on désigne par le terme *Type Abstrait de Données* (TAD) de sa réalisation concrète que l'on désigne par le terme *Structure de données*.

■ **Définition 79 — Type abstrait de données (TAD).** Un type de données abstrait est une abstraction d'une structure de données qui ne se préoccupe pas de son implémentation sur une machine : sa structure interne est indiscernable, le type abstrait est vu de l'extérieur comme une boite noire.

Un TAD spécifie le quoi, c'est-à-dire le type de données contenues ainsi que les opérations possibles. Par contre, il ne spécifie pas comment dont les données sont stockées ni comment les opérations sont implémentées.

■ **Définition 80 — Structure de données.** Une structure de données est une mise en œuvre concrète d'un type abstrait, une implémentation d'un type abstrait sur dans un langage de programmation. On y décrit donc le **comment**, c'est-à-dire la manière avec laquelle sont codées les données et les opérations en machine.

R Un type abstrait de données est à une structure de donnée ce qu'un algorithme est à un programme. On spécifie un algorithme ou un type abstrait de données, mais on implémente un programme ou une structure de données.

■ **Exemple 44 — Un entier.** Un entier est un TAD qui :

(données) contient une suite de chiffres [a] éventuellement précédés par un signe − ou +,

(opérations) fournit les opérations $+, -, \times, //, \%$.

Selon le langage, ce TAD entier est implémenté en machine par un type concret différent :

- `int` en Python,
- `Integer` ou `int` en Java,
- `char`, `short`, `int`, `uint`, `long int` en C,

- `int` en OCaml.

a. peu importe la base pour l'instant...

■ **Exemple 45 — Un booléén.** De la même manière, on peut définir un TAD qui désigne un booléen. Un booléen est un TAD qui :

(données) se note Vrai ou Faux,

(opérations) fournit les opérations logiques conjonction, disjonction et négation...

Selon le langage, ce TAD booléen est implémenté en machine par un type concret différent :

- `bool` valant `True` ou `False` en Python,
- `boolean` valant `true` ou `false` en Java,
- `bool` valant `1` ou `0` en C,
- `bool` valant `true` ou `false` en OCaml.

Les exemples précédents de types abstrait de données étaient limités à des types simples. Mais il est possible de définir des types abstraits de données composés.

■ **Exemple 46 — Types abstraits de données composés.** Voici quelques types abstraits composés parmi les plus courants : liste, file, pile, arbre binaire, dictionnaire ou tableau associatif, ensemble, graphe.

■ **Exemple 47 — Pile.** Une pile est un TAD composé de type LIFO (Last In First Out) qui gère une collection d'éléments et dont les trois principales opérations sont :

- empiler (push) : ajouter un élément à la collection,
- dépiler (pop) : retirer le dernier élément ajouté à la collection qui n'a pas été retiré,
- consulter le sommet (peek) : consulter l'élément sur le sommet de la pile.

Une pile peut être implémentée à l'aide d'une liste ou d'un tableau.

■ **Exemple 48 — File.** Une file est un TAD composé de type FIFO (First In First Out) qui gère une collection d'éléments et dont les deux principales opérations sont :

- enfiler (push) : ajouter un élément à la fin de la file,
- défiler (get) : retirer l'élément en tête de la file.

Une file peut être implémentée à l'aide d'une liste ou d'un tableau.

B TAD Tableau

■ **Définition 81 — TAD tableau.** Un TAD tableau représente une structure finie indiçable par des entiers. Cela signifie qu'on peut accéder à la lecture ou à l'écriture de n'importe quel élément directement en utilisant un indice : par exemple `a = t[3]` pour la lecture et `t[7] =`

`67.6` pour l'écriture.

(données) le plus souvent des nombres, en tout cas des types identiques : on appelle cette donnée l'élément d'un tableau.

(opérations) on distingue deux opérations principales caractéristiques :

- l'accès à un élément via un indice entier via un opérateur de type `[]`,
- l'enregistrement de la valeur d'un élément d'après son indice.

Les implémentations du TAD tableau sont la plupart du temps des structures des données linéaires en mémoire : les données d'un tableau sont rangées dans des zones mémoires **continues**, les unes derrières les autres. On peut décliner le TAD tableau de manière :

1. **statique** : la taille du tableau est fixée à la création du tableau. Il n'est pas possible d'ajouter ou d'enlever des éléments.
2. **dynamique** : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

P En Python, il n'existe pas à proprement parlé de type tableau dans le cœur du langage. Cependant, la liste Python est implémentée par un tableau dynamique et permet donc souvent de pallier ce manque. Néanmoins, pour un calcul numérique efficace, il faut absolument privilégier l'usage des tableaux Numpy qui implémentent le TAD tableau statique.

C TAD Liste

■ **Définition 82 — TAD liste.** Un TAD liste représente **une séquence finie d'éléments d'un même type** qui possède un **rang** dans la séquence. Les données sont traitées séquentiellement, dans l'ordre du rang.

Un TAD liste est **dynamique**, c'est-à-dire qu'on peut ajouter ou enlever des éléments.

La longueur d'une liste est le nombre d'éléments qu'elle contient. On dit qu'une liste est vide si elle ne contient aucun élément, sa longueur vaut alors zéro. Le début de la liste est désigné par le terme tête de liste (**head**), le dernier élément de la liste par la fin de la liste (**tail**).

(données) de type simple ou composé

(opérations) on peut trouver [a] :

- un constructeur de liste vide,
- un opérateur de test de liste vide,
- un opérateur pour ajouter en tête de liste,
- un opérateur pour ajouter en fin de liste,
- un opérateur pour déterminer et/ou retirer la tête de la liste,
- un opérateur pour déterminer et/ou retirer la queue de la liste (tout sauf la tête),
- un opérateur pour accéder au ième élément.

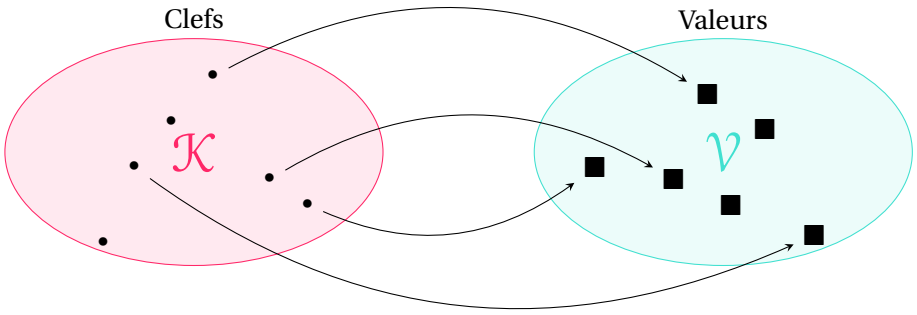


FIGURE 5.1 – Illustration du concept de dictionnaire

- un opérateur pour accéder au dernier élément de la liste.

a. Toutes les implémentations ne proposent pas nécessairement toutes ces opérations !

D TAD Dictonnaire

■ **Définition 83 — TAD Dictionnaire.** Un dictionnaire est une extension du TAD tableau dont les éléments $\mathcal{V}$, au lieu d'être indicés par un entier sont indicés par des clefs appartenant à un ensemble $\mathcal{K}$. Soit $k \in \mathcal{K}$, une clef d'un dictionnaire `D`. Alors `D[k]` est la valeur v de $\mathcal{V}$ qui correspond à la clef k.

On dit qu'un dictionnaire est un **tableau associatif** qui associe une clef `k` à une valeur `v`.
Les opérations sur un dictionnaire sont :

1. rechercher la présence d'une clef dans le dictionnaire,
2. accéder à la valeur correspondant à une clef,
3. insérer une valeur associée à une clef dans le dictionnaire,
4. supprimer une valeur associée à une clef dans le dictionnaire.

R L'intérêt principal d'un dictionnaire est que l'on connaît des implémentation qui permettent de rechercher et d'accéder à un élément en un temps constant $\mathcal{O}(1)$. Rechercher un élément dans une liste est une opération linéaire en $\mathcal{O}(n)$ dans le pire des cas. Dans le cadre d'un tableau, si la recherche par dichotomie est implémentée, alors la recherche d'un élément est logarithmique en $\mathcal{O}(\log(n))$. Si on implémente bien un dictionnaire, tester l'appartenance à un dictionnaire est de complexité constante, ce qui peut accélérer grandement l'exécution d'un algorithme.

P Un dictionnaire relie donc directement une clef, qui n'est pas nécessairement un en-

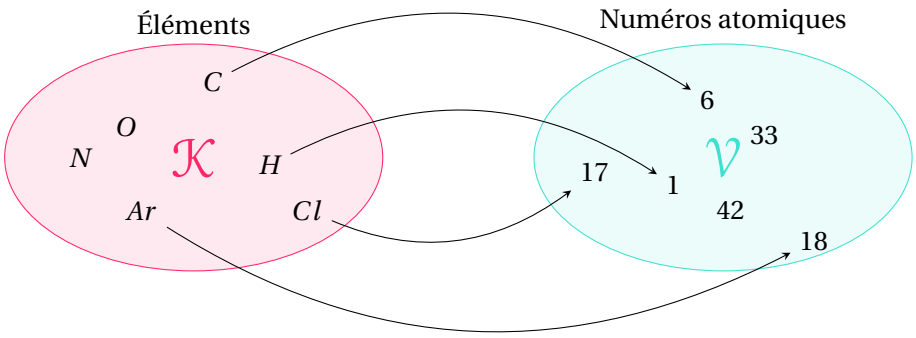


FIGURE 5.2 – Illustration du concept de dictionnaire, ensembles concrets

tier, à une valeur : pas besoin d'index intermédiaire pour rechercher une valeur comme dans une liste ou un tableau. Par contre, cette clef est nécessairement d'un type immuable. Considérons l'exemple donnée sur l'exemple de la figure 5.2. On suppose que les éléments chimiques sont enregistrés via une chaîne de caractères : `"C"`, `"O"`,`"H"`, `"Cl"`, `"Ar"`, `"N"`. Soit `d`, un dictionnaire correspondant à la figure 5.2. Accéder au numéro atomique de l'élément `C` s'écrit : `d["C"]`.

R Un dictionnaire n'est pas une structure ordonnée, à la différence des listes ou des tableaux.

■ **Exemple 49 — Usage des dictionnaires.** Les dictionnaires sont utiles notamment dans le cadre de la programmation dynamique pour la mémoïsation, c'est-à-dire l'enregistrement des valeurs d'une fonction selon ses paramètres d'entrée. Par exemple :

- a-t-on déjà rencontré un sommet lorsqu'on parcourt un graphe ?
- a-t-on déjà calculé la suite de Fibonacci pour $n = 4$?

Répondre à ces questions exige de savoir si pour une clef donnée il existe une valeur.

E Implémentations des tableaux

a Implémentation d'un tableau statique

Dans sa version statique, un TAD tableau de taille fixe n est implémenté par un bloc de mémoire contiguë contenant n cases(cf. figure 5.3). Ces cases sont capables d'accueillir le type d'élément que contient le tableau.

Par exemple, pour un TAD tableau statique de cinq entiers codés sur huit bits, on alloue un espace mémoire de 40 bits subdivisés en cinq octets comme indiqué sur la figure 5.3. Dans la majorité des langages, l'opérateur [] permet alors d'accéder aux éléments [1], par exemple `t[3]`.

1. mais pas en OCaml !

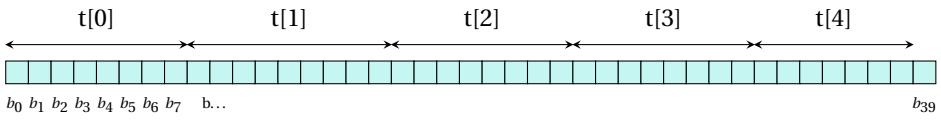


FIGURE 5.3 – Représentation d'un tableau statique en mémoire. Il peut représenter un tableau t de cinq entiers codés sur huit bits. On accède directement à l'élément i en écrivant $t[i]$.

Les éléments sont numérotés à partir de zéro : t[0] est le premier élément.

On peut estimer les coûts associés à l'utilisation d'un tableau statique comme le montre le tableau 5.1.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	créer un nouveau tableau
Ajout d'un élément à la fin	$O(n)$	créer un nouveau tableau
Suppression d'un élément au début	$O(n)$	créer un nouveau tableau
Suppression d'un élément à la fin	$O(n)$	créer un nouveau tableau

TABLE 5.1 – Complexité des opérations associées à l'utilisation d'un tableau statique.

O En OCaml, les types `Array` sont des tableaux muables, c'est-à-dire les éléments sont modifiables.

■ **Exemple 50 — Tableau statique en OCaml.** En OCaml les tableaux statique sont nommés `Array` et l'API est consultable en ligne. Voici un exemple d'utilisation :

```
let t = [|3;9;0;1;7;4;5;2;6|];;
let n = Array.length t;;
let t1 = Array.make 10 0;; (* construire un tableau de 10 éléments initialisés à 0 *)
print_int t2.(0);;      (* accès au premier élément *)
t2.(3) <- 42;;        (* modification d'un élément *)
let m = Array.make_matrix 3 3 0;;
let t2 = Array.init 10 (fun i -> i);;
```

b Implémentation d'un tableau dynamique

Un tableau dynamique est implémenté par un tableau statique de taille n_{max} supérieure à la taille nécessaire pour stocker les données. Les n données contenues dans un tel tableau le sont donc simplement entre les indices 0 et $n-1$. Si la taille n_{max} n'est plus suffisante pour

stocker toutes les données, on crée un nouveau tableau statique plus grand de taille kn_{max} et on recopie les données dedans.

Toute la subtilité des tableaux dynamiques réside dans la manière de gérer les nouvelles allocations mémoires lorsque le tableau doit être modifié.

R Comme le montre le tableau 5.2, l'intérêt majeur du tableau dynamique est de proposer un accès direct constant comme dans un tableau statique tout en évitant les surcouts liés à l'ajout d'éléments.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	décaler tous les éléments contigus
Ajout d'un élément à la fin	$O(1)$	amorti : il y a de la place ou pas
Suppression d'un élément au début	$O(n)$	décaler tous les éléments contigus
Suppression d'un élément à la fin	$O(1)$	amorti : il y a de la place, parfois trop

TABLE 5.2 – Complexité des opérations associées à l'utilisation d'un tableau dynamique.

R Certaines opérations sont à coût constant ou linéaire : lorsqu'il n'y a plus de place dans le tableau, il faut bien créer la nouvelle structure adaptée au nombre d'éléments et cela a un coût linéaire $O(n)$. Donc le coût **amorti** en $O(1)$ signifie qu'il constant la plupart du temps mais que parfois cela peut être linéaire.

■ **Exemple 51 — Complexité amortie de l'ajout en fun dans un tableau dynamique.** Pour illustrer la notion de complexité amortie, on choisit un tableau dynamique dont la taille est **doublée** à chaque fois qu'on redimensionnne le tableau. Imaginons qu'on a inséré $n = 2^m$ éléments. À la fin des opérations, on a effectué $C(n)$ opérations, n insertions dont le coût est en :

- $O(1)$ si la taille est suffisante
- $O(i)$ si $i-1$, la taille du tableau avant insertion, est une puissance de 2 : dans ce cas, on crée un nouveau tableau et on recopie les $i-1$ premiers éléments plus le ième. D'où un coût linéaire par rapport à la taille du tableau.

$$C(n) = n \times 1 + \sum_{k=0}^{m} 2^k = n + 2\frac{1-2^{m+1}}{1-2} = n + 2(2^{m+1}-1) = O(n+4n) = O(n) \tag{5.1}$$

Cela montre que lorsqu'on insère n éléments dans un tableau dynamique, le coût est proportionnel à n. Donc l'insertion d'un seul élément est en $O(1)$, en complexité amortie.

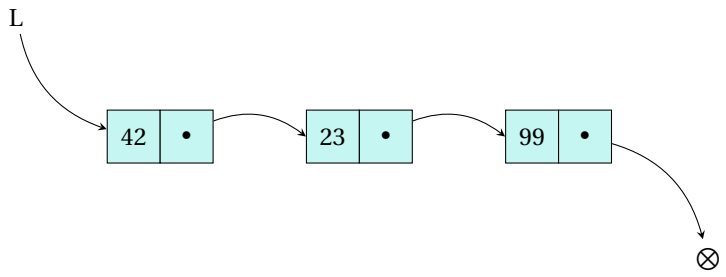


FIGURE 5.4 – Représentation d'une liste simplement chaînée d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

P En python le type `list` est implémenté par un tableau dynamique mais se comporte bien comme un TAD liste !

Cela a pour conséquence que :

- `L.pop()` et `L.append()` sont de complexité $O(1)$, donc supprimer ou ajouter en fin ne coûte pas cher,
- alors que `L.pop(0)` et `L.insert(0,elem)` sont de complexité $O(n)$ et donc supprimer ou ajouter en tête coûte cher.

Lorsqu'un algorithme doit supprimer ou ajouter en tête, il vaut mieux utiliser une autre structure de données qu'une `list` Python. Dans la bibliothèque `collections`, le type `deque` représente une liste sur laquelle les opérations d'ajout et de suppression en tête ou en fin sont en $O(1)$.

R Rechercher un élément dans un tableau statique ou dans un tableau dynamique présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

F Implémentations des listes

a Listes simplement chaînées

Un élément d'une liste simplement chaînée est une cellule constituée de deux parties :

- la première contient une donnée, par exemple un entier pour une liste d'entiers,
- la seconde contient un pointeur, c'est-à-dire une adresse mémoire, vers un autre élément (l'élément suivant) ou rien.

Une liste simplement chaînée se présente donc comme une succession d'éléments composites, chacun pointant sur le suivant et le dernier sur rien. En général, la variable associée à une liste simplement chaînée n'est qu'un pointeur vers le premier élément.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 5.3 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

R Rechercher un élément dans une liste chaînée présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

■ **Exemple 52 — Les listes en OCaml.** Les listes OCaml sont des listes chaînées dont le type est :

```
type 'a list =
        | []
        | (::) of 'a * 'a list
```

Un type `list` est donc soit une liste vide `[]` soit un couple composé d'un élément de type `'a` et d'une liste de `'a`. Le constructeur `::` est noté entre parenthèse car il possède une syntaxe infixe. Ce constructeur permet d'ajouter en tête de liste un élément. On l'utilise ainsi :

```
let l = [1;3;4];;
let l2 = 0::l;; (* l2 vaut [0;1;3;4] *)
```

L'API List OCaml est consultable en ligne. Celle-ci est riche et on y trouve notamment les fonctions ci-dessous :

```
let n = List.length l;;
let head = List.hd l;;
let tail = List.tl l;;
let fourth = List.nth l 3;;
let b = List.mem 3 l;;
```

O En OCaml, les types `List` sont des listes immuables, c'est-à-dire les éléments ne sont pas modifiables, on ne peut pas ajouter, retirer ou modifier un élément d'une liste. Pour réaliser ces opérations, il est nécessaire de construire une autre liste avec un élément en plus, en moins ou un élément différent.

Le filtrage de motif utilise la déconstruction de liste pour parcourir une liste de la tête de liste à la fin comme suit :

```
let rec rm e l = (* supprimer les éléments qui valent e dans l *)
    match l with
        | [] -> []
        | h::t when h = e -> rm e t
        | h::t -> h::(rm e t);;
```

Sur cet exemple, on se rend compte que supprimer un élément d'une liste, c'est en construire une autre identique sans l'élément à supprimer. Pour construire cette autre liste, on en construit en fait plusieurs intermédiaires : le ramasse miettes (Garbage Collecter) d'OCaml efface de la mémoire automatiquement les listes créées qui ne sont plus nécessaires. Sans ce mécanisme, ce processus serait terriblement inefficace d'un point de vue mémoire.

b Listes doublement chaînées

Un élément d'une liste doublement chaînée (cf. figure 5.5) est une cellule constituée de trois parties :

- la première contient un pointeur vers l'élément précédent,
- la deuxième contient une donnée,
- la troisième contient un pointeur vers l'élément suivant.

Une liste doublement chaînée enregistre dans sa structure un pointeur vers le premier élément et un pointeur vers le dernier élément. Ainsi on peut toujours accéder directement à la tête et à la fin de liste. Par contre, c'est un peu plus lourd en mémoire et plus difficile à implémenter qu'une liste simplement chaînée. Le tableau 5.4 recense les coûts associés aux opérations sur les listes doublement chaînées.

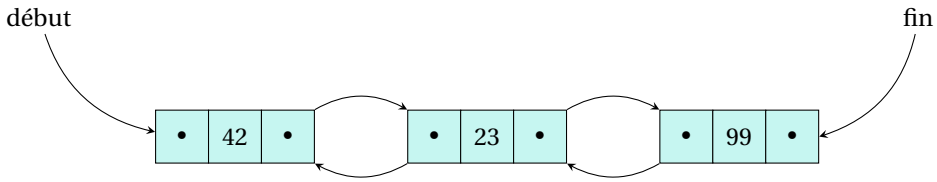


FIGURE 5.5 – Représentation d'une liste doublement chaînée d'entiers L. On conserve un pointeur sur le premier élément et un autre sur le dernier élément de la liste.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	pointeur sur le premier élément
Accès à un élément à la fin	$O(1)$	pointeur sur le dernier élément
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	pointeur sur le premier élément
Ajout d'un élément à la fin	$O(1)$	pointeur sur le dernier élément
Suppression d'un élément au début	$O(1)$	pointeur sur le premier élément
Suppression d'un élément à la fin	$O(1)$	pointeur sur le dernier élément

TABLE 5.4 – Complexité des opérations associées à l'utilisation d'une liste doublement chaînée.

G Bilan des opérations sur les structures listes et tableaux

Opération	Tableau statique	Liste chaînée	Liste doublement chaînée	Tableau dynamique
Accès à un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Accès à un élément à la fin	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Accès à un élément au milieu	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ajout d'un élément au début	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Ajout d'un élément à la fin	$O(n)$	$O(n)$	$O(1)$	$O(1)$ amorti
Suppression d'un élément au début	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Suppression d'un élément à la fin	$O(n)$	$O(1)$	$O(1)$	$O(1)$ amorti
Recherche d'un élément	$O(n)$	$O(n)$	$O(n)$	$O(n)$

TABLE 5.5 – Complexité des opérations associées à l'utilisation des listes et des tableaux.

H Implémentation d'un TAD dictionnaire

On peut implémenter efficacement un TAD dictionnaire à l'aide

1. des tables de hachage,
2. d'arbres binaires de recherche équilibrés (AVL ou arbres rouges et noirs).

Il existe de nombreuses implémentations possibles du TAD dictionnaire. On peut, par exemple, les implémenter avec des listes, mais l'efficacité n'est pas au rendez-vous. .

Opération	**Table de hachage**	**Arbre de recherche équilibré**
Ajout (pire des cas)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Accès (pire des cas)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Ajout (en moyenne)	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Accès (en moyenne)	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

TABLE 5.6 – Complexité des opérations associées à l'utilisation des tables de hachage ou des arbres pour implémenter un TAD dictionnaire. Les coûts indiqués sont dans le pire des cas ou en moyenne.

6

ARBRES BINAIRES ET ABR

À la fin de ce chapitre, je sais :

- ☞ définir un arbre binaire de manière inductive
- ☞ calculer la hauteur et la taille d'un arbre binaire en utilisant la définition inductive
- ☞ parcourir en profondeur un arbre binaire dans un ordre préfixe, infixe ou postfixe
- ☞ calculer la complexité d'un parcours en profondeur
- ☞ parcourir un arbre de recherche pour trouver un élément
- ☞ insérer et supprimer un élément dans un arbre de recherche
- ☞ donner les complexités associées aux opérations sur un arbre binaire de recherche

■ **Définition 84 — Type immuable.** Type de données que l'on ne peut pas modifier en mémoire.

Les listes chaînées et les arbre constituent les deux types **immuables** au programme de l'option informatique. À la différence d'une liste chaînée qui est une structure ordonnée séquentielle, un arbre est une **structure de données hiérarchique**. Elle permet d'implémenter différents types abstraits : des dictionnaires, des tas binaires et des files.

A Des arbres

■ **Définition 85 — Arbre.** Un arbre est un graphe connexe, acyclique et enraciné.

Ⓡ La racine d'un arbre $\mathcal{A}$ est un sommet r particulier que l'on distingue : le couple (A, r) est un nommé arbre enraciné. On le représente un tel arbre verticalement avec la racine

placée tout en haut comme sur la figure 6.1. Dans le cas d'un graphe orienté, la représentation verticale permet d'omettre les flèches.

R On confondra par la suite les arbres enracinés et les arbres.

■ **Définition 86 — Nœuds.** Les nœuds d'un arbre sont les sommets du graphe associé. Un nœud qui n'a pas de fils est une **feuille** (ou nœud externe). S'il possède des descendants, on parle alors de **nœud interne**.

■ **Définition 87 — Descendants, père et fils.** Si une arête mène du nœud i au nœud j, on dit que i est le **père** de j et que j est le **fils** de i. On représente l'arbre de telle sorte que le père soit toujours au-dessus de ses fils.

■ **Définition 88 — Arité d'un nœud.** L'arité d'un nœud est le nombre de ses fils.

■ **Définition 89 — Feuille.** Un nœud d'arité nulle est appelé une feuille.

■ **Définition 90 — Profondeur d'un nœud.** La profondeur d'un nœud est le nombre d'arêtes qui le sépare de la racine.

■ **Définition 91 — Hauteur d'un arbre.** La hauteur d'un arbre est la plus grande profondeur d'une feuille de l'arbre.

■ **Définition 92 — Taille d'un arbre.** La taille d'un arbre est le nombre de ses nœuds.

R Attention, la taille d'un graphe est le nombre de ses arêtes... Un arbre possède toujours $n-1$ arêtes si sa taille est n.

■ **Définition 93 — Sous-arbre.** Chaque nœud d'un arbre $\mathcal{A}$ est la racine d'un arbre constitué de lui-même et de ses descendants : cette structure est appelée sous-arbre de l'arbre $\mathcal{A}$.

R La notion de sous-arbre montre qu'un arbre est une structure intrinsèquement récursive ce qui sera largement utilisé par la suite !

■ **Définition 94 — Arbre recouvrant.** Un arbre recouvrant d'un graphe G est un sous-graphe couvrant de G qui est un arbre.

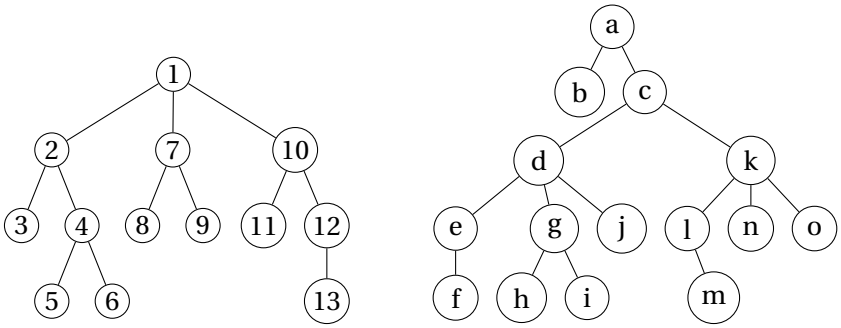


FIGURE 6.1 – Exemples d'arbres enracinés.

B Arbres binaires

On peut facilement transformer un arbre n-aire en un arbre binaire, beaucoup plus facile à coder. C'est pourquoi on s'intéresse tout particulièrement aux arbres binaires.

■ **Définition 95 — Arbre binaire.** Un arbre binaire est un arbre tel que tous les nœuds ont une arité inférieure ou égale à deux : chaque nœud possède au plus deux fils.

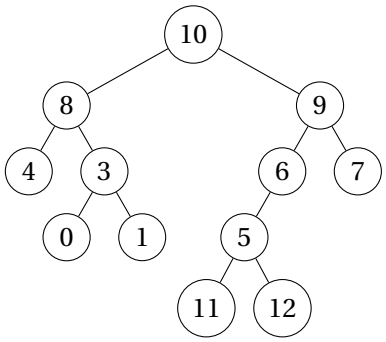


FIGURE 6.2 – Arbre binaire

■ **Définition 96 — Arbre binaire strict.** Un arbre binaire strict est un arbre dont tous les nœuds possèdent zéro ou deux fils.

■ **Définition 97 — Arbre binaire parfait.** Un arbre binaire parfait est un arbre dans lequel tous les niveaux sauf le dernier doivent être totalement remplis. Si le dernier n'est pas rempli totalement alors il doit être rempli de gauche à droite.

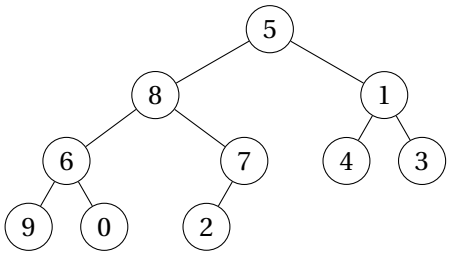


FIGURE 6.3 – Arbre binaire parfait

■ **Définition 98 — Arbre binaire équilibré.** Un arbre binaire est équilibré si sa hauteur est minimale, c'est à dire $h(a) = O(\log|a|)$.

R Un arbre parfait est un arbre équilibré.

Théorème 9 — La hauteur d'un arbre parfait de taille n vaut $\lfloor \log n \rfloor$. Soit a un arbre binaire parfait de taille n. Alors on a :

$$h(a) = \lfloor \log n \rfloor \tag{6.1}$$

Démonstration. Soit a un arbre binaire parfait de taille n. Comme a est parfait, on sait que tous les niveaux sauf le dernier sont remplis. Ainsi, il existe deux niveaux de profondeur $h(a)-1$ et $h(a)$. On peut encadrer le nombre de nœuds de a en remarquant que chaque niveau k possède 2^k nœuds, sauf le dernier. On a donc :

$$1+2+\ldots+2^{h(a)-1} < |a| \leqslant 1+2+\ldots+2^{h(a)} \tag{6.2}$$

$$\sum_{k=0}^{h(a)-1} 2^k < |a| \leqslant \sum_{k=0}^{h(a)} 2^k \tag{6.3}$$

$$2^{h(a)} - 1 < |a| \leqslant 2^{h(a)+1} - 1 \tag{6.4}$$

$$2^{h(a)} \leqslant |a| < 2^{h(a)+1} \tag{6.5}$$

On en conclut que $\lfloor \log_2 |a| \rfloor - 1 < h(a) \leqslant \lfloor \log_2 |a| \rfloor$ et donc que $h(a) = \lfloor \log_2(n) \rfloor$. ■

C Définition inductive des arbres binaires

La plupart des caractéristiques et des résultats importants liés aux arbres binaires peuvent se démontrer par induction structurelle. Cette méthode est une généralisation des démonstrations par récurrences sur $\mathbb{N}$ pour un ensemble défini par induction (cf. figure 6.4).

■ **Définition 99 — Étiquette d'un nœud.** Une étiquette d'un nœud est une information portée au niveau d'un nœud d'un arbre.

■ **Définition 100 — Définition inductive d'un arbre binaire.** Soit E un ensemble d'étiquettes. L'ensemble $\mathcal{A}_E$ des arbres binaires étiquetés par E est défini inductivement par :

1. VIDE est un arbre binaire appelé arbre vide (parfois noté $\emptyset$),
2. Si $e \in E$, $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ sont deux arbres binaires, alors $(f_g, e, f_d) \in \mathcal{A}_E$, c'est à dire que le triplet (f_g, e, f_d) est un arbre binaire étiqueté par E.

f_g et f_d sont respectivement appelés fils gauche et fils droit.

L'implémentation en OCaml donne :

```
type 'a btree = Vide | Noeud of 'a * 'a btree * 'a btree
```

C'est un type **polymorphe**, c'est-à-dire que les étiquettes sont de type générique `'a`. On peut donc créer des arbres dont les étiquettes sont des `int`, des `char` ou des `float` uniquement à partir de cette définition. Par exemple :

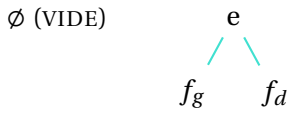


FIGURE 6.4 – Arbre binaire et définition inductive : arbre vide à gauche, arbre induit par e, f_g et f_d à droite

```
let  arbre = Noeud(3, Vide, Vide)
(* val arbre : int bintree = Noeud (3, Vide, Vide) *)
let  arbre = Noeud(3, Noeud(5, Noeud(7,Vide,Vide), Noeud(9, Vide,Vide)), Noeud(2, Noeud
    (8,Vide,Vide), Noeud(6, Vide,Vide)))
(* val arbre : int bintree =
Noeud (3, Noeud (5, Noeud (7, Vide, Vide), Noeud (9, Vide, Vide)),
Noeud (2, Noeud (8, Vide, Vide), Noeud (6, Vide, Vide))) *)
```

Ces arbres correspondent aux figure ci-dessous :

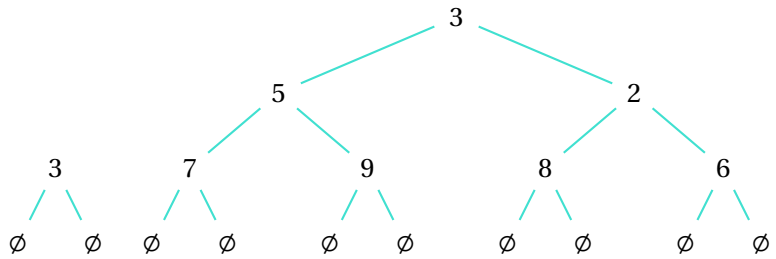


FIGURE 6.5 – Arbres binaires correspondant aux codes OCaml

■ **Définition 101 — Définition inductive des arbres binaires stricts non vides.** Soit E un ensemble d'étiquettes. L'ensemble $\mathcal{A}_E$ des arbres binaires étiquetés par E est défini inductivement par :

1. LEAF(e) est un arbre binaire appelé feuille qui porte une étiquette e,
2. Si $e \in E$, $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ sont deux arbres binaires, alors $(f_g, x, f_d) \in \mathcal{A}_E$, c'est à dire que le triplet (f_g, x, f_d) est un arbre binaire étiqueté par E.

L'étiquette de la feuille peut-être d'une nature différente de celle des nœuds internes.

En OCaml on peut définir ainsi un arbre binaire strict :

```
type ('a, 'b) sbtree = Feuille of 'a | Noeud of 'b * ('a, 'b) sbtree * ('a, 'b) sbtree
```

On peut ainsi définir les mêmes arbres que sur la figure 6.5 :

```
let  arbre = Feuille 3
(* val arbre : (int, 'a) sbtree = Feuille 3 *)
let  arbre = Noeud(3, Noeud(5, Feuille 7, Feuille 9), Noeud(2, Feuille 8, Feuille 6))
(* val arbre : (int, int) sbtree =
Noeud (3, Noeud (5, Feuille 7, Feuille 9), Noeud (2, Feuille 8, Feuille 6)) *)
```

On peut également définir des arbres plus complexes :

```
let  arbre = Noeud(3, Noeud(5, Feuille 'a', Feuille 'b'), Noeud(2, Feuille 'c', Feuille '
    d'))
(* val arbre : (char, int) sbtree =
Noeud (3, Noeud (5, Feuille 'a', Feuille 'b'), Noeud (2, Feuille 'c', Feuille 'd')) *)
```

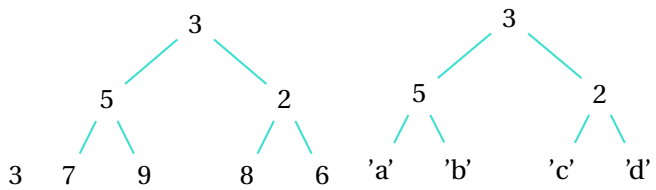


FIGURE 6.6 – Arbres binaires stricts non vides

R On utiliser dans ce qui suit des arbres binaires tels qu'ils sont définis en 100, c'est-à-dire des arbres binaires qui peuvent être vide. On peut naturellement extrapoler ces définitions dans le cas des arbres binaires stricts non vide.

D Démonstration par induction structurelle

■ **Définition 102 — Démonstration par induction structurelle sur un arbre binaire.** Soit $\mathcal{P}(a)$ un prédicat exprimant une propriété sur un arbre a de $\mathcal{A}_E$, l'ensemble des arbres binaires étiquetés sur un ensemble E. On souhaite démonter cette propriété.

La démonstration par induction structurelle procède comme suit :

1. (**CAS DE BASE**) Montrer que $\mathcal{P}(\text{VIDE})$ est vraie, c'est-à-dire que la propriété est vraie pour l'arbre vide,
2. (**PAS D'INDUCTION (Constructeur `Noeud`)**) Soit $e \in E$ une étiquette et $f_g \in \mathcal{A}$ et $f_d \in \mathcal{A}$ deux arbres binaires pour lesquels $\mathcal{P}(f_g)$ et $\mathcal{P}(f_d)$ sont vraies. Montrer que $\mathcal{P}\big((f_g, e, f_d)\big)$ est vraie.
3. (**CONCLUSION**) Conclure que quelque soit $a \in \mathcal{A}$, comme la propriété est vérifiée pour le cas de base et qu'elle est vérifiée pour le constructeur, $\mathcal{P}(a)$ est vraie.

E Définitions inductives de fonction sur les arbres

■ **Définition 103 — Définition inductive d'une fonction à valeur dans $\mathcal{A}_E$.** On définit une fonction ϕ de $\mathcal{A}_E$ à valeur dans un ensemble $\mathcal{Y}$ par :

1. la donnée de la valeur de $\phi(\text{VIDE})$,
2. en supposant connaître $e \in E$, $\phi(f_g)$ et $\phi(f_d)$ pour f_g et f_d dans $\mathcal{A}_E$, la définition de $\phi((f_g, e, f_d))$.

■ **Exemple 53 — Définition inductive de la hauteur d'un arbre.** Soit $a \in \mathcal{A}$ un arbre binaire. La hauteur $h(a)$ de a est donnée par :

1. $h(\text{VIDE}) = -1$,
2. $h\big((f_g, e, f_d)\big) = 1 + \max\big(h(f_g), h(f_d)\big)$.

R La figure 6.7 justifie le fait qu'un arbre vide est une hauteur égale à -1 : dans le cas d'une feuilles, on a alors $h = 1 - 1 = 0$. La figure 6.8 justifie le fait qu'une feuille étiquetée dans une arbre binaire strict non vide possède une hauteur de 0.

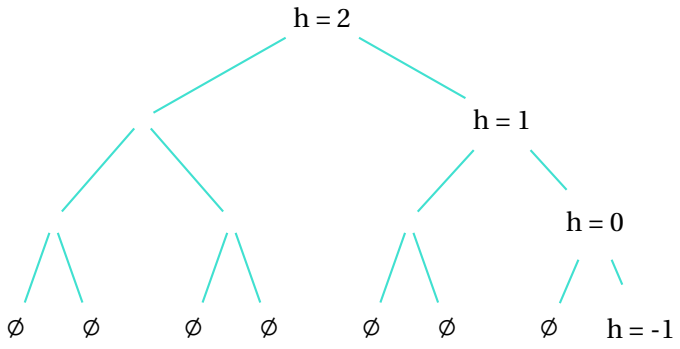


FIGURE 6.7 – La hauteur d'un arbre vide vaut -1 dans le cas d'un arbre binaire défini par 100

■ **Exemple 54 — Définition inductive de la taille d'un arbre.** Soit $a \in \mathcal{A}$ un arbre binaire. La taille $|a|$ de a est donnée par :

1. $|\text{VIDE}| = 0$,
2. $|(f_g, e, f_d)| = 1 + |f_g| + |f_d|$.

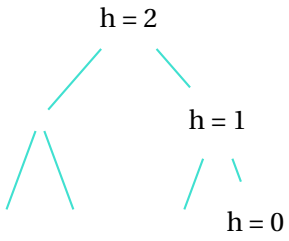


FIGURE 6.8 – La hauteur d'une feuille étiquetée vaut 0 dans le cas d'un arbre binaire strict non vide définie par 101

F Parcours en profondeur d'un arbre binaire

■ **Définition 104 — Parcours d'un arbre.** Le parcours d'un arbre est l'action de visiter une seule fois chaque nœud. L'intérêt d'un parcours est que l'on peut alors effectuer un calcul sur tous les nœuds de l'arbre : recherche d'une étiquette, compilation d'information ou modification de l'arbre.

■ **Définition 105 — Parcours en profondeur.** Un parcours en profondeur traite en priorité les enfants d'un nœud avant de traiter ses frères.

Les parcours en profondeur se programment naturellement récursivement et sont illustrés sur la figure 6.9. On distingue les parcours :

préfixe pour lequel l'étiquette du nœud en cours est traitée **avant** celles des deux sous-arbres gauche et droit,

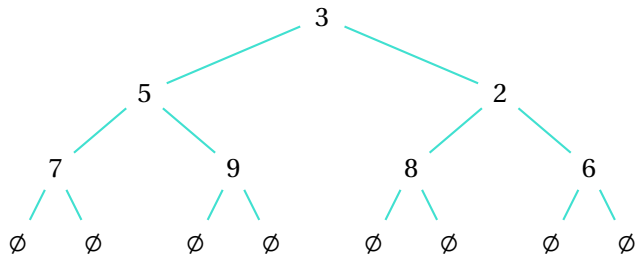
infixe pour lequel l'étiquette du nœud en cours est traitée **entre** celles des sous-arbres gauche et droit,

postfixe pour lequel l'étiquette du nœud en cours est traitée **après** celles des deux sous-arbres gauche et droit.

En OCaml, on peut par exemple coder le parcours préfixe ainsi :

```
let rec prefixe a =
    match a with
    | Vide -> []
    | Noeud (e, fg, fd) -> e :: prefixe fg @ prefixe fd
```

R La complexité d'un parcours préfixe dépend de la structure de l'arbre. Si celui-ci est équilibré, moins d'appels récursifs seront nécessaires que s'il est en forme de peigne. On notera cependant que, comme l'opérateur concaténation `@` présente une complexité linéaire, proportionnelle à la longueur de la première opérande, la performance du code ci-dessus n'est pas



Parcours préfixe : 3 5 7 9 2 8 6
Parcours infixe : 7 5 9 3 8 2 6
Parcours postfixe : 7 9 5 8 6 2 3

FIGURE 6.9 – Parcours en profondeur d'un arbre binaire : on choisit la convention de traiter le fils gauche avant le fils droit.

optimale.

Soit n le nombre de nœuds de l'arbre. Le calcul de la complexité de `(e : : prefixe fg @ prefixe fd)` conduit à :

$$C(n) = 1 + C_g + n_g + C_d$$

car le coût de la concaténation est proportionnel à la longueur de la première liste. Dans le cas d'un peigne à gauche, on aurait trouvé :

$$C(n) = 1 + n - 1 + C(n-1) + C_0 = 1 + n - 1 + C(n-1) + 1 = n + 1 + C(n-1) = \frac{(n+1)(n+2)}{2} = O(n^2)$$

ce qui légitime la seconde approche avec accumulateur !

```
let rec prefixe a acc =
    match a with
    | Vide -> acc
    | Noeud (e, fg, fd) -> e :: prefixe fg (prefixe fd acc)
```

R D'après l'expression `e : : prefixe fg (prefixe fd acc)`, on peut écrire la complexité de la manière suivante : $C(n) = 1 + C_g + C_d$.

Supposons que l'arbre est en forme de peigne à gauche : c'est le cas pour lequel il y aura un maximum d'appels récursifs. Alors on peut simplifier la relation précédente en :

$$C(n) = 1 + C_{n-1}$$

Il s'agit d'une suite arithmétique de raison 1 et de premier terme $C(0) = 1$. On en déduit que $C(n) = n + 1 = O(n)$. La complexité du parcours prédfixe dans le pire des cas est linéaire.

G Proprétés et manipulation des arbres binaires

■ **Exemple 55 — Relation entre la taille et les nœuds.** Soit un arbre binaire à n nœuds et de hauteur h. On se propose de démontrer les propriétés suivantes :

1. $h+1 \leqslant n \leqslant 2^{h+1}-1$
2. cet arbre possède $n+1$ sous-arbres vides.

■ **Exemple 56 — Dénombrer les arbres binaires.** On considère des arbre binaires et on cherche à trouver toutes les structures possibles avec n nœuds.

1. Dénombrer les arbres binaires qui possèdent 0, 1, 2, 3 et 4 nœuds.
2. On peut montrer [a] la suite ainsi formée constitue les nombres de Catalan :

$$C(n) = \frac{1}{1+n}\binom{2n}{n}$$

a. à faire en cours de math ;-)

■ **Exemple 57 — Nombre de feuilles.** On considère un arbre binaire à n nœuds. Soit f le nombre de feuilles de l'arbre. Montrer que $f \leqslant \frac{n+1}{2}$.

H Arbre binaire de recherche (ABR)

■ **Définition 106 — Arbre binaire de recherche.** Soit un ensemble d'étiquettes $\mathcal{E}$ muni d'un ordre total. Un arbre binaire de recherche est un arbre binaire étiqueté par $\mathcal{E}$ dont les nœuds $N(e,g,d)$ vérifient la propriété suivante :

- l'élément e est plus grand que toutes les étiquettes du sous-arbre gauche g,
- l'élément e est plus petit que toutes les étiquettes du sous-arbre droit d.

■ **Exemple 58 — Arbres binaires de recherche.** La figure 6.10 représentent deux arbres binaires de recherche : l'un est étiqueté avec des entiers et est équilibré, l'autre est étiqueté avec des chaînes de caractères et n'est pas équilibré.

L'implémentation en OCaml s'appuie sur l'arbre binaire défini au chapitre précédent :

```
type 'a bst = Vide | Noeud of 'a * 'a abr * 'a abr
```

C'est un type **polymorphe**, c'est-à-dire que les étiquettes sont de type générique `'a` mais possèdent un ordre total, c'est-à-dire on doit pouvoir comparer n'importe quel élément de l'ensemble avec un autre.

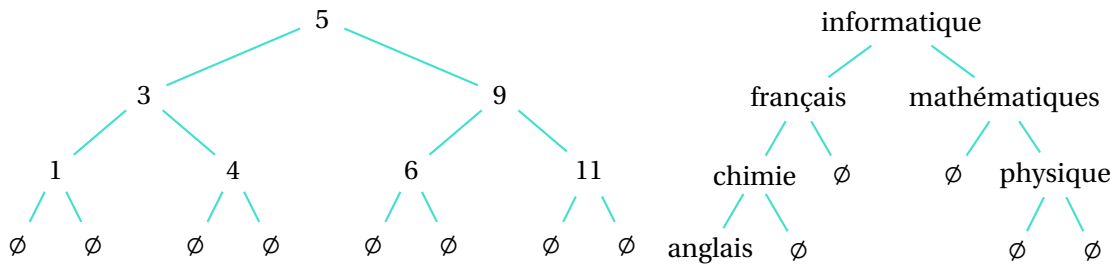


FIGURE 6.10 – Arbres binaires de recherche équilibré (à gauche) et non équilibré (à droite)

I Opérations sur les arbres binaires de recherche

Les principales opérations sur les arbres de recherche sont :

1. la recherche d'un élément, opération de complexité $O(h)$,
2. l'insertion d'un élément, opération de complexité $O(h)$,
3. la suppression d'un élément $O(h)$.

R Si l'arbre est un peigne, ces opérations deviennent de complexité $O(n)$, si n est le nombre de nœuds de l'arbre. C'est le pire des cas.

Le meilleur des cas se produit lorsque l'arbre est équilibré : ces opérations sont alors donc de complexité $O(\log n)$.

Toute la question est donc d'opérer sur un arbre binaire de recherche et, **simultanément**, de le maintenir dans un état équilibré pour obtenir des performances optimales.

R L'insertion ou la suppression dans un arbre binaire de recherche garantit que la structure d'arbre binaire est respectée à la fin de l'opération. L'équilibre de l'arbre n'est pas garanti.

Les figure 6.11 et 6.12 illustrent les opérations d'insertion et de suppression dans un arbre binaire.

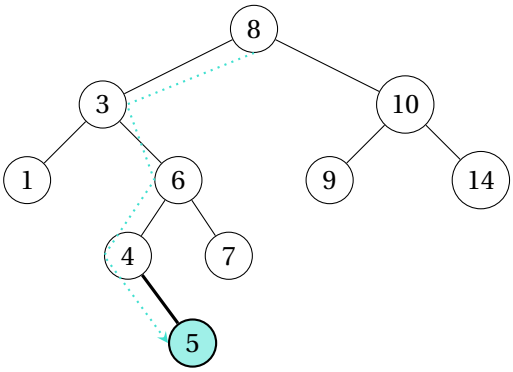
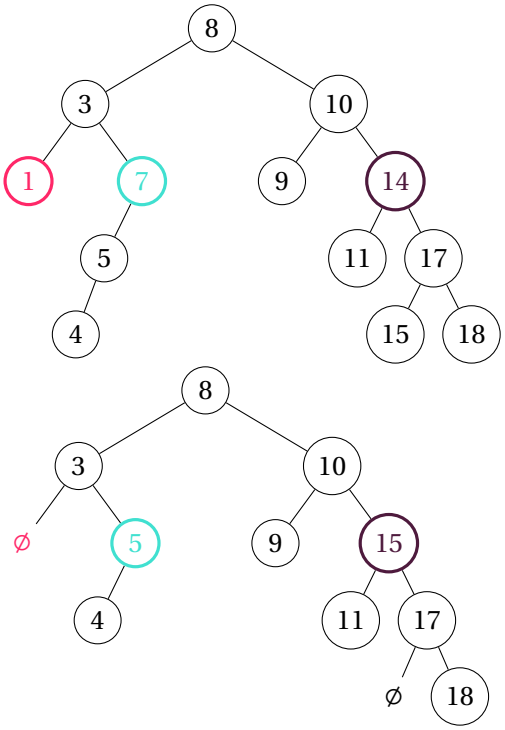


FIGURE 6.11 – Illustration de l'insertion d'un nœud d'étiquette 5 dans un arbre binaire de recherche



Trois cas sont possibles : le nœud qu'on retire

1. ne possède pas de fils,
2. possède un seul fils,
3. possède deux fils.

FIGURE 6.12 – Illustration de l'opération supprimer un nœud dans un arbre binaire de recherche

J Maintenir l'équilibre ⇢ HORS PROGRAMME

Le déséquilibre d'un arbre binaire équilibré est engendré par une opération d'insertion ou de suppression. Les techniques d'équilibrage s'appuient généralement sur les opérations d'insertion et de suppression normales suivies d'une manipulation de l'arbre pour lui redonner l'équilibre. Cette manipulation est souvent une rotation des sous-arbres.

■ **Exemple 59 — Arbres binaires de recherche automatiquement équilibrés.** On peut citer notamment :

1. Les arbres AVL,
2. Les arbres rouges et noirs.

7

ARBRES GÉNÉRIQUES ET PRÉFIXES

À la fin de ce chapitre, je sais :

- ☞ convertir un arbre générique en arbre binaire
- ☞ expliquer ce qu'est un arbre préfixe
- ☞ expliquer le principe du codage d'Huffmann
- ☞ implémenter un trie dans le cas où l'arbre est binaire

A Arbres génériques

■ **Définition 107 — Arbre générique .** Un arbre générique est un arbre qui possède un nombre d'enfants quelconque.

En OCaml on peut définir un arbre générique comme suit :

```
type 'a gtree =
    | Empty
    | Node of 'a * 'a gtree list;;
```

Les fils de l'arbre sont contenus dans une **liste de sous-arbres**.

R Un arbre générique peut se parcourir par niveau grâce à un parcours en largeur. Il peut également se parcourir en profondeur dans les ordres préfixes, infixes et postfixes.

Sur l'arbre générique de la figure 7.1, un parcours par niveau résulte en `['A' ; 'B' ; 'F' ; 'I' ; 'O' ; 'C' ; 'D' ; 'E' ; 'G' ; 'H' ; 'J' ; 'K' ; 'L' ; 'M' ; 'N']`.

De même, le parcours en profondeur préfixe résulte en `['A' ; 'B' ; 'C' ; 'D' ; 'E' ; 'F' ; 'G' ; 'H' ; 'I' ; 'J' ; 'K' ; 'L' ; 'M' ; 'N' ; 'O']`.

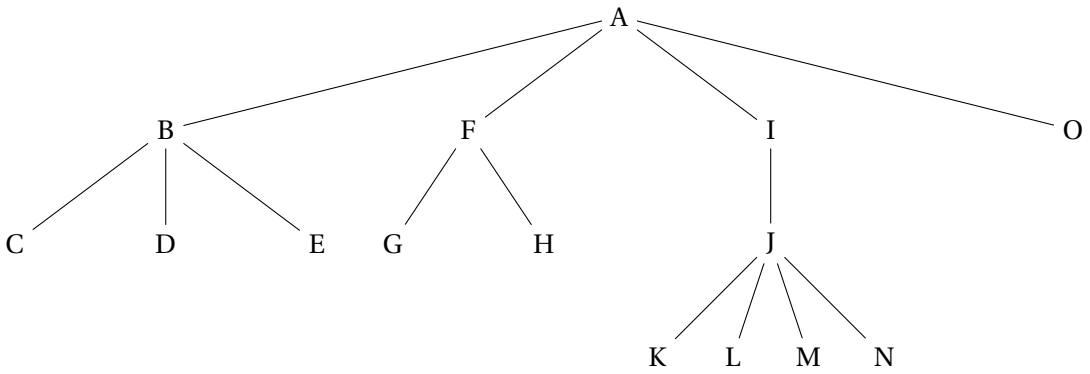


FIGURE 7.1 – Arbres générique

a Calculs sur un arbre générique

Pour calculer la hauteur d'un arbre générique, on doit évaluer chaque sous-arbre, c'est-à-dire traiter tous les sous-arbres présents dans la liste. Il est nécessaire de procéder par récursivité mutuelle ou bien en utilisant les fonctions `List.fold_left` et `max` comme le propose le code ci-dessous.

```
let rec h a =
        match a with
            | Empty -> -1
            | Node (_, []) -> 0
            | Node (_, fils) -> 1 + h_fils fils
    and h_fils fils =
        match fils with
            | [] -> 0
            | f::t -> max (h f) (h_fils t);;

(* Version folding *)
let rec hauteur a =
    match a with
        | Empty -> -1
        | Node (_, []) -> 0
        | Node (_, fils) -> 1 + List.fold_left (fun acc f -> max acc (hauteur f)) 0 fils
            ;;
```

b Transformer un arbre générique en arbre binaire

Il est possible de transformer un arbre générique en arbre binaire en adoptant la convention suivante : le fils gauche est un fils du nœud, le fils droit est un frère de ce fils. Avec cette convention, l'arbre générique de la figure 7.1 est transformé en l'arbre binaire de la figure 7.2

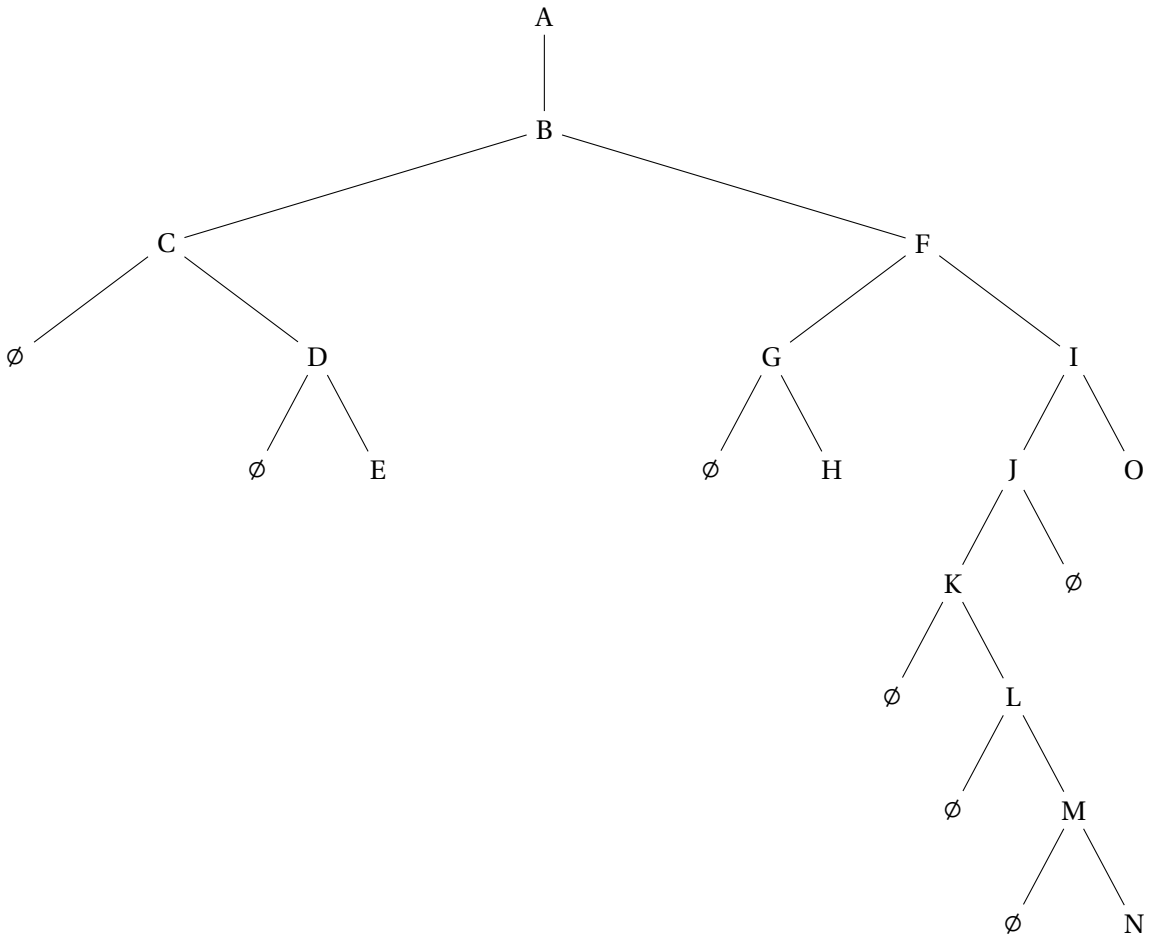


FIGURE 7.2 – Transformation de l'arbre générique de la figure 7.1 en arbre binaire

B Arbres préfixes

■ **Définition 108 — Arbre préfixe .** Soit $\mathcal{E}$ un ensemble d'étiquettes et $\mathcal{S}$ un ensemble de séquences sur ces étiquettes. L'arbre préfixe de $\mathcal{S}$ est un arbre enraciné tel que :

1. chaque **arête** est étiquetée par une étiquette de $\mathcal{E}$,
2. pour un nœud n et deux de ses fils g et d, l'arête (n, g) ne porte pas la même étiquette que l'arête (n, d),
3. à chaque séquence s de l'ensemble $\mathcal{S}$ correspond une feuille f de l'arbre telle que la concaténation des étiquettes des arêtes sur le chemin de la racine à f est égale à s.
4. de même, chaque chemin de la racine à une feuille correspond à une séquence.

R Un arbre préfixe garantit qu'une séquence ne peut pas être le préfixe d'une autre. L'avantage en termes de codage est considérable : le code ainsi généré est uniquement déchiffrable, il ne peut pas y avoir d'ambiguïté.

La figure 7.3 représente un arbre préfixe de chaînes de caractères : les chaînes sont les séquences de l'arbre.

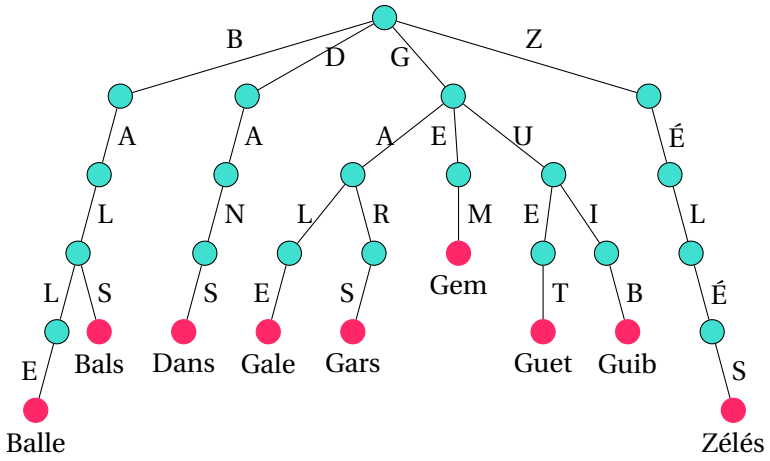


FIGURE 7.3 – Arbre préfixe que l'ensemble des séquences {Balle, Bals, Dans, Gale, Gars, Gem, Guet, Guib, Zélés}

La figure 7.4 représente un arbre d'Huffman binaire. L'ensemble des étiquettes est {0,1}, les nœuds sont étiquetées par les occurences et les feuilles par le symbole correspondant à la séquence binaire. Par exemple, pour coder la lettre A il faut utiliser le code binaire 010.

C Compression statistique

Un code de Huffman est un code optimal, au sens où il minimise la quantité d'information nécessaire pour encoder, préfixé de longueur variable : il est donc uniquement déchiffrable. Il se fonde sur les **statistiques** de la source à compresser et l'utilisation d'un arbre préfixe.

La procédure pour construire ce code est décrite par l'algorithme 2. Il s'agit tout d'abord de dénombrer les occurrences des symboles de la source puis de calculer le nombre d'occurrences associé à chacun. On construit ensuite une file de priorités qui contient chaque symbole à encoder associé à son nombre d'occurrences dans la source à compresser. La priorité de la file est basée sur le nombre d'occurrences : plus celui est faible, plus le symbole est prioritaire.

Tant que la file n'est pas vide, on construit l'arbre de Huffmann au fur et à mesure en partant des feuilles et en regroupant les plus petits éléments. Les nœuds intermédiaires ont pour étiquette la somme des occurrences de leurs fils.

Pour déchiffrer un fichier compressé avec l'algorithme 2, il est nécessaire de disposer des probabilités associées à chaque symbole de la source.

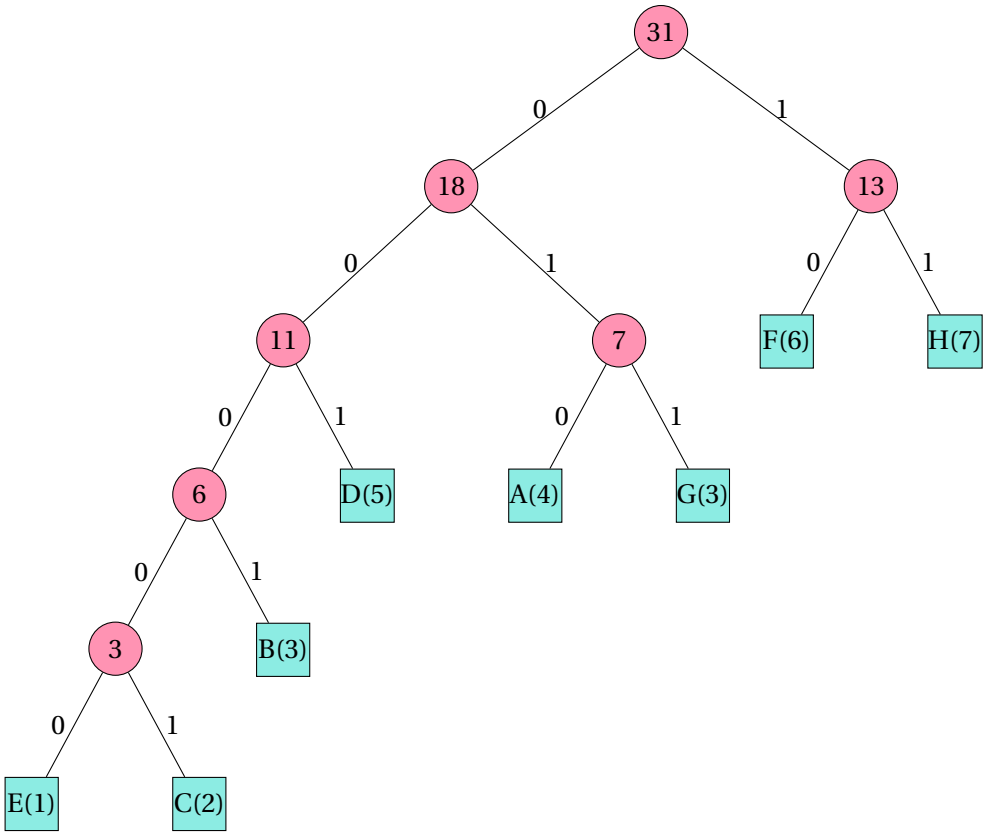


FIGURE 7.4 – Arbre d'Huffmann permettant de coder des caractères en binaire

Algorithme 2 Construction d'un arbre préfixe associé à un code de Huffmann

```
1 : Fonction HUFFMANN(S, O)                          ▷ S est un ensemble de symboles
2 :     F ← file de priorités                        ▷ O les occurrences de chaque de symboles
3 :     pour chaque couple (s, o) de S × O répéter
4 :         ENFILER(F, (FEUILLE(s),o))
5 :     tant que LONGUEUR(F) > 1 répéter
6 :         g, o_g ← DÉFILER(F)
7 :         d, o_d ← DÉFILER(F)
8 :         o ← o_g + o_d                            ▷ Addition des occurrences des sous-arbres
9 :         s ← SYMBOLE(g) + SYMBOLE(d)              ▷ Concaténation des symboles
10 :        parent ← NŒUD((s, o), g, d)              ▷ Création du nouveau nœud
11 :        ENFILER(F, (parent, o))
12 :    renvoyer DÉFILER(F)
```

Les algorithmes 3 et 4 donnent les procédures à suivre pour compresser et décompresser à l'aide d'un code d'Huffmann.

Algorithme 3 Compresser à l'aide d'un arbre de Huffmann

```
1 : Fonction H_COMPRESS(s, r)                    ▷ s un symbole source, r la racine l'arbre
2 :     si s est présent dans SYMBOLE(GAUCHE(r)) alors
3 :         renvoyer CONCAT(0, H_COMPRESS(s, GAUCHE(r)))
4 :     si s est présent dans SYMBOLE(DROIT(r)) alors
5 :         renvoyer CONCAT(1, H_COMPRESS(s, DROITE(r)))
6 :     renvoyer ϵ                                ▷ On retourne le mot vide ϵ
```

Algorithme 4 Décompresser à l'aide d'un arbre de Huffmann

```
1 : Fonction H_DECOMPRESS(m, r)                  ▷ m un mot codé, r la racine l'arbre
2 :     n ← r
3 :     d ← ϵ                                    ▷ Le mot vide
4 :     pour chaque bit b de m répéter
5 :         si n est une feuille alors
6 :             d ← CONCAT(d, SYMBOLE(n))
7 :             n ← r                            ▷ On a trouvé un symbole, on en cherche un autre
8 :         sinon si b = 0 alors
9 :             n ← GAUCHE(n)
10 :        sinon si b = 1 alors
11 :            n ← DROIT(n)
12 :        sinon
13 :            renvoyer ϵ                       ▷ Ce symbole binaire n'existe pas, erreur
14 :    renvoyer d
```

L'algorithme de Huffmann peut être décliné de plusieurs manières :

statique si les probabilités associées à chaque symboles sont figées et toujours les mêmes. Ainsi on n'a pas besoin de les calculer ni de les stocker (ou transmettre) avec les données compressées pour pouvoir le décompresser. Si les éléments à compresser sont des textes dans une seule langue, par exemple le français, cela peut se justifier, les probabilités étant connues statistiquement pour chaque langue.

semi-adaptatif si les probabilités associées aux éléments à compresser sont calculées et stockées (ou transmises) avec les données compressées. Les rendements sont meilleurs mais il faut stocker ou transmettre les probabilités.

adaptatif l'arbre est construit dynamiquement au fur et à mesure de la réception des symboles à coder. La complexité de l'algorithme de codage est plus grande mais le stockage ou la transmission de l'arbre n'est plus nécessaire. Les algorithmes FGK [6] et V [10] permettent de réaliser cet arbre de Huffman dynamique.

Les codes de Huffmann sont optimaux mais il présentent certains inconvénients :

- la version non adaptative de ces codes ne peut pas être synchrone à la transmission des données,
- ils sont très sensibles aux erreurs de transmission. Si on fait par exemple l'erreur de décoder un mot par un autre de longueur différente suite à une erreur de transmission alors cette erreur se propage naturellement à la suite du décodage : on interprètera mal les symboles suivants.

D Compression à dictionnaire

Issus des travaux initiaux de Jacob Ziv, Abraham Lempel et Terry Welch [11, 12, 13], ces codes ne se fondent pas sur les statistiques de la source mais consiste à **associer à des séquences de symboles une entrée dans un dictionnaire**. Le code est constitué par les clefs du dictionnaire. La position des clefs est la valeur associée à la clef dans le dictionnaire. L'opération de décompression consiste sélectionner les valeurs associées aux clefs du dictionnaire. On n'a donc pas besoin de connaître la source a priori pour utiliser ces algorithmes. De plus, on peut montrer qu'ils sont asymptotiquement optimaux.

Quatrième partie

Programmation récursive

Cinquième partie

Exploration et graphes

8

RETOUR SUR TRACE

À la fin de ce chapitre, je sais :

- ☞ expliquer le principe du retour sur trace
- ☞ donner des exemples d'utilisation
- ☞ coder un algorithme de retour sur trace en OCaml

A Exploration

Soit un problème $\mathcal{P}$ de satisfaction de contraintes tel que les solutions $\mathcal{S}$ se trouvent dans une ensemble fini $\mathcal{E}$ de candidats. On cherche à trouver les solutions de $\mathcal{P}$ en explorant l'ensemble $\mathcal{E}$ tout en respectant les contraintes.

Il est imaginable aujourd'hui d'envisager l'usage de la force brute pour résoudre des problèmes dont la dimension est pourtant élevée.

■ **Définition 109 — Recherche par force brute.** Énumérer tous les éléments candidats de $\mathcal{E}$ et tester s'ils sont solution de $\mathcal{P}$.

Il s'agit donc d'une approche simple à énoncer et à implémenter comme le montre l'algorithme 5.

Algorithme 5 Algorithme de recherche par force brute, problème de satisfaction de contraintes

```
1 : Fonction FORCE_BRUTE(ℰ)
2 :     𝒮 ← ∅
3 :     pour e ∈ ℰ répéter
4 :         si e est un solution de 𝒫 alors
5 :             𝒮 ← 𝒮 ∪ {e}
6 :     renvoyer 𝒮
```

■ **Exemple 60 — Exemples d'algorithmes de recherche par force brute.** Parmi les algorithmes de recherche par force brute utilisés, on note :

- la recherche d'un code secret à quelques chiffres : on teste toutes les permutations possibles jusqu'à trouver la bonne,
- la recherche d'un élément dans un tableau : on teste tous les éléments les uns après les autres jusqu'à trouver le bon,
- le tri bulle : on essaie de placer un élément dans une case, puis on essaie la case du dessus,
- quand on n'a pas d'autres idées...

■ **Exemple 61 — Problème des huit reines.** On cherche à placer sur un échiquier de 8x8 cases huit reines sans que celles-ci s'attaquent les unes les autres. Une solution est présentée sur la figure 8.1. On connaît les solutions de ce problème [1] et on peut même les formuler simplement.

En choisissant la recherche par force brute, il est nécessaire de tester $8^8 = 16777216$ configurations. Si le test de la validité de l'échiquier est effectué en moins d'une microseconde [a], l'intégralité des configurations sera examinée en un temps proche de la seconde.

a. ce qui est très réaliste avec une machine standard et un programme non optimisé

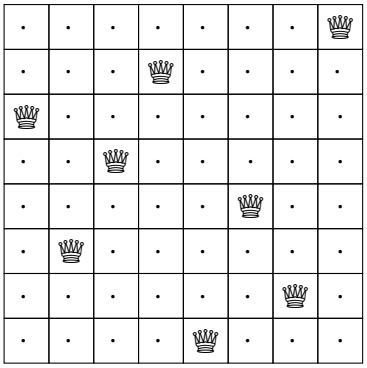


FIGURE 8.1 – Exemple d'échiquier 8x8 solution au problème des huit reines.

Même si elle est simple à énoncer et à implémenter, la recherche par force brute présente un inconvénient majeur : elle ne supporte pas le passage à l'échelle, c'est à dire qu'elle devient rapidement inutilisable à cause de l'explosion du cardinal de l'ensemble $\mathcal{E}$ à explorer qui induit un temps de calcul nécessaire rédhibitoire.

■ **Exemple 62 — Problème des n reines.** Le problème des n reines est la généralisation du problème des huit reines sur un échiquier de taille $n \times n$ et avec n reines à placer. L'ensemble des candidats est maintenant de taille n^n. Pour $n = 16$, le temps de calcul dépasse

déjà la dizaine de milliers d'années. En effet, admettons que le test de validité de l'échiquier soit toujours de l'ordre de la microseconde, on a : $(16^{16} \times 1^{-6})/(60 \times 60 \times 24 \times 365) \approx 584942$ années...

B Principe du retrour sur trace

Le retour sur trace est une technique exploratoire utilisée afin guider l'exploration et de ne pas tester toutes les configurations possibles.

■ **Définition 110 — Retour sur trace.** Le retour sur trace construit des ensembles de solutions partielles au problème $\mathcal{P}$. Ces solutions partielles peuvent être complétées de différentes manières pour former une solution au problème. La complétion des solutions se fait de manière incrémentielle.

Le retour sur trace utilise une représentation de l'espace des candidats $\mathcal{E}$ sous la forme d'un arbre de recherche. Chaque solution partielle est un nœud de cet arbre. Chaque solution complète forme un chemin descendant de la racine à une feuille de l'arbre. Au niveau de la feuille, on ne peut plus compléter la solution par quoi que ce soit.

L'algorithme de retour sur trace est un algorithme récursif qui parcours en profondeur l'arbre de recherche **en vérifiant qu'il peut compléter la solution partielle** par le nouveau nœud trouvé : si c'est le cas, alors il **continue** l'exploration de cette branche. Si ce n'est pas le cas, cette branche est **écartée**, car elle ne peut pas donner de solutions.

Vocabulary 5 — Backtracking ⇜⇝ Retour sur trace

Algorithme 6 Algorithme d'exploration des solutions avec retour sur trace

```
1 : Fonction EXPLORER(v)                          ▷ v est un nœud de l'arbre de recherche
2 :     si v est une feuille alors
3 :         renvoyer Vrai                          ▷ On a trvoué une solution
4 :     sinon
5 :         pour chaque fils u de v répéter
6 :             si u peut compléter une solution partielle au problème P alors
7 :                 EXPLORER(u)
8 :                                                ▷ Ici, on revient sur la trace!
9 :     renvoyer Faux
```

■ **Exemple 63 — Types de problèmes pour le retour sur trace.** Le retour sur trace s'applique couramment aux problèmes de satisfaction de contraintes tels que les problèmes :

- de décision : y-a-t-il une solution et si oui laquelle ?
- d'énumération : peut-on lister toutes les solutions à un problème donné ?
- d'optimisation : parmi toutes les solutions, y-en-a-t-il une optimale ?

■ **Exemple 64 — Retour sur trace sur le problème des quatre reines.** L'arbre de recherche nécessaire à l'exécution de l'algorithme de retour sur trace pour le problème des quatre reines est représenté sur la figure 8.2. La racine de l'arbre est le début de l'algorithme : on n'a pas encore placé de reines. Le première étape est le placement d'une reine sur l'échiquier : sur une même ligne on peut la placer sur quatre colonnes différentes qu'il va falloir tester. Chaque colonne représente donc une solution partielle différente et est un nœud fils de la racine. On peut placer la première reine sur n'importe quelle colonne.

La seconde étape est le placement d'une deuxième reine. Comme on parcours l'arbre en profondeur, on teste la première configuration en premier, c'est à dire une reine sur la première case de la première ligne. On teste alors toutes les solutions partielles possibles. Le premier nœud ne satisfait pas les conditions de validité : on ne peut pas placer une reine dans la première colonne car il y en a déjà une sur la première ligne. Tout le sous-arbre lié à cette solution partiel est élagué. De même pour le second nœud pour lequel la première reine peut attaquer en diagonale.

On revient donc là où on en était et on continue avec les autres fils valides.

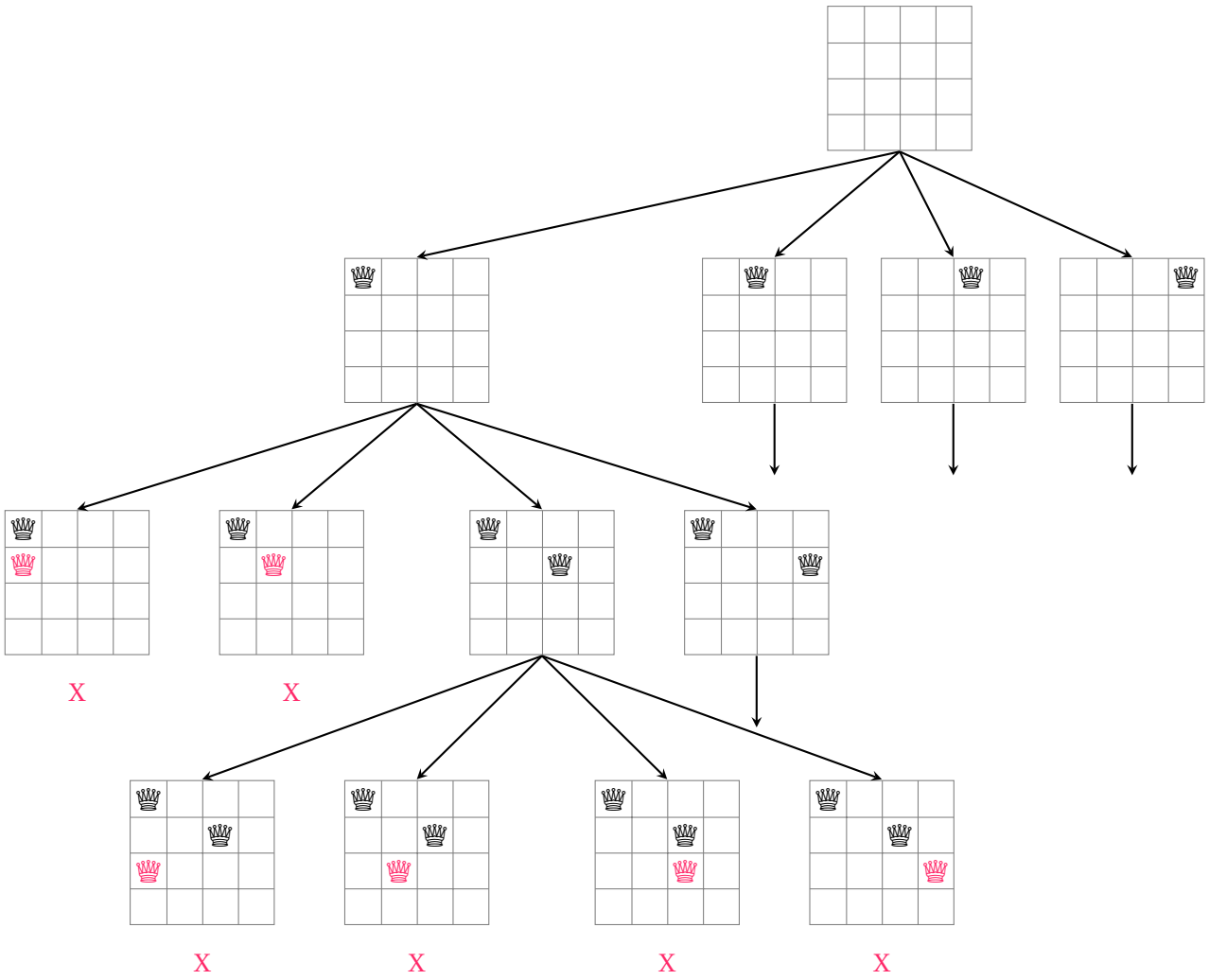


FIGURE 8.2 – Exemple d'arbre de recherche structurant l'algorithme de retour sur trace. Application au problème de quatre reines.

9

DES ARBRES AUX TAS

À la fin de ce chapitre, je sais :

- ☞ définir un tas-min et un tas-max
- ☞ expliquer l'algorithme du tri par tas
- ☞ utiliser un tas pour créer une file de priorité
- ☞ appliquer les files de priorités à l'algorithme de Dijsktra

A Tas binaires

a Définition

■ **Définition 111 — Tas max et tas min.** On appelle tas max (resp. tas min) un arbre binaire parfait étiqueté par un ensemble ordonné E tel que l'étiquette de chaque nœud soit inférieure (resp. supérieure) ou égale à l'étiquette de son père. La racine est ainsi la valeur maximale (resp. minimale) du tas.

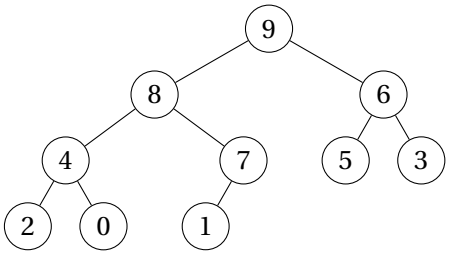


FIGURE 9.1 – Tas max

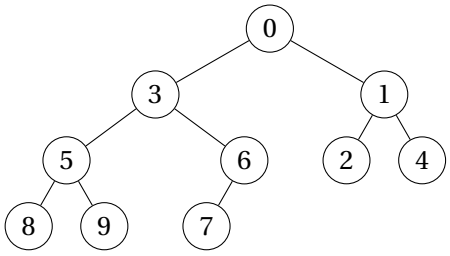


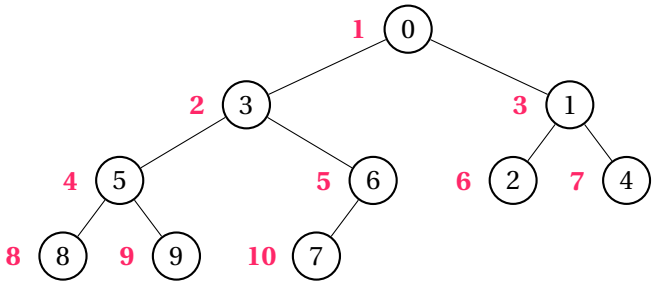
FIGURE 9.2 – Tas min

b Implémentation

On peut naturellement implémenter un tas par un type `a' arbre` mais également par un tableau `Array` en numérotant les nœuds selon la numérotation Sosa-Stradonitz (cf. figure 9.3).

R Un tas implémenté par un tableau est une structure de taille donnée, fixée dès la construction du tas : on ne pourra donc pas représenter tous les tas, uniquement ceux qui pourront s'inscrire dans le tableau. Par ailleurs, pour construire cette structure et la préserver lors de l'exécution d'algorithmes, il est important que l'on puisse faire évoluer les éléments à l'intérieur du tas. C'est pourquoi cette structure de donnée doit être muable.

■ **Définition 112 — Numérotation Sosa-Stradonitz d'un arbre binaire.** Cette numérotation utilise les puissances de deux pour identifier les nœuds d'un arbre binaire. La racine se voit attribuer la puissance 0. Le premier élément de chaque niveau k de la hiérarchie possède l'indice 2^k. Ainsi, sur le troisième niveau d'un arbre binaire, on trouvera les numéros $8, 9, 10, 11, 12, 13, 14$ et 15. Cette numérotation est utilisée dans le domaine de la généalogie.



Indices	1	2	3	4	5	6	7	8	9	10
Valeurs	0	3	1	5	6	2	4	8	9	7

FIGURE 9.3 – Implémentation d'un tas min par un tableau selon les indices de la numérotation Sosa-Stradonitz. On vérifie que les fils du nœud à l'indice k se trouvent à l'indice $2k$ et $2k+1$

R Comme, en informatique, les indices commencent à 0, on choisit souvent la convention de positionner la racine du tas dans la case d'indice 0 et décaler ensuite tous les indices de 1. Les fils d'un nœud d'indice k se situe alors en $2k+1$ et $2k+2$.

c Opérations

On s'intéresse à la construction et à l'évolution d'un tas au cours du temps : comment préserver la structure de tas lorsqu'on ajoute ou retire un élément ?

On définit des opérations *descendre* et *faire monter* un élément dans un tas qui préservent la structure du tas. Ce sont des opérations dans un tas sont des opérations dont la complexité est $O(h(a)) = O(\log n)$.

Faire monter un élément dans le tas

Cette opération est expliquée sur la figure 9.4.

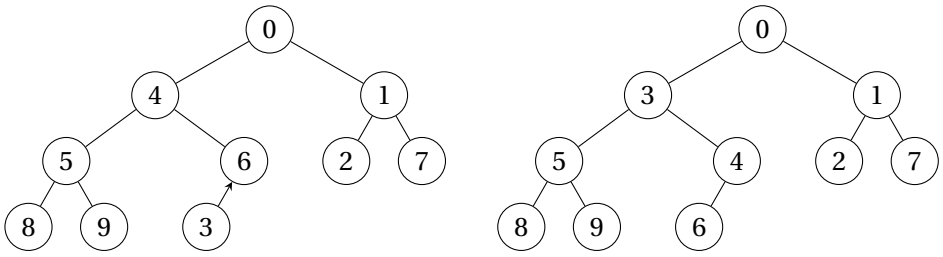


FIGURE 9.4 – Tas min : à gauche, l'élément 3 doit monter dans le tas min. À droite, on a échangé les places de 3 avec les pères (6 puis 4) jusqu'à ce que la structure soit conforme à un tas min

Faire descendre un élément dans le tas

Faire descendre dans le tas se dit aussi tamiser le tas et est expliqué sur la figure 9.5.

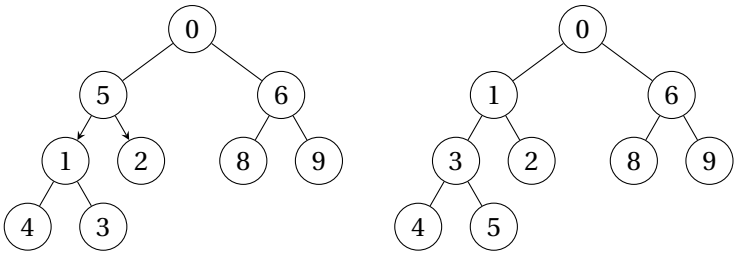


FIGURE 9.5 – Tas min : à gauche, l'élément 5 doit descendre dans le tas min. À droite, on a échangé la place de 5 avec le fils le plus petit (d'abord 1 puis 3) pour que la structure soit conforme à un tas min

Construire un tas

On peut imaginer construire un tas en faisant monter les éléments ou en faisant descendre les éléments au fur et à mesure. Ces deux méthodes ne présentent pas la même complexité .

1. Si l'on procède en faisant monter les éléments, on considère que la racine est un tas à un élément et on intègre les éléments restant du tableau dans ce tas en les faisant monter. Dans le pire des cas, on doit faire monter les $n-1$ nœuds depuis le niveau de profondeur maximale (hauteur de l'arbre) jusqu'à la racine et donc répéter l'opération monter (de complexité $\log n$). C'est pourquoi cette méthode présente une complexité en $O(n \log n)$.
2. La seconde méthode considère que chaque feuille est un tas à un élément. On fait descendre les $\lfloor n/2 \rfloor$ premiers éléments à partir du $\lfloor n/2 \rfloor$ e dans les tas. Au fur et à mesure, on réunit ces tas en un plus gros tas. Dans le pire des cas, on peut montrer que cette méthode est linéaire en $O(n)$. En effet, un élément de hauteur i descend dans le pire des cas $h-1-i$ pour trouver sa place. $C(h) = \sum_{i=0}^{h-1} 2^i (h-1-i)$ car à la hauteur i il y a au plus 2^i éléments. Donc [1] $C(h) = 2^h - h - 1 = O(2^h)$ Or, dans un arbre binaire parfait, on a $2^h \leqslant n \leqslant 2^{h+1}$ si n est la taille de l'arbre. Finalement, $C(h) = O(n)$. Il est donc plus efficace de faire descendre les éléments pour construire un tas.

R Dans un tas de taille n, la première feuille se situe à l'indice $\lfloor n/2 \rfloor$.

B Tri par tas binaire

■ **Définition 113 — Tri par tas.** Le tri par tas procède en formant d'un tas à partir du tableau à trier. Pour un tri ascendant, on utilise un tas-max et pour un tri descendant un tas-min.

On peut considérer que cette méthode est une amélioration du tri par sélection : le structure de tas permet d'éviter la recherche de l'élément à sélectionner.

Le tri par tas est un tri comparatif en place et non stable. Sa complexité dans le pire des cas est en $O(n \log n)$ car il nécessite au pire de descendre les n éléments au niveaux des feuilles dans le tas.

Vocabulary 6 — Heap sort ⇔ Tri par tas

L'algorithme 7 fait appel à un tas-max et son implémentation est radicalement simple, tout le cœur du mécanisme de tri reposant sur la structure de tas.

1. à vous de calculer !

Algorithme 7 Tri par tas, ascendant

```
1 : Fonction TRI_PAR_TAS(t)
2 :     n ← nombre d'éléments de t
3 :     Faire un tas-max de t
4 :     pour k de n − 1 à 0 répéter
5 :         Échanger t[0] et t[k]                    ▷ Le plus grand va à la fin
6 :         Faire descendre t[0] dans le tas t[: k]   ▷ Le nouvel élément descend
7 :     renvoyer t
```

C File de priorités implémentée par un tas

Une autre application des tas binaires est l'implémentation d'une file à priorités.

■ **Définition 114 — TAD File de priorités.** Une file de priorités est une extension du TAD file dont les éléments sont à valeur dans un ensemble $E = (V, P)$ où P est un ensemble totalement ordonné. Les éléments de la file sont donc des couples valeur - priorité.

Les opérations sur une file de priorités sont :

1. créer une file vide,
2. insérer dans la file une valeur associée à une priorité (ENFILER),
3. sortir de la file la valeur associée la priorité maximale (ou minimale) (DÉFILER).

L'utilisation d'un tas permet d'obtenir une complexité en $O(\log n)$ pour les opérations DÉFILER et ENFILER d'une file de priorités. Pour des algorithmes comme celui du plus court chemin de Dijkstra, c'est une solution intéressante pour améliorer les performances de l'algorithme.

10

GRAPHES AVANCÉS

À la fin de ce chapitre, je sais :

- ☞ expliquer le concept d'arbre recouvrant
- ☞ connaître les algorithmes de Prim et Kruskal
- ☞ connaître la notion de tri topologique et son lien avec le parcours en profondeur
- ☞ expliquer le concept de forte connexité
- ☞ expliquer l'intérêt d'un graphe biparti

A Graphes connexes et acycliques

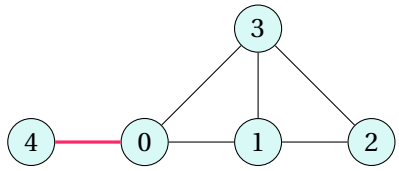
■ **Définition 115 — Graphe.** Un graphe G est un couple $G = (S, A)$ où S est un ensemble fini et non vide d'éléments appelés **sommets** et A un ensemble de paires d'éléments de S appelées **arêtes**.

■ **Définition 116 — Sous-graphe.** Soit $G = (S, A)$ un graphe, alors $G' = \{S', A'\}$ est un sous-graphe de G si et seulement si $S' \subseteq S$ et $A' \subseteq A$.

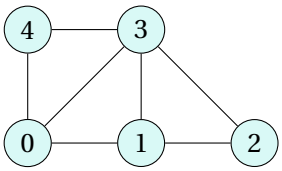
■ **Définition 117 — Sous-graphe couvrant.** G' est un sous-graphe couvrant de G si et seulement si G' est un sous-graphe de G et $S' = S$.

■ **Définition 118 — Graphe connexe.** Un graphe $G = (S, A)$ est connexe si et seulement si pour tout couple de sommets (a, b) de G, il existe une chaîne d'extrémités a et b.

■ **Définition 119 — Composantes connexes d'un graphe.** Un graphe non connexe est l'union de plusieurs sous-graphes connexes qui n'ont de sommet en commun. Les sous-graphes connexes disjoints sont appelés les composantes connexes du graphe.



Cas où un sommet au moins est de degré 1.



Cas où tous les sommets sont de degré supérieur ou égal à deux.

FIGURE 10.1 – Disjoinction des cas pour la démonstration du théorème 10

Théorème 10 — Un graphe connexe d'ordre n possède au moins $n-1$ arêtes..

Démonstration. Par récurrence sur l'ordre du graphe.

(Initialisation) Soit G un graphe d'ordre 1. Ce graphe est trivialement connexe.

(Hérédité) Supposons la propriété vraie pour les graphes d'ordre $n-1$. Soit $G = (S, A)$ un graphe d'ordre n. La figure 10.1 illustre la disjonction des cas ci-dessous.

- Si G possède un sommet d'ordre 1, alors en supprimant l'arête qui lui est connectée, on obtient deux commposantes connexes, dont une à $n-1$ sommets. Par hypothèse de récurrence, cette composante connexe à $n-1$ sommets possède au moins $n-2$ arêtes. Donc G possède au moins $n-1$ arêtes.
- Si G ne possède pas de sommets d'ordre 1, cela signifie qu'il possède des sommets qui sont au moins d'ordre 2. D'après le lemme des poignées de main, cela signifie que le nombre d'arêtes m de G vaut :

$$m = \frac{\sum_{s \in S} deg(s)}{2} \geqslant \frac{2n}{2} = n$$

Le graphe G possède donc au moins n arêtes.

(Conclusion) Comme la proprité est vraie pour $n = 1$ et que l'hérédité est vérifiée, on en conclut qu'elle est vraie pour tous les graphes connexes.

■

Lemme 1 — Un graphe dont tous les sommets sont de degré supérieur ou égal à deux possède au moins un cycle.

Démonstration. Soit G un graphe d'ordre n dont tous les sommets sont au moins de degré 2. D'après le théorème précédent et le lemme des poignées de mains, le nombre d'arêtes de G est supérieur à n et donc G est connexe.

Soit s_0 un sommet quelconque. On parcours G en profondeur à partir de s_0. Ce parcours $\mathcal{P}$ est une suite finie de sommets, chaque sommet étant découvert par adjacence et visité une seule fois. Tous les sommets, quelque soit s_0, sont visités une fois car le graphe est connexe.

Comme l'ordre d'un graphe est fini, $\mathcal{P}$ va se terminer, par exemple sur le sommet z. On a découvert z par une arête du graphe G, mais on n'en est pas sorti. Comme tous les sommets sont au moins de degré 2, cela signifie qu'il existe une autre arête incidente à z qui connecte ce sommet à un autre sommet x de G, sommet déjà découvert par le parcours en profondeur. L'arête (z,x) forme donc un cycle dans le graphe. G possède nécessairement un cycle. ■

Théorème 11 — Un graphe acyclique possède au plus $n-1$ arêtes.

Démonstration. Par récurrence sur l'ordre du graphe.

(Initialisation) Un graphe d'ordre 1 est trivialement acyclique.

(Hérédité) Supposons la propriété vraie pour un graphe d'ordre $n-1$. Soit G un graphe acyclique à n sommets. D'après le lemme précédent, G possède au moins un sommet dont le degré est 0 ou 1.

- S'il est de degré 0, alors le graphe n'est pas connexe et les composantes acycliques restantes comportent $n-1$ sommets et au plus $n-2$ arêtes, d'après notre hypothèse de récurrence.
- S'il est de degré 1, en retirant l'arête qui le relie au reste du graphe, on obtient au moins deux composantes acycliques dont une possédant $n-1$ sommets, qui par hypothèse de récurrence comporte donc au plus $n-2$ arêtes.

Dans tous les cas, G possède donc au plus $n-1$ arêtes.

(Conclusion) La propriété étant vérifié à l'ordre 1 et l'hérédité étant conservée, la propriété est vraie pour tous les graphes acycliques.

■

■ **Définition 120 — Arbre.** Un arbre est un graphe **non orienté connexe et acyclique**.

Théorème 12 — Un arbre à n sommets possède exactement $n-1$ arêtes.

Démonstration. On utilise les résultats précédents : un arbre possède au moins $n-1$ arêtes et au plus $n-1$ arêtes. Il en possède donc exactement $n-1$. ■

■ **Exemple 65 — Exercices types.** Soit $G=(S,A)$ un graphe non orienté d'ordre n.

1. Soit $G=(S,A)$ un graphe non orienté d'ordre n. Montrer que si G est connexe alors il comporte au moins $n-1$ arêtes. (Procéder par récurrence)
2. Soit $G=(S,A)$ un graphe non orienté d'ordre n. Montrer que si G est connexe et a tous les sommets de degré supérieur ou égal à 2 alors il possède un cycle. En déduire, qu'un graphe acyclique admet un sommet de degré 0 ou 1.

3. Soit $G = (S, A)$ un graphe non orienté d'ordre n. Montrer que que si G est acyclique alors il possède au plus $n-1$ arêtes. (Procéder par récurrence)
4. Montrer que la modélisation planaire des alcanes de formule C_nH_{2n+2} est un arbre. (Utiliser le lemme de l'étoile)
5. Montrer qu'un graphe d'ordre n non connexe possède au plus $\frac{(n-1)(n-2)}{2}$ arêtes.
6. On souhaite interconnecter les sept plus grandes villes de Bretagne à l'aide d'un réseau à très haut débit. Trois villes possèdent au moins deux liens les reliant à deux autres autres villes. Toutes les villes sont-elles interconnectées ?

Théorème 13 — Caractérisation des arbres. Soit G un graphe non orienté à n sommets. Les propositions suivantes sont équivalentes :

1. G est un arbre.
2. G est connexe et possède $n-1$ arêtes.
3. G est acyclique et possède $n-1$ arêtes.

Démonstration. Soit G un graphe non orienté à n sommets.

- Si G est un arbre, c'est un graphe connexe qui possède $n-1$ arêtes, car également acyclique. Donc 1. implique 2.
- Supposons que 2. est vraie. Supposons que G possède un cycle. Alors on peut retirer une arête et obtenir un graphe à n sommet connexe qui possède $n-2$ arêtes. Ceci est absurde d'après le théorème 10. Donc, G est acyclique et 2. implique 3.
- Supposons que 3. est vraie. S'il ne possède qu'une seule composante connexe, alors G est acyclique et connexe. C'est donc un arbre et 1. est vraie. S'il possède m composantes connexes $G_i = (S_i, A_i)$, alors comme les G_i sont connexes et acycliques, on peut écrire :

$$|A| = \left(\sum_{i=1}^{m} |S_i|\right) - m = |S| - m = n - 1$$

  Par identification, on trouve que m vaut 1. Le graphe G est donc connexe et acyclique : c'est un arbre.

Comme 1. $\Rightarrow$ 2. $\Rightarrow$ 3. $\Rightarrow$ 1., on en déduit que ces trois propositions sont équivalentes. ■

■ **Définition 121 — Forêt.** Une forêt est un graphe non orienté sans cycle dont chaque composante connexe est un arbre.

B Arbres recouvrants

■ **Définition 122 — Arbre recouvrant un graphe.** Dans un graphe non orienté et connexe, un arbre est dit recouvrant s'il est inclus dans ce graphe et s'il connecte tous les sommets de ce graphe.

R On peut dire de manière équivalente qu'un arbre recouvrant est :

- un sous-graphe acyclique maximal,
- un sous-graphe recouvrant connexe minimal.

■ **Définition 123 — Arbre recouvrant minimal.** Un arbre recouvrant minimal est un arbre recouvrant un graphe dont la somme des poids des arêtes est minimale.

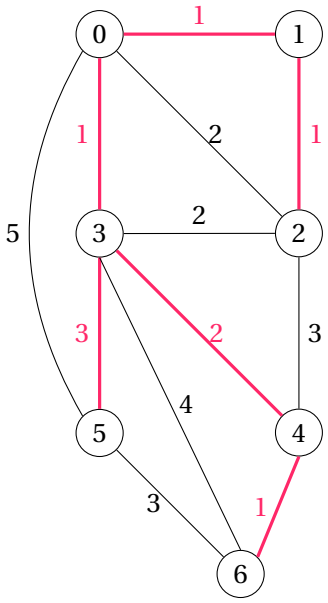


FIGURE 10.2 – Exemple d'arbre recouvrant dans un graphe non orienté et pondéré. En rouge, un arbre recouvrant minimal de poids total 9.

Les arbres recouvrants sont des éléments essentiels de monde réel car les applications sont nombreuses. La figure 10.2 peut par exemple représenter un réseau de villes qu'on souhaite interconnecter par un chemin de fer. Si le coût de construction est proportionnel au poids porté par l'arête, comment choisir les arêtes pour minimiser le coût de la construction du réseau de voies ferrées tout en interconnectant toutes les villes? Les arêtes en rouges, qui forment un arbre recouvrant de poids minimal, permettent de répondre à cette question.

Les arbres recouvrants sont utilisés dans les réseaux d'énergie, de télécommunications, de fluides, de transport, de construction mais également en intelligence artificielle et en conception de circuits électroniques. Les deux algorithmes phares pour construire des arbres recouvrants **minimaux** sont l'algorithme de Kruskal [7] et l'algorithme de Prim [8]. L'algorithme de Wilson permet quant à lui de générer des arbre recouvrants aléatoires.

■ **Définition 124 — Coupe de graphe.** On appelle coupe d'un graphe $G = (S, A)$

- une partition de l'ensemble des sommets S en deux sous-ensembles S_1 et S_2 non vides tels que $S = S_1 \cup S_2$ et $S_1 \cap S_2 = \emptyset$,
- ou bien l'ensemble des arêtes possédant une extrémité dans chaque sous-ensemble de la partition.

Le poids de la coupe est :

- soit la somme des poids respectifs des arêtes de la coupe si G est pondéré,
- soit le nombre d'arêtes dans la coupe.

Une arête de la coupe est dite traversante.

Théorème 14 — Propriété de la coupe. Soit $G = (S, A, w)$ un graphe pondéré. Soit C une coupe de ce graphe. Alors l'arête de poids minimum de la coupe fait partie de tous les arbres recouvrants minimum du graphe.

Démonstration. Par l'absurde. La figure 10.3 illustre l'objet de la propriété.

Soit $G = (S, A, w)$ un graphe pondéré. Soit $C = \{S1, S2\}$ une coupe de ce graphe. Soit $a_{\min} = (s_1, s_2)$, l'arête de poids minimum de la coupe. Supposons qu'il existe un arbre recouvrant minimum $\mathcal{A}$ de G ne contenant pas $a_{\min}$.

$\mathcal{A}$ est un graphe connexe et donc on peut y trouver un chemin qui connecte s_1 et s_2. Comme $a_{\min}$ n'est pas une arête de $\mathcal{A}$, il existe sur ce chemin de s_1 à s_2 une arête $e = (u, v)$ telle que $u \in S_1$ et $v \in S_2$ et que $w(e) > w(a_{\min})$. Alors $(\mathcal{A} \setminus \{e\}) \cup \{a_{\min}\}$ est un arbre revouvrant de G dont le poids est inférieur à celui de $\mathcal{A}$. Ce qui est absurde puisqu'on a supposé que $\mathcal{A}$ était un arbre recouvrant minimum. ■

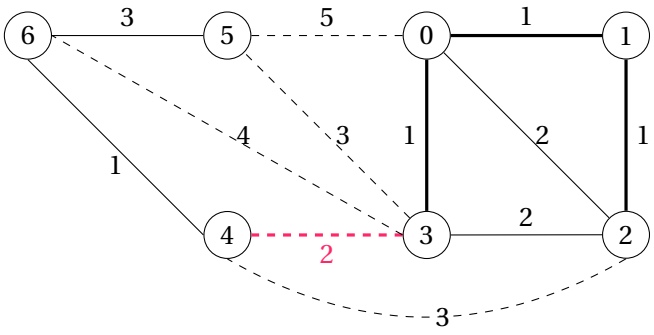


FIGURE 10.3 – Illustration de la propriété de la coupe. En pointillé, les arêtes de la coupe $(\{0,1,2,3\},\{4,5,6\})$. L'arête $(3,4)$ en rouge fait partie de tous les arbres recouvrants minimaux du graphe.

Théorème 15 — Propriété du cycle. Soit $G = (S, A, w)$ un graphe pondéré possédant un cycle. L'arête de poids maximum de ce cycle ne peut pas faire partie d'un arbre recouvrant

minimum du graphe.

Démonstration. Par l'absurde.

Soit $G = (S, A, w)$ un graphe pondéré possédant un cycle dont l'arête de poids maximum est $a_{\max} = (u, v)$. Supposons qu'il existe un arbre recouvrant minimum $\mathcal{A}$ de G contenant $a_{\max}$.

L'arête $a_{\max}$ fait partie d'un cycle dans G. Pour construire $\mathcal{A}$, on a ôté une arête e de ce cycle. Considérons $\mathcal{A} \setminus \{a_{\max}\} \cup \{e\}$: ce graphe est un arbre dont le poids est inférieur à celui de $\mathcal{A}$, car $w(e) < w(a_{\max})$. Ce qui est absurde puisqu'on a supposé que $\mathcal{A}$ était un arbre recouvrant minimum. ■

■ **Exemple 66 — Exercices types.**

- Montrer que l'arbre recouvrant minimal n'engendre pas nécessairement le plus cours chemin entre deux sommets donnés. (Donner un exemple)
- Caractérisation de la suite des degrés d'un arbre. Soit G un graphe connexe non orienté à $n \geq 2$ sommet.
  1. Montrer que G est un arbre si et seulement si la suite des degrés $(d_1, d_2, \ldots, d_n) \in (\mathbb{N}^*)^n$ des sommets de l'arbre vérifie :
  $$\sum_i d_i = 2(n-1)$$
  2. Représenter tous les arbres d'ordre inférieur à 5.

a Algorithme de Prim

L'algorithme de Prim est un algorithme **glouton optimal** qui s'applique aux **graphes pondérés connexes** et qui construit un arbre recouvrant minimal. Pour construire l'arbre, l'algorithme part d'un sommet et fait croitre l'arbre en choisissant une arête dont l'extrémité de départ appartient à l'arbre et mais pas l'extrémité d'arrivée **et** dont le poids est le plus faible, garantissant ainsi l'absence de cycle et la minimalité.

Algorithme 8 Algorithme de Prim, arbre recouvrant

```
1 : Fonction PRIM(G = (V, E, w))
2 :     T ← ∅                          ▷ la sortie : l'ensemble des arêtes de l'arbre recouvrant
3 :     S ← s un sommet quelconque de V
4 :     tant que S ≠ V répéter
5 :         (u, v) ← min(w(u, v), u ∈ S, v ∈ V \ S)          ▷ Choix glouton !
6 :         S ← S ∪ {v}
7 :         T ← T ∪ {(u, v)}
8 :     renvoyer T
```

Démonstration.

TERMINAISON. Soit v la fonction définie par :

$$\begin{aligned} v : \mathbb{N} &\longrightarrow \mathbb{N} \\ |S| &\longmapsto |V| - |S| \end{aligned}$$

La fonction v est un variant de boucle pour la boucle *tant que* de l'algorithme de Prim. En effet, un graphe possède toujours un nombre fini de sommets, donc $|V|$ est un entier naturel. Au premier tout de boucle, $|S| = 1$ et à chaque tour le cardinal de S augmente de 1. Donc, v est une fonction à valeurs entières **strictement** décroissante. Elle atteint donc 0 (théorème de la limite monotone) lorsque la condition de boucle est invalidée, c'est-à-dire lorsque $|S| = |V|$. L'algorithme de Prim se termine.

CORRECTION. Soit $G = (V, E, w)$ un graphe pondéré connexe d'ordre n.

Le choix de v se fait dans $V \setminus S$ ce qui exclue la construction de cycles dans (S, T) et garantit qu'on n'ajoute un sommet qu'une seule fois à S. Par ailleurs, le choix de l'arête se fait au départ de S, ce qui garantit la connexité dans la construction de (S, T). (S, T) est donc un graphe connexe acyclique, c'est-à-dire un arbre. Cet arbre est recouvrant puisque, à la fin de la boucle, $S = V$.

Il reste à démontrer que l'arbre est poids **minimal**. On peut le montrer en utilisant la propriété de la coupe 14 et en exhibant un invariant de boucle

On considère la propriété $\mathcal{J}$ suivante : « Le sous-graphe $G' = (S, T, w)$ est un arbre recouvrant minimal de (S, E, w). » On montre que c'est un invariant pour la boucle **tant que**.

À l'entrée de la boucle $G' = (\{s\}, \emptyset, w)$, on a $|G'| = 1$, T est vide et on a $|T| = 0 = 1 - 1$. Donc, G' est minimal car le poids de l'arbre est nul.

Conservation Supposons que la propriété soit vraie au début d'une itération.

Soit $a_{min} = (u, v)$ l'arête de poids le plus faible dans la coupe $(S, V \setminus S)$.

À la fin de l'itération, $G'' = (S \cup \{v\}, T \cup \{u, v\}, w)$. G'' est un graphe connexe et acyclique car on choisit de relier un sommet de S à un autre sommet de $V \setminus S$, donc G'' est un arbre recouvrant.

On considère la coupe $(S, \{v\})$ de $(S \cup \{v\}, E, w)$. a_{min} est également l'arête de poids le plus faible de $(S \cup \{v\}, E, w)$. D'après la propriété de la coupe, l'arête a_{min} fait partie de tous les arbres recouvrants minimaux de $(S \cup \{v\}, E, w)$. Comme G' est un arbre couvrant minimal de (S, E, w) à l'entrée de la boucle et qu'on a ajouté une arête faisant parti de tous les arbres minimaux de $(S \cup \{v\}, E, w)$, alors G'' est une arbre couvrant minimal de de $(S \cup \{v\}, E, w)$.

Conclusion La propriété est donc vraie à l'entrée de la boucle et invariante par les instructions de la boucle. On en conclut que tous les sous-graphes $G' = (S, T, w)$ sont des arbres revouvrants minimaux de de (S, E, w). À la sortie de la boucle, $S = V$: $G' = (V, T, w)$ est donc un arbre recouvrant minimal de $G = (V, E, w)$.

■

R L'algorithme de Prim est une preuve constructive que tout graphe connexe possède au moins un arbre recouvrant minimal.

R La connexité de l'arbre est garanti par le choix de v dans $V \setminus S$. C'est en même temps une limitation : cet algorithme ne fonctionnerait pas si le graphe n'était pas connexe car certains sommets seraient inaccessibles.

R Si tous les poids du graphe sont différents, T est unique car il n'y a pas d'ambiguité dans le choix de l'arête de poids le plus faible.

R La complexité de cet algorithme, si l'on utilise un tas binaire pour la file de priorités et une liste d'adjacence pour représenter G, est en $O((n+m)\log n)$ si $m = |E|$ est le nombre d'arêtes du graphe et n l'ordre du graphe.

R Quelles sont les différences entre Dijkstra et Prim ? En pratique, l'algorithme de Dijkstra est utilisé lorsque l'on souhaite économiser du temps et du carburant pour se déplacer d'un point à un autre. L'algorithme de Prim, quant à lui, est utilisé lorsque l'on souhaite minimiser les coûts de matériaux lors de la construction de routes reliant plusieurs points entre eux.

Les algorithmes de Prim et de Dijkstra présentent trois différences majeures :

- Dijkstra trouve le chemin le plus court, mais l'algorithme de Prim trouve le l'arbre recouvrant minimal. On peut le vérifier rapidement sur la figure 10.2.
- Dijkstra s'applique sur les graphes orientés et non orientés mais Prim ne s'applique qu'à des graphes pondérés non orientés.
- Prim peut gérer des pondérations négatives alors que Dijkstra ne l'admet pas.

b Algorithme de Kruskal

L'algorithme de Kruskal (cf. algorithme 9) est un algorithme **glouton optimal** qui s'applique aux **graphes pondérés** et qui construit une forêt d'arbres recouvrants minimaux. Le graphe peut ne pas être connexe. Pour construire la forêt et effectuer un choix d'arête, il ordonne les arêtes du graphe par ordre de pondération croissante.

Démonstration. Il est facile de montrer que (S, T, w) :

- est une forêt, car aucun cycle créé,
- est une forêt recouvrante : s'il existait un sommet non recouvert, c'est-à-dire non connecté à (S, T, w), cela signifierait qu'il n'est connecté par aucune arête au reste du graphe (sommet isolé, c'est donc une composante connexe de la forêt) ou que l'arête qui le relie n'a pas été considérée. Cette dernière solution est impossible puisqu'on les considère toutes en m itérations.

En utilisant la propriété de la coupe 14, on montre qu'à chaque itération, (S, T, w) est inclus dans une forêt recouvrante minimale de G. ■

Algorithme 9 Algorithme de Kruskal, arbre recouvrant

```
1 : Fonction KRUSKAL(G = (V, E, w))
2 :     T ← ∅                          ▷ la sortie : l'ensemble des arêtes de la forêt recouvrante
3 :     pour k de 1 à |E| répéter
4 :         e ← l'arête de pondération la plus faible de E          ▷ Choix glouton !
5 :         si (S, T ∪ {e}, w) est un graphe acyclique alors
6 :             T ← T ∪ {e}
7 :         E ← E \ e                  ▷ On n'a plus à tester cette arête
8 :     renvoyer T
```

La complexité de cet algorithme, si on utilise une structure unir-trouver, est en $O(m \log n)$ si $m = |E|$ est le nombre d'arêtes du graphe et n l'ordre du graphe.

c Unir et trouver --→ HORS PROGRAMME

La principale difficulté de l'algorithme de Kruskal réside au niveau du test d'acyclicité : comment savoir si le nouvel arbre qu'on projette de construire est bien acyclique ? On pourrait utiliser un parcours en profondeur car dès qu'on trouve un sommet déjà découvert un cycle est détecté. Cependant ce n'est pas optimal au niveau de la complexité. C'est pourquoi il est préférable d'utiliser une structure de type Union-Find (UF) : c'est une structure très efficace qui permet de réunir des ensembles disjoints en les étiquetant sous la même étiquette. On distingue ainsi les sommets du graphe qui sont connexes dans l'arbre en cours de construction et des autres.

■ **Définition 125 — Unir-trouver.** Unir-trouver est une structure de données qui représente une partition d'un ensemble fini ou de manière équivalente une relation d'équivalence.

Un ensemble est composé de n éléments représentés par des entiers de 0 à $n-1$. Une partie de l'ensemble est identifiée par un des éléments qui la compose. Au départ, une structure unir-trouver est initialisée de telle manière que chaque élément de l'ensemble appartient à sa propre partie.

La structure unir-trouver possède deux opérations :

Trouver (find) détermine à quelle partie appartient un élément. Cette opération permet de déterminer si deux éléments appartiennent à la partie.

Unir (union) réunit deux parties de la partition en une seule.

Pour être efficace, cette structure est composite et composée :

- d'un tableau `parent` dans laquelle on maintient l'identité de la partie à laquelle appartient l'élément : `parent[i]` est un entier qui représente la partie à laquelle appartient `i`.
- d'un tableau `rank` dans laquelle on maintient un ordre entre les parties : leur rang. Ce rang représente d'une certaine manière la taille de la partie. `rank[i]` est un entier qui vaut 0 à l'initialisation de la structure et qui croit au fur et à mesure des unions de parties.

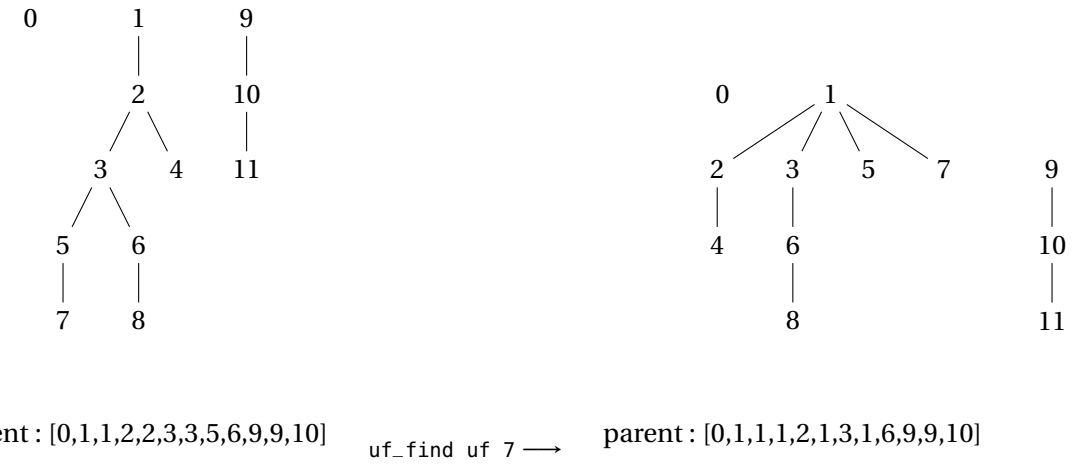


FIGURE 10.4 – Illustration de la compression de chemin lorsqu'on cherche le parent de l'élément 7. Cet ensemble est divisé en trois parties identifiées par les racines 0, 1 et 9.

■ **Exemple 67 — Exemple de tructure unir-trouver.** Par exemple, pour un ensemble à 5 éléments, `parent = [1,1,4,1,4]` signifie que les éléments d'indice 0,1 et 3 sont dans une même partie représentée par 1 et que 2 et 4 sont dans une même partie représentée par 4. `rang =[1,1,2,1,2]` signifie que le rang de la partie 1 est 1 et celui de la partie 4 est 2.

R Pour que la complexité soit optimale, il est important d'implémenter la compression de chemin dans les opérations de la structure unir-trouver. Ce concept est illustré sur la figure 10.4. Au fur que l'on interroge (trouver) la structure, on relie directement un élément à l'entier qui représente sa partie.

C Tri topologique d'un graphe orienté

a Ordre dans un graphe orienté acyclique

Dans un graphe orienté acyclique (Acyclic Directed Graph ou DAG en anglais), les arcs définissent un **ordre partiel**, le sommet à l'origine de l'arc pouvant être considéré comme le prédécesseur du sommet à l'extrémité de l'arc. Par exemple, sur la figure 10.5, a et b sont des prédécesseurs de d et e est un prédécesseur de g. Mais ces arcs ne disent rien de l'ordre entre e et h, l'ordre n'est pas total.

L'algorithme de tri topologique permet de créer un ordre total $\preceq$ sur un graphe orienté acyclique. Formulé mathématiquement :

$$\forall (v,u) \in V^2, (v,u) \in E \Longrightarrow v \preceq u \tag{10.1}$$

Sur l'exemple de la figure 10.5, plusieurs ordre topologiques sont possibles. Par exemple :

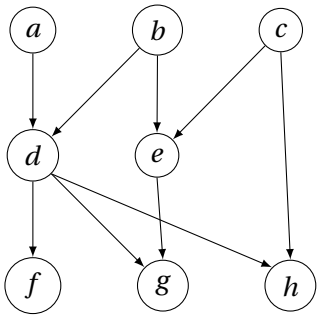


FIGURE 10.5 – Exemple de graphe orienté acyclique

- $a \preceq b \preceq c \preceq d \preceq e \preceq f \preceq g \preceq h$
- $a \preceq b \preceq d \preceq f \preceq c \preceq h \preceq e \preceq g$

b Existence d'un tri topologique

Lemme 2 Dans un graphe orienté acyclique, il existe au moins un sommet qui ne possède aucun arc entrant.

Démonstration. Par l'absurde.

Soit $G = (S, A)$ un graphe orienté acyclique. Supposons que tous les sommets du graphe possède un degré entrant non nul, c'est-à-dire $\forall s \in S, d_+(s) > 0$.

Soit s_0 un sommet de G. Comme $d_+(s_0) > 0$, il existe un sommet s_1 tel que $a = (s_1, s_0)$ est un arc du graphe. En remontant ainsi de suite, on obtient un chemin dans le graphe $(s_k, s_{k-1}, \ldots, s_1, s_0)$. Comme $d_+(s_k) > 0$ et que le graphe est acyclique, le sommet que l'on va découvrir s_{k+1} n'est pas un sommet du chemin. On découvre donc, en construisant ce chemin, une infinité de nouveaux sommets. Ce qui est absurde, car un graphe possède un nombre fini de sommet. ■

R Ce lemme est la clef de bien des démonstrations autour des graphes acyclique.

Théorème 16 — Existence d'un tri topologique. Tout graphe orienté acyclique possède un tri topologique.

Démonstration. Par récurrence sur le nombre de sommets du graphe.

- Initialisation : Pour un graphe orienté acyclique à un seul sommet, ce sommet est trivialement trié.
- Hérédité : on suppose que tout graphe orienté acyclique d'ordre strictement inférieur à n possède un tri topologique. Considérons un graphe G d'ordre n, acyclique et orienté. D'après le lemme 2 **il existe au moins un sommet s qui ne possède aucun arc entrant, sinon cela aurait pour effet de créer un cycle dans le graphe**. Si on retire s et ses arcs sortants du graphe, on obtient deux composantes dont un graphe orienté acyclique à $n-1$

sommets qui possède, par hypothèse de récurrence, un tri topologique. Comme s n'a pas de prédecesseurs dans le graphe, il suffit alors de le placer en tête de ce tri topologique pour obtenir un tri topologique de G.

- Conclusion : comme la propriété 16 est vraie pour un graphe à un sommet et que l'hérédité est démontrée, alors elle est vraie pour tout graphe acyclique orienté.

■

c Algorithmes de tri topologique

Pour construire un tri topologique, on peut utiliser l'algorithme de Kahn (cf. algorithme 10) ou le parcours en profondeur (cf. algorithme 11). Tous les deux permettent de détecter des cycles.

Algorithme 10 Algorithme de Kahn pour le tri topologique

```
1 : Fonction TT_KAHN(G)                          ▷ G = (S, A) est un graphe acyclique orienté
2 :     Créer une liste vide ordre
3 :     Créer une file file pour stocker les sommets dont le degré entrant vaut 0
4 :     Créer un dictionnaire degré_entrant pour stocker le degré entrant de chaque sommet
5 :     pour chaque sommet v ∈ S répéter
6 :         Calculer le degré entrant de v
7 :         degré_entrant[v] ← nombre d'arcs entrants pour v
8 :         si degré_entrant[v] = 0 alors
9 :             ENFILER(file, v)
10 :    tant que file n'est pas vide répéter
11 :        u ← DÉFILER(file)
12 :        AJOUTER(ordre,u)
13 :        pour pour chaque arc u → v répéter
14 :            Réduire le degré entrant de v de 1
15 :            si degré_entrant[v] = 0 alors
16 :                ENFILER(file, v)
17 :    si la taille de ordre vaut le cardinal de S alors
18 :        renvoyer ordre
19 :    sinon
20 :        Lever une exception : il y a un cycle dans le graphe et aucun un ordre topologique.
```

L'algorithme de tri topologique (cf. algorithme 11) utilise le parcours en profondeur d'un graphe pour marquer au fur et à mesure les sommets dans l'ordre topologique. Une pile est utilisée pour enregistrer l'ordre chronologique de découverte des sommets.

Au cours de l'algorithme, un sommet change d'état : il peut passer de «non visité» à «en cours» et finalement à «terminé». Il est alors possible de détecter un cycle si on redécouvre un sommet dans l'état «en cours». Des dates peuvent également être ajoutées au cours du traitement afin de pouvoir ordonnancer proprement les tâches parallélisables.

Algorithme 11 Tri topologique

```
1 : Fonction TRI_TOPOLOGIQUE(G)                ▷ G = (V, E) est un graphe orienté acyclique
2 :     Créer une liste vide ordre_topologique
3 :     Créer un ensemble visités pour les sommets visités
4 :     Créer un dictionnaire état pour l'état des sommets (non_visité, en_cours, terminé)
5 :     Fonction DFS(s)                        ▷ DFS : parcours en profondeur en anglais
6 :         si s n'est pas dans visités alors
7 :             Ajouter s à visités
8 :             état[s] ← en_cours
9 :             pour chaque arc s → v répéter
10 :                si état[v] = non_visité alors
11 :                    DFS(v)
12 :                sinon si état[v] = en_cours alors
13 :                    Lever une exception : cycle détecté
14 :            état[s] ← terminé
15 :            Ajouter s au début de ordre_topologique
16 :    pour chaque sommet v ∈ V répéter
17 :        si v n'est pas dans visités alors
18 :            DFS(v)
19 :    renvoyer ordre_topologique
```

D Composantes fortement connexes d'un graphe orienté et 2-SAT

■ **Définition 126 — Composante fortement connexe d'un graphe orienté** $G = (V, E)$**.** Une composante fortement connexe d'un graphe orienté G est un sous-ensemble S de ses sommets, maximal au sens de l'inclusion, tel que **pour tout couple** de sommets (s, t) de la composante, il existe un chemin de s à t dans G.

■ **Définition 127 — Fortement connexe, une relation binaire** $\mathcal{C}$**.** Soit u et v deux sommets de G et $\mathcal{R}$ la relation binaire définie sur les sommets d'un graphe orienté G par : $u\mathcal{R}v$ si et seulement si u et v font partie d'une même composante fortement connexe $\mathcal{C}$.

Théorème 17 — $\mathcal{R}$ est une relation d'équivalence.

Démonstration.
- Réflexivité : soit u un sommet de G. On a bien $u\mathcal{R}u$, car u est trivialement connecté à u.
- Symétrie : si $u\mathcal{R}v$ alors il existe un chemin de u à v, et v et u appartiennent à la même composante fortement connexe. D'après la définition d'une composante fortement connexe, on a donc $v\mathcal{R}u$.
- Transitive : si $u\mathcal{R}v$ et si $v\mathcal{R}w$, alors u et w appartiennent à la même composante fortement connexe que v. On a donc $w\mathcal{R}v$ et $v\mathcal{R}u$: il existe donc un chemin de w à u, c'est-à-dire $w\mathcal{R}u$.

$\mathcal{R}$ est donc une relation d'équivalence. ■

Théorème 18 — L'ensemble des composantes connexes d'un graphe orienté $G = (S, A)$ **forme une partition de** S**.**

Démonstration. Montrer :
- que les composantes fortement connexes sont disjointes par l'absurde,
- que chaque sommet appartient au moins à une composante, fût-elle réduit à un seul élément.

■

■ **Définition 128 — Graphe quotient.** Le graphe quotient d'un graphe orienté $G = (S, A)$ est le graphe $G_q = (S_q, A_q)$ où :
- S_q est l'ensemble des composantes fortement connexes de G, c'est-à-dire chaque sommet de G_q est une composante connexe de G.
- $A_q = \left\{(c_1, c_2) \in S_q^2, c_1 \neq c_2 \text{ et } \exists (u, v) \in c_1 \times c_2, (u, v) \in A\right\}$

Théorème 19 — Le graphe quotient est acyclique.

Démonstration. On procède par l'absurde. Supposons que G_q, le graphe quotient de G soit cyclique. Cela signifierait qu'il existerait un cycle dans G_q et donc on pourrait trouver un sommet c_i de S_q tel qu'il existerait un chemin dans G_q tel que $c_i \rightarrow^* c_i$. Autrement dit, il existerait un cycle qui relierait des composantes connexes de G. Mais alors, ce cycle serait lui-même une composante connexe de G et devrait donc être un sommet de G_q. On aboutit alors une contradiction : G_q ne serait pas le graphe quotient de G.

C'est pourquoi un graphe orienté est un graphe acyclique de ses composantes fortement connexes. ■

Le calcul des composantes connexes d'un graphe est par exemple utilisé pour résoudre le problème 2-SAT. Dans le cadre de ce problème, on dispose d'une formule logique sous la forme conjonctive normale et chaque clause comporte deux variables. L'idée est de construire un graphe à partir de la formule de cette formule F. Supposons qu'elle soit constituée de m clauses et n variables $(v_1, v_2, \ldots, v_n)$. On élabore alors un graphe $G = (V, E)$ à $2n$ sommets et $2m$ arêtes. Les sommets représentent les n variables v_i ainsi que leur négation $\neg v_i$. Les arêtes sont construites de la manière suivante : on transforme chaque clause de F de la forme $v_i \vee v_j$ en deux implications $\neg v_1 \Longrightarrow v_2$ ou $\neg v_2 \Longrightarrow v_1$. Cette transformation utilise le fait que la formule $a \Longrightarrow b$ est équivalent à $\neg a \vee b$.

Théorème 20 F n'est pas satisfaisable si et seulement s'il existe une composante fortement connexe contenant une variable v_i et sa négation $\neg v_i$.

Démonstration. ($\Longleftarrow$) S'il existe une composante fortement connexe contenant a et $\neg a$, alors cela signifie $F : (a \Longrightarrow \neg a) \wedge (\neg a \Longrightarrow a)$. Or cette formule n'est pas satisfaisable. En effet, si a est vrai alors $(a \Longrightarrow \neg a)$ est faux, car du vrai on ne peut pas conclure le faux d'après la définition sémantique de l'implication. De même, si a est faux alors $(\neg a \Longrightarrow a)$ est faux, pour la même raison. Dans tous les cas, la formule est fausse. F n'est pas staisfaisable.

($\Longrightarrow$) Par contraposée. Supposons qu'il n'existe pas de composante fortement connexe contenant a et $\neg a$. Cela peut se traduire en la formule $\neg F : \neg(a \Longrightarrow \neg a) \vee \neg(\neg a \Longrightarrow a)$. Or, cette formule F est toujours satisfaisable. En effet, $\neg F$ s'écrit

$$\neg(\neg a \vee \neg a) \vee \neg(a \vee a) = a \vee \neg a \tag{10.2}$$

ce qui est toujours vérifié. S'il n'existe pas de composante fortement connexe, alors F est satisfaisable. Par contraposition, si F n'est pas satisfaisable alors il existe une composante fortement connexe. ■

On peut montrer que SAT-2 est un problème de décision polynomial. C'est une restriction à des clauses de deux variables du problème général SAT qui lui est NP-complet.

E Graphes bipartis et couplage maximum

a Caractérisation des graphes bipartis

■ **Définition 129 — Graphe biparti.** un graphe $G = (S, A)$ est biparti si l'ensemble S de ses sommets peut être divisé en deux sous-ensembles disjoints U et W tels que chaque arête de A ait une extrémité dans U et l'autre dans W.

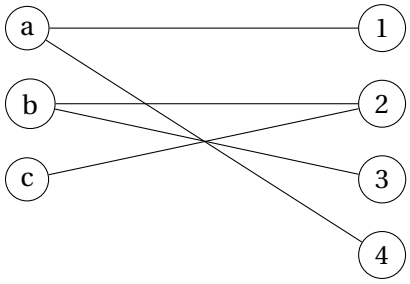


FIGURE 10.6 – Exemple de graphe biparti

Théorème 21 — Caractérisation des graphes bipartis par les cycles. Un graphe est biparti si et seulement s'il ne possède aucun cycle de longueur impaire,

Démonstration. ($\Longrightarrow$) Par l'absurde. Soit $G = (U, W, A)$ un graphe biparti. Soit $C = (s_0, s_1, s_2, \ldots, s_k, s_0)$ un cycle de longueur impaire de G. On suppose sans perte de généralité que s_0 est dans U. Alors, comme G est biparti, $s_i \in U$ si i est pair et $s_i \in W$ sinon. Comme C est un cycle impair, k est nécessairement pair. Donc s_k est dans U. Mais s_0 est dans U également, ce qui contredit le fait que G soit biparti. C'est donc absurde et il n'existe donc pas de cycle impair dans un graphe biparti.

($\Longleftarrow$) Soit un graphe $G = (S, A)$ ne possédant aucun cycle impair. Soit un T un arbre couvrant de racine r de G. On construit les ensembles U et W de la manière suivante :

- r appartient à U,
- Tout sommet séparé de r par un nombre pair d'arêtes appartient à U,
- Les autres sommets à W.

Les ensembles U et W sont disjoints par construction et recouvrent l'ensemble S de sommets de G. Il s'agit maintenant de montrer qu'aucune arête ne relie deux sommets de U ou de W en procédant par l'absurde. Supposons qu'il en existe une. Soit (a, b) une arête reliant deux sommets de U. Il existe un chemin de a à b dans l'arbre couvrant T et ce chemin possède un nombre pair d'arêtes. Si on complète ce chemin par un l'arête (b, a), on obtient un cycle de longueur impaire. Ce qui est absurde par hypothèse. Donc une telle arête n'existe pas et les ensemble U et W forment donc une bipartition de S.

■

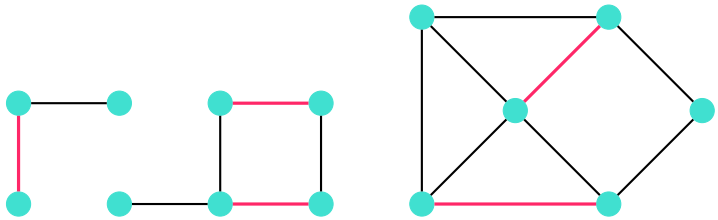


FIGURE 10.7 – Illustration de la notion de couplage maximal

Pour savoir si un graphe est biparti on pourrait donc rechercher les cycles et évaluer leurs longueurs. Il existe cependant une autre approche. Un graphe biparti est un graphe bicolorable comme le montre la figure 10.9, c'est-à-dire son nombre chromatique est égal à 2.

Théorème 22 — Caractérisation des graphes bipartis et coloration. Un graphe est biparti si et seulement si est bi-colorable.

b Couplage dans un graphe biparti

■ **Définition 130 — Couplage.** Un couplage Γ dans un graphe non orienté $G = (S, A)$ est un ensemble d'arêtes deux à deux non adjacentes. Formellement :

$$\forall (a_1, a_2) \in \Gamma^2, a_1 \neq a_2 \Longrightarrow a_1 \cap a_2 = \emptyset \tag{10.3}$$

c'est-à-dire que les sommets de a_1 et a_2 ne sont pas les mêmes.

■ **Définition 131 — Sommets couplés, sommets exposés.** Un sommet est couplé s'il fait parti d'une arête de Γ. Un sommet est exposé s'il ne fait pas parti des arêtes de Γ, c'est-à-dire il n'est pas couplé.

■ **Définition 132 — Couplage maximal.** Un couplage maximal est tel que si on lui ajoute une arête, ce n'est plus un couplage. Il est donc maximal au sens de l'inclusion.

■ **Définition 133 — Couplage de cardinal maximum.** Un couplage de cardinal maximum est un couplage contenant le plus grand nombre d'arêtes possible.

■ **Exemple 68 — Affectation des cadeaux sous le sapin .** Au pied du sapin de Noël, un papa a disposé six cadeaux dont les paquets sont tous différents et numérotés de 0 à 5 [a]. Il a décidé que les cadeaux seraient répartis en fonction des paquets que les enfants préfèrent.

Ses cinq enfants expriment donc leurs préférences. Le papa pourra-t-il affecter un cadeau à chaque enfant et faire en sorte que ce cadeau soit un de leurs préférés ?

Évidemment la réponse à cette question dépend des préférences émises par les enfants. Supposons qu'ils se soient exprimés ainsi :

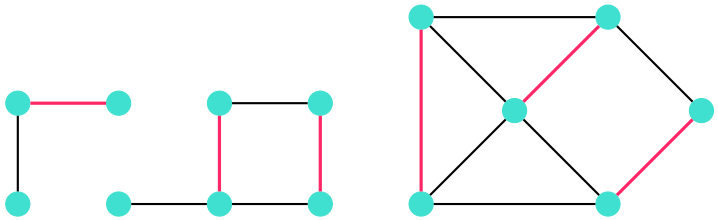


FIGURE 10.8 – Illustration de la notion de couplage de cardinal maximum

Alix 0,2

Brieuc 1,3,4,5

Céline 1,2

Dimitri 0,1,2

Enora 2

On peut représenter par un graphe biparti cette situation comme sur la figure 10.9. Dans ce cas précis, comme il y a quatre enfants qui ne veulent que les trois premiers cadeaux, il n'y a pas de solution. Mais si Enora avait choisit 4 et 5 ???

a. Ce papa est informaticien !

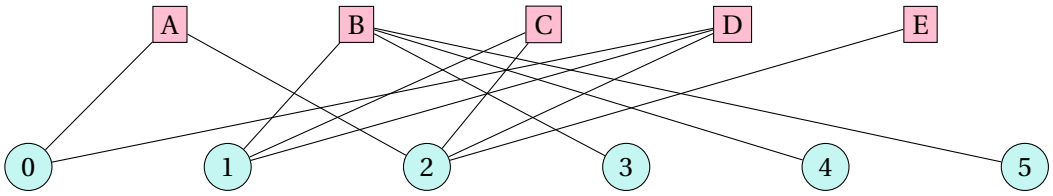


FIGURE 10.9 – Exemple de graphe biparti pour un problème d'affectation sans solution.

c Chemin augmentant

Pour résoudre le problème de trouver un couplage de cardinal maximum, on peut procéder en suivant l'algorithme 12. Il s'agit de construire un chemin augmentant pour atteindre un couplage de cardinal maximum.

■ **Définition 134 — Chemin alternant.** Une chemin alternant dans un graphe non orienté G et pour un couplage Γ est tel que les arêtes appartiennent successivement à Γ et $E \setminus \Gamma$.

■ **Définition 135 — Chemin augmentant.** Un chemin augmentant est un chemin alternant dont les extrémités sont des sommets exposés, c'est-à-dire qui n'appartiennent pas au couplage Γ.

Théorème 23 — Lemme de Berge. Soit un couplage Γ dans un graphe. Γ est de cardinal maximum si et seulement s'il ne possède aucun un chemin augmentant.

Démonstration. ($\Rightarrow$) Soit Γ une couplage de cardinal maximum. Si un chemin augmentant existait, on pourrait améliorer Γ en augmentant le nombre d'arêtes de ce couplage, ce qui contredit l'hypothèse que Γ est maximum.

($\Leftarrow$) Soit Γ une couplage ne possédant aucun un chemin augmentant. Si Γ n'est pas maximum, cela signifie qu'on peut ajouter une arête au couplage et le faire croitre. Mais cela signifie également qu'il existe un chemin augmentant. Ce qui est contredit notre hypothèse.

■

R Soit un couplage Γ dans un graphe. Soit π un chemin augmentant de Γ. Alors il existe un couplage Γ' qui possède plus d'arêtes que Γ. Ce couplage vaut :

$$\Gamma' = \Gamma \Delta \pi$$

où Δ dénote la différence symétrique de deux ensembles.

$$\Gamma \Delta \pi = \{e \in E, e \in \{\Gamma \setminus \pi \cup \pi \setminus \Gamma\}\}$$

L'algorithme de recherche d'un couplage de cardinal maximum 12 s'appuie sur le lemme de Berge 23. La stratégie est la suivante : à partir d'un couplage Γ, on construit un nouveau couplage de cardinal supérieur à l'aide d'un chemin augmentant comme le montre la figure 10.10.

Dans un graphe **biparti**, il est facile d'augmenter la taille d'un couplage jusqu'au cardinal maximum :

1. s'il existe deux sommets exposés reliés par une arête, il suffit d'ajouter cette arête au couplage. Puis, on appelle récursivement l'algorithme sur ce nouveau couplage.
2. sinon il faut trouver un chemin augmentant π dans le graphe.
   (a) s'il n'y en a pas, l'algorithme est terminé.
   (b) sinon on effectue la différence symétrique entre le couplage Γ et l'ensemble des arêtes du chemin augmentant π pour obtenir le nouveau couplage $\Gamma \Delta \pi$. Puis, on appelle récursivement l'algorithme avec ce nouveau couplage.

Pour trouver un chemin augmentant dans un graphe biparti $G = ((U, D), E)$, on cherche le plus court chemin entre deux sommets exposés dans un graphe orienté auxiliaire G_o construit de la manière suivante :

1. toutes les arêtes de E qui n'appartiennent pas au couplage Γ sont orientés de U vers D.
2. toutes les arêtes de Γ sont orientées de D vers U.

Le plus court chemin entre deux sommets exposés de G_o est nécessairement un chemin augmentant, son caractère alternant vient du fait que le graphe est biparti.

La figure 10.11 illustre les différentes étapes de l'algorithme. On vérifie bien que le résultat est correct : chaque enfant aura bien un cadeau qu'il avait mis sur sa liste.

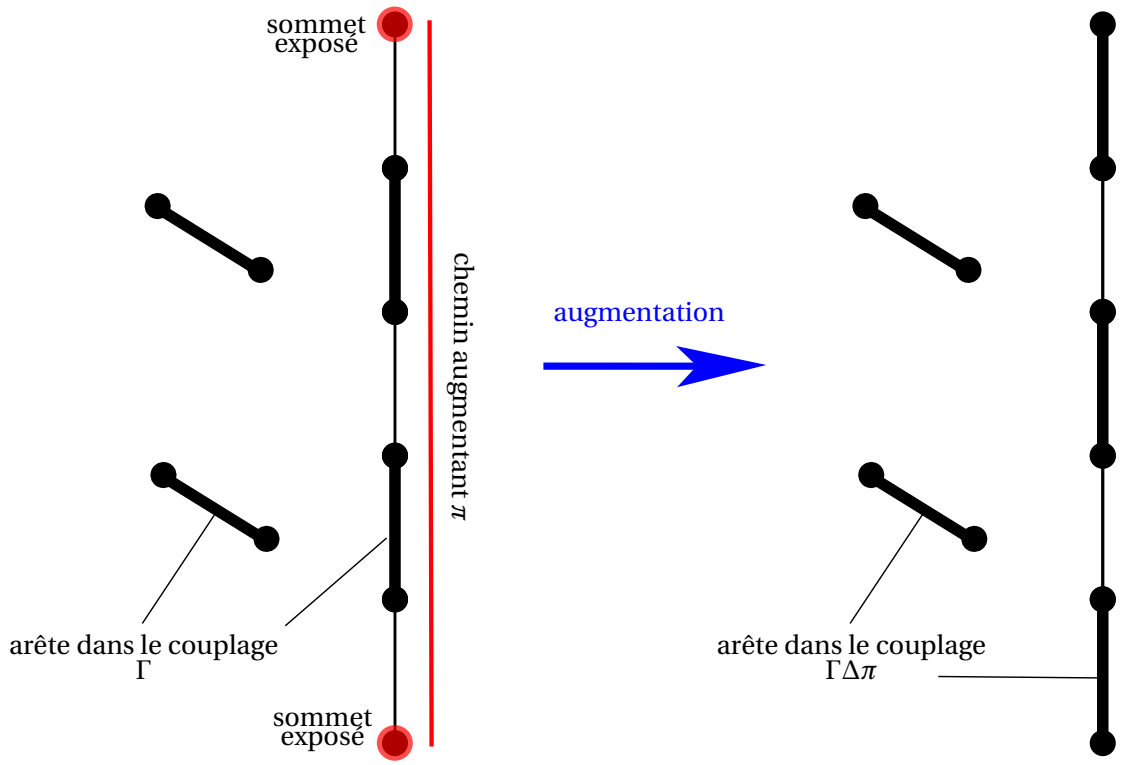


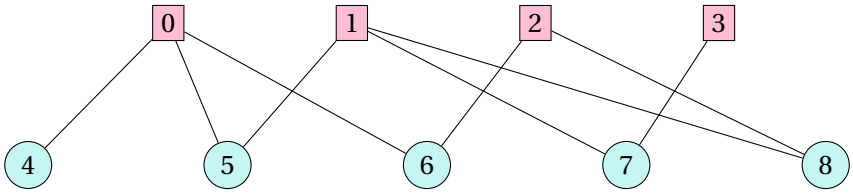
FIGURE 10.10 – Illustration de la construction d'un couplage de cardinal supérieur grâce à un chemin augmentant. (Source originale : Wikimedia Commons)

Algorithme 12 Recherche d'un couplage de cardinal maximum

Entrée : un graphe biparti $G = ((U, D), E)$
Entrée : un couplage Γ initialement vide
Entrée : F_U, l'ensemble de sommets exposés de U initialement U
Entrée : F_D, l'ensemble de sommets exposés de D initialement D

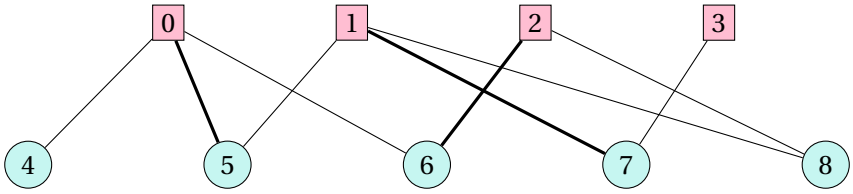
```
1 : Fonction C_MAX(G = (V = (U, D), E), Γ, F_U, F_D))        ▷ Γ est le couplage, vide initialement
2 :     si une arête e = (u, v) entre un sommet de F_U et un sommet de F_D existe alors
3 :         C_MAX(G, Γ ∪ {e}, F_U \ {u}, F_D \ {v})
4 :     sinon
5 :         Créer le graphe orienté G_o                  ▷ ∀e ∈ E, e de U vers D si e ∉ Γ, l'inverse sinon
6 :         Calculer le plus court chemin π entre un sommet de F_U et un de F_D dans G_o
7 :         si un tel chemin π n'existe pas alors
8 :             renvoyer Γ
9 :         sinon
10 :            C_MAX(G, ΓΔπ, F_U \ {π_start}, F_D \ {π_end})
11 :                                  ▷ π_start et π_end : début et fin du chemin π
```

Le graphe de départ de l'algorithme est le suivant :



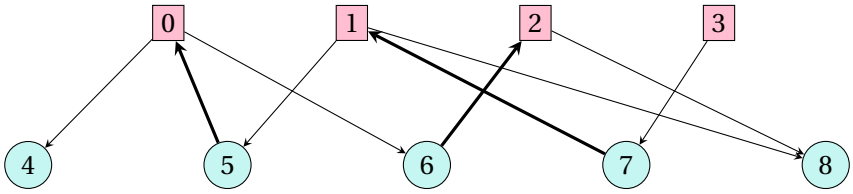
$\Gamma = \emptyset$

On effectue **trois appels récursifs** et, à chaque fois, on a trouvé une arête dont les sommets sont tous les deux exposés.



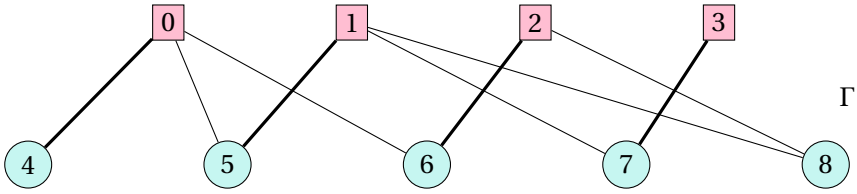
$\Gamma = \{(0,5),(1,7),(2,6)\}$

À ce stade de l'algorithme, aucun sommet exposé n'est relié par une arête à un autre sommet exposé. Donc, on construit le graphe G_o d'après le couplage Γ. On trouve le chemin le plus court entre les deux premiers sommets exposés 3 et 4 : π.



$\Gamma = \{(0,5),(1,7),(2,6)\}$
$\pi = \{3,7,1,5,0,4\}$

On en déduit un nouveau couplage $\Gamma = \Gamma \Delta \pi$:



$\Gamma = \{(0,4),(1,5),(3,7),(2,6)\}$

FIGURE 10.11 – Étapes de l'algorithme de recherche d'un couplage de cardinal maximum

F Pour aller plus loin --→ HORS PROGRAMME

a Affectation de ressources

Si les arêtes du graphe sont valuées (graphe pondéré), alors la recherche d'un couplage de cardinal maximum et de poids minimum dans un graphe biparti est en fait le problème de l'**affectation des ressources**. Le problème peut être énoncé de la manière suivante :

■ **Définition 136 — Problème d'affectation de ressources.** Soit un ensemble de personnes P et un ensemble de tâches T. Soit le graphe biparti $G = ((P,T),A)$ et une fonction de pondération sur les arêtes $w : A \to \mathbb{R}$. Le problème d'affectation consiste à trouver un couplage $\Gamma \subseteq A$ tel que :

- $|\Gamma| = |T|$,
- et la somme $\sum_{a\in\Gamma} w(a)$ est minimale.

L'algorithme hongrois résout ce problème en $O(n^4)$.

b Mariages stables ou appariement de ressources

Si les sommets du graphe biparti expriment des vœux de préférences, alors le problème considéré est celui des mariages stables. Ceci n'est pas au programme mais pourrait intéresser des rédacteurs d'épreuves de concours. Si vous avez du temps libre, les algorithmes d'acceptation différée [3] et des cycles d'échanges optimaux [9] sont à considérer.

La difficulté réside dans la capacité relative des algorithmes d'appariement de ressources de satisfaire les critères suivants :

Efficacité c'est-à-dire la satisfaction des préférences exprimées des candidats. Il s'agit de chercher à améliorer la satisfaction d'un candidat sans diminuer celle d'un autre.

Équité c'est-à-dire le respect des priorités émises par les institutions pour chaque candidat. Il n'est pas souhaitable en effet qu'un candidat plus prioritaire et désirant intégrer cette institution se voit refuser l'entrée alors qu'un autre candidat moins prioritaire y est admis.

Non manipulabilité c'est-à-dire la meilleure stratégie pour un candidat consiste toujours à classer ses vœux dans l'ordre réel de ses préférences, quels que soient les vœux soumis par les autres candidats. Il s'agit d'empêcher les stratégie individuelles de positionnement. Certains candidats pourraient être tentés par exemple de ne postuler qu'à certaines institutions pour lesquelles ils estiment avoir une chance. Il est possible aussi que les institutions manipulent l'algorithme.

Aucun algorithme n'est optimal sur ces trois critères, mais certains algorithmes comme [3] et [9] réalisent des compromis optimaux sur deux des trois critères et font au mieux sur le troisième. Le premier est non manipulable et respecte les priorités des candidats, mais ne produit pas nécessairement un appariement efficace. Le second algorithme est également non manipulable mais produit un appariement efficace au prix d'une possible violation des priorités des candidats.

Sixième partie

Langages et automates

11

INTRODUCTION AUX LANGAGES

À la fin de ce chapitre, je sais :

- ☞ définir les concepts d'alphabet, de mot, de mot vide et de langage
- ☞ expliquer les concepts de suffixe, de préfixe, de facteur et de sous-mot
- ☞ expliquer le résultat du lemme de Levi

L'informatique est la construction de l'information par le calcul. Force est de constater que le seul outil conceptuel, universel et pratique pour construire et manipuler l'information est le langage : un langage est un moyen de communiquer, stocker et transformer de l'information. Le calcul de l'information par un ordinateur au moyen d'un ou plusieurs langages peut créer un sens ou pas, tout comme l'interprétation d'un texte par un humain.

C'est pourquoi la théorie des langages est un des principaux fondements de l'informatique. Qui dit langage dit alphabet, mots, préfixes, suffixes mais aussi ensembles de mots, agrégation de mots... Comment définir clairement ces concepts afin de pouvoir les calculer ? C'est la question qui guide ce chapitre.

A Alphabets

■ **Définition 137 — Ensemble.** Un ensemble est une collection de concepts qu'on appelle éléments. L'ensemble vide est noté $\emptyset$.

■ **Définition 138 — Cardinal d'un ensemble fini.** Le cardinal d'un ensemble fini E est son nombre d'éléments. On le note $|E|$.

■ **Définition 139 — Alphabet.** Un alphabet est un ensemble fini de lettres (ou symboles) non vide.

■ **Définition 140 — Longueur d'un alphabet.** La longueur d'un alphabet est le nombre de lettres de celui-ci, c'est-à-dire $|\Sigma|$.

■ **Exemple 69 — Alphabet latin commun.** L'alphabet latin commun

$$\Sigma = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, X, Y, Z\}$$

a une longueur de 26.

■ **Exemple 70 — Héxadécimal.** L'alphabet héxadécimal

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

a une longueur de 16.

■ **Exemple 71 — ASCII.** L'alphabet ASCII est constitué des nombres entiers de 0 à 127 et représente les caractères nécessaire à l'écriture de l'américain, y compris les caractères de contrôle nécessaires à la pagination. Il possède une longueur de 128.

B Mots

Un mot d'un alphabet Σ est une séquence de lettres de Σ. Un mot peut être une séquence vide notée ϵ, car $\emptyset \subset \Sigma$ par définition d'un ensemble. On peut donner plusieurs définition d'un mot.

■ **Définition 141 — Mot (comme application).** Un mot de longueur $n \in \mathbb{N}^\star$ est une application de $[\![1, n]\!] \longrightarrow \Sigma$. À chaque position dans le mot correspond une lettre de l'alphabet.

■ **Définition 142 — Mot vide ϵ.** Le mot vide ϵ est l'application de l'ensemble vide dans Σ.

R Le mot vide ϵ est un mot ne comportant aucun symbole. Dans le contexte des mots, le mot vide est l'élément neutre de la concaténation de mots. On peut comparer son rôle au 1 pour la multiplication des entiers naturels $\mathbb{N}$.

■ **Définition 143 — Longueur d'un mot.** La longueur d'un mot w est le nombre de lettres qui composent sa séquence. On note souvent cette longueur $|\ w\ |$.

■ **Définition 144 — Ensemble de mots possibles.** On note Σ^* l'ensemble des mots possibles crées à partir d'un alphabet Σ.

■ **Définition 145 — Ensemble de mots de longueur n.** On note Σ^n l'ensemble de tous les mots de longueur n crées à partir d'un alphabet Σ.

■ **Définition 146 — Égalité de deux mots.** Soit un alphabet Σ. Soient u et v deux mots de Σ^*. On dit que u est égal à v et on note $u = v$ si et seulement si u et v ont même longueur et que leurs lettres correspondent.

Formellement :

$$u = v \iff |u| = |v| \text{ et } \forall i \in \mathbb{N}, i < |u|, u_i = v_i \tag{11.1}$$

où les u_i désignent les lettres de u et les v_i les lettres de v.

R Cette définition est équivalente à dire que les fonctions u et v sont égales : elles ont même ensemble de définition et sont identiques en tout point.

■ **Définition 147 — Concaténation de mots.** Soit $v, w \in \Sigma^*$. On appelle concaténation de v et w l'opération $\circ$ notée $vw = v \circ w$ qui est obtenue par agrégation du mot w à la suite du mot v.

R Dans les notations suivantes, on omettra d'écrire le symbole $\circ$ entre les éléments, la concaténation étant juste une agrégation de symboles.

Théorème 24 — La concaténation est une loi de composition interne sur un ensemble Σ^* et ϵ en est l'élément neutre. .

Démonstration. Soit $v, w \in \Sigma^*$. On observe que $vw \in \Sigma^*$. Donc c'est une application de $\Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. De plus, on peut observer que $v\epsilon = \epsilon v = v$. Donc ϵ est l'élément neutre de cette loi. ■

■ **Définition 148 — Monoïde.** Un ensemble E muni d'une loi de composition interne associative et d'un élément neutre e est nommée monoïde $(E, \star)$.

R On peut facilement montrer que la concaténation de mots est une loi de composition interne associative. C'est pourquoi, $(\Sigma^*, \circ)$ est un **monoïde**.

Il faut bien remarquer cependant qu'il n'existe pas a priori de concept d'inverse dans un monoïde, c'est-à-dire il **n'existe pas** de mots v et w tels que $vw = \epsilon$.

Toutefois, on peut simplifier par la gauche ou la droite :

$$vw = vx \Longrightarrow w = x \tag{11.2}$$

$$wv = xv \Longrightarrow w = x \tag{11.3}$$

R La longueur d'un mot est un morphisme de monoïde car $|vw| = |v| + |w|$.

■ **Définition 149 — Ensemble de mots non vides.** On note Σ^+ l'ensemble des mots non

vides crées à partir d'un alphabet Σ. C'est le plus petit ensemble tel que :

$$\forall a \in \Sigma, a \in \Sigma^+ \tag{11.4}$$
$$\forall w \in \Sigma^+, \forall a \in \Sigma, wa \in \Sigma^+ \tag{11.5}$$

On a $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.

C Mots définis inductivement

■ **Définition 150 — Mot (inductivement par la droite).** Soit Σ un alphabet. Alors on définit un mot de manière inductive par la droite ainsi :

Base ϵ est un mot sur Σ,

Règle de construction si w est un mot sur Σ et a une lettre de Σ, alors $w.a$ est un mot sur Σ.

où l'opération . est l'ajout d'une lettre à droite à un mot.

■ **Définition 151 — Mot (inductivement par la gauche)).** Soit Σ un alphabet. Alors on définit un mot de manière inductive par la droite ainsi :

Base ϵ est un mot sur Σ,

Règle de construction si w est un mot sur Σ et a une lettre de Σ, alors $a.w$ est un mot sur Σ.

où l'opération . est l'ajout d'une lettre à gauche à un mot.

■ **Définition 152 — Concaténation de mots (définie inductivement sur la première opérande).** On définit l'opérateur concaténation de mots $\circ$ par

$$\forall w \in \Sigma^*, \epsilon \circ w = w \text{ (Base)} \tag{11.6}$$
$$\forall v, w \in \Sigma^*, \forall a \in \Sigma, (a.v) \circ w = a.(v \circ w) \text{ (Règle de construction)} \tag{11.7}$$

où l'opération . est l'ajout d'une lettre à gauche à un mot.

■ **Définition 153 — Concaténation de mots (définie inductivement sur la seconde opérande).** On définit l'opérateur concaténation de mots $\circ$ par

$$\forall w \in \Sigma^*, w \circ \epsilon = w \tag{11.8}$$
$$\forall v, w \in \Sigma^*, \forall a \in \Sigma, v \circ (w.a) = (v \circ w).a \tag{11.9}$$

où l'opération . est l'ajout d'une lettre à droite à un mot.

Théorème 25 — ϵ est l'élément neutre de la concaténation.

Démonstration. on procède par induction sur la première opérande.

- Cas de base : pour $w = \epsilon$, on a $\epsilon \circ \epsilon = \epsilon$.
- Pas d'induction : soit $w \in \Sigma^*$. On suppose que ϵ est l'élément neutre pour ce mot : $w \circ \epsilon = \epsilon \circ w = w$. Considérons maintenant un élément a de l'alphabet Σ pour créer un mot plus long à partir de w. Par construction on a :

  $$(a.w) \circ \epsilon = a.(w \circ \epsilon)$$

  En utilisant l'hypothèse d'induction, on en déduit que $(a.w) \circ \epsilon = a.w$.
  ϵ est donc toujours l'élément neutre. On procède de même avec la définition sur la deuxième opérande.

■

R Une conséquence de ces définitions est qu'on peut confondre les opérateurs . et $\circ$ dans les notations. C'est ce qui est fait dans la suite de ce cours. On omettra également souvent l'opérateur lorsqu'il n'y a pas d'ambiguïtés.

■ **Définition 154 — Puissances d'un mot.** Les puissances d'un mot sont définies inductivement. Soit $w \in \Sigma$

$$(B) \quad w^0 = \epsilon \tag{11.10}$$

$$(I) \quad w^n = w w^{n-1} \text{ pour } n \in \mathbb{N}^\star \tag{11.11}$$

D Langages

■ **Définition 155 — Langage.** Un langage sur un alphabet Σ est un ensemble de mots sur Σ.

R Un langage peut être vide, on le note alors $\mathcal{L} = \emptyset$, son cardinal est nul. C'est l'élément neutre de l'union des langages et l'élément absorbant de la concaténation de langages [a]. Il ne faut pas confondre ce langage vide avec le langage qui ne contient que le mot vide $\mathcal{L} = \{\epsilon\}$ dont le cardinal vaut un et qui est l'élément neutre de la concaténation des langages.

a. comme le zéro pour l'addition et la multiplication des entiers

R Σ^*, l'ensemble de tous les mots sur Σ, est également appelé langage universel.

■ **Exemple 72 — Langages courants et concrets.** Voici quelques exemples de langages concrets utilisés couramment :

- le langage des dates : une expression est-elle une date ? Par exemple, les dates 21/11/1943 et 11/21/43 sont-elles admissibles ?
- le langage des emails : utilisé pour détecter la conformité ou les erreurs dans les adresses

emails,

- les protocoles réseaux : par exemple le protocole DHCP.

■ **Exemple 73 — Langage des mots de longueur paire.** Soit l'ensemble E de mots sur l'alphabet Σ de longueur paire. On peut définir ce langage en compréhension comme suit :

$$E = \{w \in \Sigma^*, |w| = 0 \bmod 2\}$$

■ **Exemple 74 — Langage des puissances n d'un alphabet.** Soit l'ensemble E de mots sur l'alphabet $\Sigma = \{a, b\}$ qui comportent autant de a que de b. On peut définir ce langage en compréhension comme suit :

$$E = \{w \in \Sigma^*, \exists n \in \mathbb{N}, w \text{ est une permutation de } (a^n b^n)\}$$

Un langage est un ensemble. On peut donc définir les opérations ensemblistes sur les langages.

Soit deux langages $\mathcal{L}_1$ sur Σ_1 et $\mathcal{L}_2$ sur Σ_2.

■ **Définition 156 — Union de deux langages.** L'union de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur $\Sigma_1 \cup \Sigma_2$ contenant tous les mots de $\mathcal{L}_1$ et de $\mathcal{L}_2$.

$$\mathcal{L}_1 \cup \mathcal{L}_2 = \{w, w \in \mathcal{L}_1 \text{ ou } w \in \mathcal{L}_2\} \tag{11.12}$$

■ **Définition 157 — Intersection de deux langages.** L'intersection de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur $\Sigma_1 \cap \Sigma_2$ contenant tous les mots à la fois présents dans $\mathcal{L}_1$ et dans $\mathcal{L}_2$.

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \{w, w \in \mathcal{L}_1 \text{ et } w \in \mathcal{L}_2\} \tag{11.13}$$

■ **Définition 158 — Complémentaire d'un langage.** Le complémentaire d'un langage $\mathcal{L}$ est le langage défini sur Σ qui contient tous les mots non qui ne sont pas dans $\mathcal{L}$.

$$C(\mathcal{L}) = \overline{\mathcal{L}} = \{w, w \in \Sigma^* \text{ et } w \notin \mathcal{L}\} \tag{11.14}$$

■ **Définition 159 — Différence de deux langages .** La différence de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur Σ_1 contenant tous les mots présents dans $\mathcal{L}_1$ qui ne sont pas dans $\mathcal{L}_2$.

$$\mathcal{L}_1 \setminus \mathcal{L}_2 = \{w, w \in \mathcal{L}_1 \text{ et } w \notin \mathcal{L}_2\} \tag{11.15}$$

■ **Définition 160 — Produit de deux langages ou concaténation .** Le produit de $\mathcal{L}_1$ et $\mathcal{L}_2$ est le langage défini sur $\Sigma_1 \cup \Sigma_2$ contenant tous les mots formés par une mot de $\mathcal{L}_1$ suivi d'un mot de $\mathcal{L}_2$.

$$\mathcal{L}_1.\mathcal{L}_2 = \{vw, v \in \mathcal{L}_1 \text{ et } w \in \mathcal{L}_2\} \tag{11.16}$$

■ **Définition 161 — Puissances d'un langage.** Les puissances d'un langage $\mathcal{L}$ sont définies par induction :

$$\mathcal{L}^0 = \{\epsilon\} \tag{11.17}$$

$$\mathcal{L}^n = \mathcal{L}.\mathcal{L}^{n-1} \text{ pour } n \in \mathbb{N}^\star \tag{11.18}$$

■ **Définition 162 — Fermeture de Kleene d'un langage.** La fermeture de Kleene d'un langage $\mathcal{L}$ ou étoile de Kleene notée $\mathcal{L}^*$ est l'ensemble des mots formés par un nombre fini de concaténation de mots de $\mathcal{L}$. Formellement :

$$\mathcal{L}^* = \bigcup_{n \geqslant 0} \mathcal{L}^n \tag{11.19}$$

La fermeture d'un langage peut également être définie inductivement par :

$$\epsilon \in \mathcal{L}^* \tag{11.20}$$

$$v \in \mathcal{L}, w \in \mathcal{L}^* \Longrightarrow vw \in \mathcal{L}^* \tag{11.21}$$

$$v \in \mathcal{L}^*, w \in \mathcal{L} \Longrightarrow vw \in \mathcal{L}^* \tag{11.22}$$

R Il existe un nombre dénombrable de mots sur un alphabet Σ, c'est-à-dire qu'on peut les mettre en bijection avec $\mathbb{N}$, il y en a une infinité mais on peut les compter. Néanmoins, le nombre de langages sur Σ n'est pas dénombrable puisqu'il s'agit des parties d'un ensemble dénombrable.

E Préfixes, suffixes, facteurs et sous-mots

■ **Définition 163 — Préfixe.** Soit v et w deux mots sur Σ. v est un préfixe de w et on le note $v \preceq w$ si et seulement s'il existe un mot u sur Σ tel que :

$$vu = w \tag{11.23}$$

■ **Définition 164 — Suffixe.** Soit v et w deux mots sur Σ. w est un suffixe de v si et seulement s'il existe un mot u sur Σ tel que :

$$uw = v \tag{11.24}$$

■ **Définition 165 — Facteur.** Soit v et w deux mots sur Σ. v est un facteur de w si et seulement s'il existe deux mots t et u sur Σ tel que :

$$tvu = w \tag{11.25}$$

Vocabulary 7 — Subword ⇜⇝ Facteur. Attention l'imbroglio n'est pas loin…

■ **Définition 166 — Sous-mot.** Soit $w = a_1 a_2 \dots a_n$ un mot sur $\Sigma = \{a_1, a_2, \dots, a_n\}$ de longueur n. Alors $v = a_{\psi(1)} a_{\psi(2)} \dots a_{\psi(p)}$ est un sous-mot de w de longueur p si et seulement s'il $\psi : [\![1, p]\!] \longrightarrow [\![1, n]\!]$ est une application strictement croissante.

R Cette définition implique que l'ordre d'apparition des lettres dans un sous-mot est préservé par rapport à l'ordre de lettres du mot.

Vocabulary 8 — Scattered Subword ⇜⇝ Sous-mot...

■ **Exemple 75 — Illustrations des concepts précédents.** Prenons par exemple le mot le plus long de la langue française, *anticonstitutionnellement*. Alors

- *anti* est un préfixe, tout comme antico mais uniquement pour les informaticiens, pas les linguistes...
- *ment* est un suffixe,
- *constitution* est un facteur,
- *colle* est un sous-mot.

Théorème 26 — Relations d'ordre partiel. Les relations «être préfixe de», «être suffixe de» et «être facteur de » sont des relations d'ordre partiel.

Démonstration. Il suffit de montrer que ces relations sont réflexives, transitives et antisymétriques. C'est un exercice à faire. ■

R L'ordre est partiel car certains mots n'ayant aucune lettre en commun ne sont pas comparables, c'est-à-dire un mot peut n'être ni préfixe, ni suffixe ni facteur d'un autre mot.

■ **Définition 167 — Ordre lexicographique.** Soit u et v deux mots sur un alphabet Σ sur lequel on dispose d'un ordre total $\prec$. Alors on peut définir [a] l'ordre lexicographique $<_{\text{lex}}$ entre deux mots $u <_{\text{lex}} v$ par :

- u est un préfixe de v
- ou bien $\exists w, s, t \in \Sigma^*, \exists a, b \in \Sigma, a \prec b$ et $u =$ was et $v = wbt$

a. Il était temps après 18 ans d'école!

■ **Définition 168 — Ordre militaire (ou hiérarchique) sur les mots.** Soit Σ un alphabet muni d'un ordre total $\prec$. On définit l'ordre militaire sur les mots $<_{\text{mil}}$ par

$$\forall u, v \in \Sigma^*, u <_{\text{mil}} v \Leftrightarrow \begin{cases} |u| < |v| \\ \text{ou} \\ |u| = |v| \text{ et } u <_{\text{lex}} v \end{cases}$$

R Sur les mots, l'ordre militaire est bien fondé, mais pas l'ordre lexicographique. Cete remarque permet de définir des fonctions par induction structurelle aux mots définis inductivement, car l'ordre militaire fait des mots un ensemble bien ordonné.

■ **Exemple 76 — Suite infinie de mots.** Soit l'alphabet $\Sigma = a, b$ et la suite de mots $(w_n)_{n\in\mathbb{N}}$ définie par $w_n = a^n b$. Si on considère l'ordre lexicographique, cette suite est infinie et strictement décroissante. C'est pourquoi l'ordre lexicographique sur les mots n'est **pas** bien fondé.

■ **Définition 169 — Distance entre deux mots.** Supposons que l'on dispose d'une fonction λ capable de calculer le plus long préfixe, le plus long suffixe ou le plus long facteur commun entre deux mots. Alors on peut définir une distance entre deux mots v et w par :

$$d(v,w) = |vw| - 2\lambda(v,w) \tag{11.26}$$

■ **Définition 170 — Fermeture d'un langage par préfixe.** La fermeture par préfixe d'un langage $\mathcal{L}$ notée Pref($\mathcal{L}$) est le langage formé par l'ensemble des préfixes des mots de $\mathcal{L}$.

$$\text{Pref}(\mathcal{L}) = \left\{w \in \Sigma^*, \exists v \in \Sigma^*, wv \in \mathcal{L}\right\} \tag{11.27}$$

■ **Définition 171 — Fermeture d'un langage par suffixe.** La fermeture par suffixe d'un langage $\mathcal{L}$ notée Suff($\mathcal{L}$) est le langage formé par l'ensemble des suffixes des mots de $\mathcal{L}$.

$$\text{Suff}(\mathcal{L}) = \left\{w \in \Sigma^*, \exists v \in \Sigma^*, vw \in \mathcal{L}\right\} \tag{11.28}$$

■ **Définition 172 — Fermeture d'un langage par facteur.** La fermeture par facteur d'un langage $\mathcal{L}$ notée Fact($\mathcal{L}$) est le langage formé par l'ensemble des facteurs des mots de $\mathcal{L}$.

$$\text{Fact}(\mathcal{L}) = \left\{w \in \Sigma^*, \exists u, v \in \Sigma^*, uwv \in \mathcal{L}\right\} \tag{11.29}$$

F Propriétés fondamentales

On peut dors et déjà observer plusieurs faits :

- Soit v et w deux mots de Σ^*. A priori $vw \neq wv$, la concaténation n'est pas commutative.
- La décomposition d'un mot de Σ^* est unique en éléments de Σ. Ceci est dû au fait qu'il n'y a pas d'inverse dans un monoïde. On peut le démontrer en raisonnant par l'absurde, en supposant qu'il existe deux décompositions différentes d'un même mot. Comme les lettres ne peuvent pas disparaître par inversion et qu'elles sont atomiques, c'est-à-dire non décomposables, on aboutit à une contradiction.

Ces deux observations engendrent de nombreux développements dans la théorie des langages.

Théorème 27 — Lemme de Levi. Soient t,u,v et w quatre mots de Σ^*. Si $tu = vw$ alors il existe un unique mot $z \in \Sigma^*$ tel que :

- soit $t = vz$ et $zu = w$,
- soit $v = tz$ et $zw = u$.

Démonstration. Supposons que $|t| \geqslant |v|$. Alors v est un préfixe de t et il existe un mot z tel que $t = vz$. Or, on a $tu = vw = vzu$. Par simplification à gauche, on obtient $w = zu$. On procède de même pour la seconde égalité. ■

Le lemme de Levi est illustré sur la figure 11.1.

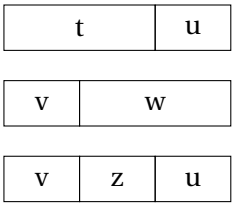


FIGURE 11.1 – Illustration du lemme de Levi

12

EXPRESSIONS RÉGULIÈRES

À la fin de ce chapitre, je sais :

- ☞ expliquer les définitions inductives des expressions régulières et des langages réguliers
- ☞ utiliser la sémantique des langages réguliers
- ☞ utiliser les identités remarquables sur les expressions régulières pour simplifier une expression
- ☞ construire un arbre représentant une expressions régulière

On peut décrire un langage de différentes manières :

- par compréhension : $\mathcal{L} = \{u \in \Sigma^*, |u| = 0 \bmod 2\}$, c'est-à-dire en utilisant une propriété spécifique au langage. Néanmoins, ceci n'est pas toujours évident et, de plus, cela n'implique pas de méthode concrète pour construire des mots de ce langage.
- pour les langages réguliers, par une expression régulière ou une grammaire régulière.

Les expressions régulières sont un moyen de caractériser de manière inductive certains langages et offre des **règles pour construire les mots** de ces langages. Tout comme en couture, on dit qu'elles constituent un **patron de conception**. À chaque expression régulière est associé un langage, c'est-à-dire l'ensemble des mots qu'elle permet d'élaborer. L'avantage principal des expressions régulières est qu'elles fournissent en plus une vision algébrique des langages réguliers et permettent donc le calcul.

R Régulier ou rationnel? Ces deux adjectifs sont employés de manière équivalente en français dans le cadre de la théorie des langages. Il existe des arguments en faveur de l'utilisation de chacun :

- rationnel : c'est le mot historique utilisé en France. Sa racine latine évoque le calcul et il s'agit donc des langages que l'on peut calculer.

- régulier : c'est un anglicisme mais dont la racine latine [a] évoque la conformation à une norme ou un règle. Il s'agit donc des langages que l'on peut décrire par une règle.

Ces deux adjectifs sont donc cohérents et utilisables en français pour décrire les langages et les expressions qui font l'objet de ce chapitre. L'un est plus pragmatique que l'autre !

a. via les anglo-normands et Guillaume le conquérant

A Définition des expressions régulières

■ **Définition 173 — Syntaxe des expressions régulières.** L'ensemble des expressions régulières $\mathcal{E}_R$ sur un alphabet Σ est défini inductivement par :

(Base) $\{\emptyset, \epsilon\} \cup \Sigma \in \mathcal{E}_R$,

(Règle de construction (union)) $\forall e_1, e_2 \in \mathcal{E}_R, e_1 \mid e_2 \in \mathcal{E}_R$

(Règle de construction (concaténation)) $\forall e_1, e_2 \in \mathcal{E}_R, e_1 e_2 \in \mathcal{E}_R$,

(Règle de construction (fermeture de Kleene)) $\forall e \in \mathcal{E}_R, e^* \in \mathcal{E}_R$.

R Cette définition peut s'exprimer ainsi : les expressions régulières sont constituées à la base de l'ensemble vide, du mot vide et des lettres de l'alphabet. On peut construire d'autres expressions régulières à partir de ces éléments de base en appliquant un nombre fini de fois les opérateurs d'union, de concaténation et de fermeture de Kleene.

R Le symbole de la concaténation est omis pour plus de lisibilité.

R L'utilisation des parenthèses est possible et souhaitable pour réduire les ambiguïtés. La priorité des opérateurs est l'étoile de Kleene, la concaténation puis l'union. Par défaut, l'associativité des opérateurs est choisie à gauche.

Par exemple, pour une alphabet $\{a, b, c\}$, on peut écrire :

- $a \mid b \mid c$ à la place de $a \mid (b \mid c)$
- $a \mid cb^*$ à la place de $a \mid (c(b^*))$

■ **Exemple 77 — Quelques expressions régulières.** Voici quelques expressions régulières pratiques dans la vie de tous jours :

$(P|MP|PC)SI$ sur l'alphabet latin désigne l'ensemble $\{PSI, MPSI, PCSI\}$

$(19|20)00$ sur l'alphabet $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$ désigne l'année 1900 ou l'année 2000.

1010101^* sur l'alphabet $\Sigma = \{0,1\}$ désigne l'ensemble des mots binaires préfixés par 101010 et se terminant éventuellement par des 1.

$\Sigma^* k \Sigma^+ q \Sigma^*$ il y a un mot de sept lettres engendré par cette expression régulière en français. Comme quoi, il y a toujours de l'espoir au Scrabble.

■ **Exemple 78 — D'autres expressions régulières.** Ces exemples sont typiques des expressions utilisées lors des épreuves de concours.

- Σ^* tous les mots
- $a\Sigma^*$ les mots commençant par a
- $\Sigma^* a$ les mots finissant par a
- $a^*|b^*$ les mots ne comportant que des a, que des b ou le mot vide
- $(ab^*a|b)^*$ les mots comportant un nombre pair de a ou le mot vide
- $(aa|b)^*$ les mots comportant des blocs de a de longueur paire ou le mot vide
- $(\Sigma^2)^*$ les mots de longueur paire

R Attention, l'ordre des puissances et des fermetures de Kleene importe : $(\Sigma^*)^2$ est équivalent à Σ^*. On peut le démontrer ainsi :

Démonstration. $(\Sigma^*)^2 = \left\{ \bigcup_{n\geqslant 0} \Sigma^n . \bigcup_{m\geqslant 0} \Sigma^m \right\} = \left\{ \bigcup_{n\geqslant 0, m\geqslant 0} \Sigma^n \Sigma^m \right\} = \left\{ \bigcup_{k\geqslant 0} \Sigma^k \right\} = \Sigma^*$ ■

R Les expressions régulières sont très puissantes et il est illusoire d'imaginer qu'on puisse les exprimer simplement avec des mots. C'est très rarement possible, uniquement sur des cas simples comme ci-dessus.

■ **Définition 174 — Langage dénoté par une expression régulière.** Pour toute expression régulière e, on note $\mathcal{L}_{ER}(e)$ le langage unique qui dénote cette expression. C'est en fait le résultat de l'application $\mathcal{L}_{ER} : \mathcal{E}_R(\Sigma^*) \longrightarrow \mathcal{P}(\Sigma^*)$ qui, à une expression régulière, fait correspondre l'ensemble des mots (le langage) engendré par cette expression régulière.

■ **Définition 175 — Sémantique des expressions régulières.** L'interprétation des expressions régulières en termes de langages, la sémantique d'une expression régulière, permet de définir inductivement l'ensemble des langages dénotés par une expression régulière ainsi :

(Base (i)) $\mathcal{L}_{ER}(\emptyset) = \{\} = \emptyset$,

(Base (ii)) $\mathcal{L}_{ER}(\epsilon) = \{\epsilon\}$,

(Base (iii)) $\forall a \in \Sigma, \mathcal{L}_{ER}(a) = \{a\}$,

(Règle de construction (i union)) $\forall e_1, e_2 \in \mathcal{E}_R, \mathcal{L}_{ER}(e_1 \mid e_2) = \mathcal{L}_{ER}(e_1) \cup \mathcal{L}_{ER}(e_2)$,

(Règle de construction (ii concaténation)) $\forall e_1, e_2 \in \mathcal{E}_R, \mathcal{L}_{ER}(e_1 e_2) = \mathcal{L}_{ER}(e_1).\mathcal{L}_{ER}(e_2)$,

(Règle de construction (iii fermeture de Kleene)) $\forall e \in \mathcal{E}_R, \mathcal{L}_{ER}(e^*) = \mathcal{L}_{ER}(e)^*$.

Théorème 28 — Un langage $\mathcal{L}$ est régulier si et seulement s'il existe une expression régulière e telle que $\mathcal{L}_{ER}(e) = \mathcal{L}$.

R Les apparences sont parfois trompeuses. Soit $\Sigma = \{a, b\}$ un alphabet et $\mathcal{L}$ le langage défini par $\mathcal{L} = \{a^n b^n, n \in \mathbb{N}\} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$. $\mathcal{L}$ n'est pas un langage régulier. Si la formule entre crochets dénote bien un langage, un ensemble de mots, on ne peut cependant pas exprimer l'ensemble de ces mots par une expression régulière. Par exemple, $a^* b^*$ est une expression régulière dont le langage associé dénote l'ensemble $\mathcal{L}$ ainsi que d'autres mots comme $\{a, b, aaa, bbb, aab, abb, abbb, \ldots\}$. Le lemme de l'étoile (cf. chapitre suivant) permet de démontrer que $\mathcal{L}$ n'est pas un langage régulier.

B Définition des langages réguliers

Il est également possible de donner une définition inductive directe des langages réguliers.

■ **Définition 176 — Ensemble des langages réguliers.** L'ensemble des langages réguliers $\mathcal{L}_{ER}$ sur un alphabet Σ est défini inductivement par :

(Base (i)) $\emptyset \in \mathcal{L}_{ER}$,

(Base (ii)) $\{\epsilon\} \in \mathcal{L}_{ER}$,

(Base (iii)) $\forall a \in \Sigma, \{a\} \in \mathcal{L}_{ER}$,

(Règle de construction (i)) $\forall \mathcal{L}_1, \mathcal{L}_2 \in \mathcal{L}_{ER}, \mathcal{L}_1 \cup \mathcal{L}_2 = \mathcal{L}_{ER}$

(Règle de construction (ii)) $\forall \mathcal{L}_1, \mathcal{L}_2 \in \mathcal{L}_{ER}, \mathcal{L}_1.\mathcal{L}_2 = \mathcal{L}_{ER}$,

(Règle de construction (iii)) $\forall \mathcal{L} \in \mathcal{L}_{ER}, \mathcal{L}^* = \mathcal{L}_{ER}$.

R Cette définition peut s'exprimer ainsi : les langages réguliers sont constitués à la base de l'ensemble vide, du langage ne contenant que le mot vide et des lettres de l'alphabet. On peut construire d'autres langages réguliers à partir de ces éléments de base en appliquant un nombre fini de fois les opérateurs d'union, de concaténation et de fermeture de Kleene.

■ **Définition 177 — Opérations régulières sur les langages.** L'union, la concaténation et la fermeture de Kleene sont les trois opérations régulières sur les langages.

R On note que l'intersection et la complémentation ne sont pas des opérations régulières.

Théorème 29 — Stabilité des langages réguliers. Les langages réguliers sont stables pour les opérations d'**union**, de **concaténation** et de **fermeture de Kleene**. Cela signifie que l'union, la concaténation ou la fermeture de Kleene de langages réguliers est un langage régulier.

Démonstration. Conséquence directe de la définition inductive. ■

C Identités remarquables sur les expressions régulières

Le tableau 12.1 recense quelques identités remarquables à connaître sur les expressions régulières. Leurs démonstrations s'appuient sur la sémantique des expressions régulières et les opérations sur les langages. Elles constituent un excellent exercice.

Démonstration. Par exemple, démontrons que $e|\emptyset = e$. D'après la sémantique des expressions régulières, on a :

$$\mathcal{L}_{ER}(e|\emptyset) = \mathcal{L}_{ER}(e) \cup \mathcal{L}_{ER}(\emptyset) \tag{12.1}$$
$$= \mathcal{L}_{ER}(e) \cup \emptyset \tag{12.2}$$
$$= \mathcal{L}_{ER}(e) \tag{12.3}$$

car l'ensemble vide est l'élément neutre de l'union ensembliste. ■

Expression régulière	Équivalent	Raison
$e\|\emptyset$	e	$\emptyset$ est l'élément neutre de l'union ensembliste
$e\emptyset$	$\emptyset$	Déf. de la concaténation, $\emptyset$ seul élément commun à $\mathcal{L}_{ER}(e)$ et $\{\emptyset\}$
$e\epsilon$	e	Déf. de la concaténation, ϵ élément neutre de la concaténation
$(e\|f)\|g$	$e\|f\|g$	Associativité de l'union ensembliste
$(e.f).g$	$e.f.g$	Associativité de la concaténation sur les langages
$e(f\|g)$	$ef\|eg$	Distributivité de la concaténation sur l'union
$(e\|f).g$	$eg\|fg$	Idem
$e\|f$	$f\|e$	Commutativité de l'union ensembliste
e^*	$\epsilon\|ee^*$	Définition de l'étoile de Kleene
e^*	$\epsilon\|e^*e$	Idem
$(\emptyset)^*$	ϵ	Définition de l'étoile de Kleene et des puissances d'un langage
$e\|e$	e	Idempotence
$(e^*)^*$	e^*	Idempotence

TABLE 12.1 – Identités remarquables des expressions régulières.

■ **Exemple 79 — Exemple de calcul sur les expressions régulières.** À l'aide des identités remarquables du tableau 12.1, il est possible de transformer des expressions régulières, de les simplifier par calcul. Voici un exemple sur $\Sigma = \{a, b\}$:

$$bb^*(a^*b^*|\epsilon)b = bb^*a^*b^*b \tag{12.4}$$

Cette expression est généralement exprimée ainsi $b^+a^*b^+$ dans les langages informatiques. Elle désigne l'ensemble des mots comportant au moins deux lettres qui commencent et

terminent par un b. S'ils contiennent des a, ceux-ci sont consécutifs et encadrés par les b. On notera que le langage associé $\mathcal{L}_{ER}(b^+a^*b^+)$ ne contient pas le mot vide.

D Arbre associé à une expression régulière

De part sa nature inductive, une expression régulière peut naturellement être représentée sous la forme d'un arbre dont les feuilles sont les éléments de base et les nœuds les opérateurs réguliers. Ces arbres permettent de prouver par induction de nombreuses propriétés des expressions régulières.

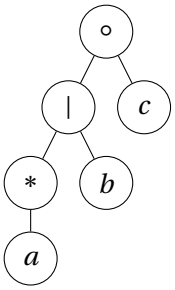


FIGURE 12.1 – Arbre associé à l'expression régulière $(a^*|b)c$

E Expressions régulières dans les langages --→ HORS PROGRAMME

Les expressions régulières sont intensivement utilisées par les informaticiens en pratique pour le test, le filtrage ou la transformation de l'information. Tous les langages évolués proposent une interface qui permet d'utiliser les expressions régulières.

Concrètement, les expressions rationnelles s'appuient sur un ensemble de caractères et de métacaractères qui permettent d'abstraire un motif de chaîne de caractères.

■ **Exemple 80 — Ensemble des lignes d'un fichier qui commencent au moins par un chiffre et qui se termine par Z.** On peut utiliser l'expression régulière suivante : `"^[0-9]+.*Z$"` où :

- `^` désigne le début d'une ligne
- `[0—9]` désigne l'ensemble des caractères 0,1,2,3,4,5,6,7,8,9
- `+` signifie au moins une occurrence du caractère précédent
- `.` symbolise n'importe quel caractère
- `*` signifie 0 ou un nombre quelconque d'occurrences du caractère précédent
- `Z` est le caractère Z
- `$` signifie la fin d'une ligne

■ **Définition 178 — Métacaractère.** Un métacaractère est un caractère qui a une signification abstraite, autre que son interprétation littérale.

Métacaractère	Signification
`^`	début de la chaîne
`$`	fin de la chaîne
`.`	n'importe quel caractère sauf retour à la ligne
`*`	0 ou plusieurs fois le caractère précédent
`+`	au moins 1 fois le caractère précédent
`?`	0 ou 1 fois le caractère précédent
`|`	alternative (union)
`()`	groupement, motif
`[]`	ensemble de caractères
`{}`	nombre de répétition du motif

TABLE 12.2 – Liste des métacaractères

Expression	Signification
`e*`	zéro ou plusieurs e
`e+`	un ou plusieurs e
`e?`	0 ou 1 fois e
`e{m}`	exactement m fois e
`e{m,}`	au moins m fois e
`e{m,n}`	au minimum m fois a et au maximum n fois e

TABLE 12.3 – Répétition des motifs

Selon le langage utilisé, on pourra utiliser les classes ou les séquences spéciales (mode Perl).

■ **Exemple 81 — Les expressions célèbres.** Voici une liste non exhaustive d'expressions régulières couramment utilisées :

- `\d+` les entiers naturels
- `– ?\d+` les entiers
- `(\d{2}|\s)]{5}` un numéro de téléphone avec des espaces
- `[A–Za–z0–9_–]{3,16}` les noms des utilisateurs d'un système (login),
- `[[:alnum:]_–^@#!$% ]{8,42}` les mots de passe,
- `([a–z0–9_.–]+)@([0–9a–z.–]+).([a–z.]{2,24})` les adresses email

Séquence	Signification
`\t`	tabulation
`\n`	nouvelle ligne
`\r`	retour chariot
`\b`	bordure de mot
`\B`	pas en bordure de mot
`\d`	correspond à n'importe quel chiffre [0-9]
`\D`	correpond à n'importe quel caractère sauf un chiffre
`\w`	correspond à n'importe quel mot [0-9a-zA-Z_]
`\W`	correspond à n'importe quelle séquence qui n'est pas un mot
`\s`	correspond à n'importe quel espace (espace, tabulation, nouvelle ligne)
`\S`	correspond à n'importe quel caractère qui n'est pas un espace

TABLE 12.4 – Séquences spéciales (mode Perl disponible en Python)

Classe	Signification
`[:alnum:]`	correspond aux caractères alphabétiques et numériques. [A-Za-z0-9]
`[:alpha:]`	correspond aux caractères alphabétiques. [A-Za-z]
`[:blank:]`	correspond à un espace ou à une tabulation
`[:cntrl:]`	correspond aux caractères de contrôle
`[:digit:]`	correspond aux chiffres [0-9]
`[:graph:]`	caractères graphiques affichables
`[:lower:]`	correspond aux caractères alphabétiques minuscules. [a-z]
`[:print:]`	caractères imprimables
`[:space:]`	correspond à tout espace blanc (espace, tabulation, nouvelle ligne)
`[:upper:]`	correspond à tout caractère alphabétique majuscule. [A-Z]
`[:xdigit:]`	correspond aux chiffres hexadécimaux. [0-9A-Fa-f]

TABLE 12.5 – Classes (mode expressions régulières étendues de grep)

13

AUTOMATES FINIS DÉTERMINISTES

À la fin de ce chapitre, je sais :

- ☞ définir un automate fini déterministe
- ☞ représenter un automate fini déterministe
- ☞ qualifier les états d'un automates en termes d'accessibilité et de co-accessibilité
- ☞ définir un langage reconnu par un automate
- ☞ compléter un automate fini déterministe
- ☞ complémenter un automate fini déterministe
- ☞ faire le produit de deux automates finis déterministes
- ☞ utiliser la stabilité des langages reconnus par complémentation et intersection

Les automates sont utilisés pour réaliser :

- l'automatisation de comportements simples (systèmes embarqués),
- des circuits électroniques, des protocoles de communication, des processus,
- la recherche d'un mot dans un texte en temps linéaire par rapport à la taille du texte,
- la compilation d'un code informatique (analyse lexicale et syntaxique)

Les automates sont des machines simples dont les entrées sont des lettres et la sortie un résultat. Ces machines sont constituées par des états qui sont interconnectés par des liens permettant le passage d'un état à un autre selon un entrée et une direction. Selon les entrées reçues, l'automate réagit et se positionne donc dans un certain état. Les automates sont des éléments essentiels de l'informatique : ils établissent un lien fort entre la théorie des graphes et la théorie des langages.

Les chapitres de ce cours se focalisent sur les automates dont le résultat est l'acceptation ou non d'un mot d'un langage. Plus particulièrement, ce chapitre traite des automates finis déterministes (AFD), le déterminisme étant la capacité de l'automate à se positionner dans un seul état possible après la réception d'un lettre.

A Automate fini déterministe (AFD)

■ **Définition 179 — Automate fini déterministe (AFD).** Un automate fini déterministe est un quintuplet $(Q, \Sigma, q_i, \delta, F)$ tel que :

1. Q est un ensemble non vide et fini dont les éléments sont les états,
2. Σ est l'alphabet,
3. $q_i \in Q$ est l'état initial,
4. $\delta : Q \times \Sigma \longrightarrow Q$ est la **fonction** de transition de l'automate,
5. $F \subseteq Q$ est l'ensemble des états accepteurs ou terminaux.

R Le déterminisme d'un AFD est dû aux faits que :

- l'état initial est un **singleton**,
- δ est une **fonction** : à un couple (état,lettre) (q, a), δ associe au plus un état q'.

■ **Définition 180 — Fonction de transition partielle.** On dit que la fonction de transition δ est partielle s'il existe au moins un couple (état,lettre) pour lequel elle n'est pas définie.

■ **Définition 181 — Automate complet.** Un AFD $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ est dit complet si δ est une **application**, c'est-à-dire δ n'est pas partielle, il existe une transition pour chaque lettre de Σ pour tous les états.

■ **Définition 182 — Automate normalisé.** Un automate est normalisé s'il ne possède pas de transition entrante sur son état initial et s'il possède un seul état final sans transition sortante.

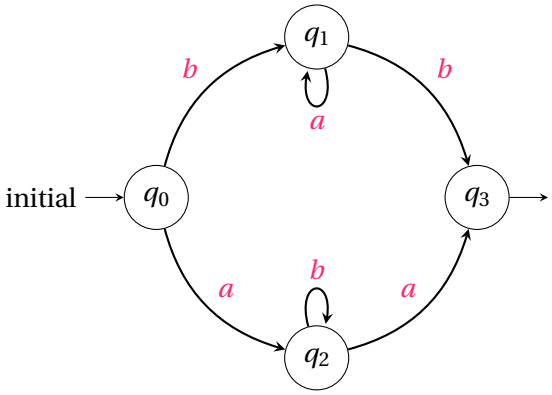


FIGURE 13.1 – Exemple d'automate fini déterministe normalisé

B Représentations d'un automate

Les automates peuvent être représentés sous la forme de tableaux ou de graphes comme le montre les figures 13.1 et 13.1. Les tableaux sont utiles lors de l'exécution manuelle des algorithmes sur les automates. Les algorithmes sur les graphes pourront être d'une aide précieuse pour l'étude des automates.

	$\downarrow q_0$	q_1	q_2	$\uparrow q_3$
a	q_2	q_1	q_3	
b	q_1	q_3	q_2	

	a	b
$\downarrow q_0$	q_2	q_1
q_1	q_1	q_3
q_2	q_3	q_2
$\uparrow q_3$		

TABLE 13.1 – Exemple d'automate fini déterministe représenté sous la forme de tableaux : on peut choisir de représenter les états en ligne ou en colonne.

C Acceptation d'un mot

■ **Définition 183 — Fonction de transition étendue aux mots.** La fonction de transition peut être étendue aux mots par passages successifs d'un état à un autre en lisant les lettres d'un mot.

On définit inductivement cette fonction étendue noté δ^* :

$$\forall q \in Q, \delta^*(q, \epsilon) = q \tag{13.1}$$

$$\forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \delta^*(q, w.a) = \delta(\delta^*(q, w), a) \tag{13.2}$$

■ **Définition 184 — Acceptation d'un mot par un automate.** Un mot $w \in \Sigma^*$ est **accepté** par un automate $\mathcal{A}$ si et seulement si $\delta^*(q_i, w) \in F$, c'est-à-dire la lecture du mot w par l'automate conduit à un état accepteur.

■ **Définition 185 — Langage reconnu par un AFD.** Le langage $\mathcal{L}_{\text{rec}}(\mathcal{A})$ **reconnu** par un automate fini déterministe $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$:

$$\mathcal{L}_{\text{rec}}(\mathcal{A}) = \{w \in \Sigma^*, w \text{ est accepté par } \mathcal{A}\} \tag{13.3}$$

■ **Définition 186 — Langage reconnaissable.** Un langage $\mathcal{L}$ sur un alphabet Σ est reconnaissable s'il existe un automate fini déterministe $\mathcal{A}$ d'alphabet Σ tel que $\mathcal{L} = \mathcal{L}_{\text{rec}}(\mathcal{A})$.

D Accessibilité et co-accessibilité

■ **Définition 187 — Accessibilité d'un état.** Un état q d'un automate est dit accessible s'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q_i, u) = q$, c'est-à-dire il est possible de l'atteindre depuis l'état initial.

■ **Définition 188 — Co-accessibilité d'un état.** Un état q d'un automate est dit co-accessible s'il existe un mot $w \in \Sigma^*$ tel que $\delta^*(q, u) \in F$, c'est-à-dire à partir de cet état, il est possible d'atteindre un état accepteur.

■ **Définition 189 — Automate émondé.** Un automate est dit émondé tous ses états sont à la fois accessibles et co-acessibles.

Théorème 30 — Automate d'un langage reconnaissable. Si un langage est reconnaissable alors il existe un automate fini déterministe :

- normalisé qui le reconnaît.
- émondé qui le reconnaît.
- complet qui le reconnaît.

E Complétion d'un AFD

Sur la figure 13.2, on observe que certaines transitions ne sont pas précisées : par exemple, si la lettre a arrive en l'état q_1, aucune transition n'est spécifiée. Compléter un automate, c'est préciser ces transitions en ajoutant un état puits.

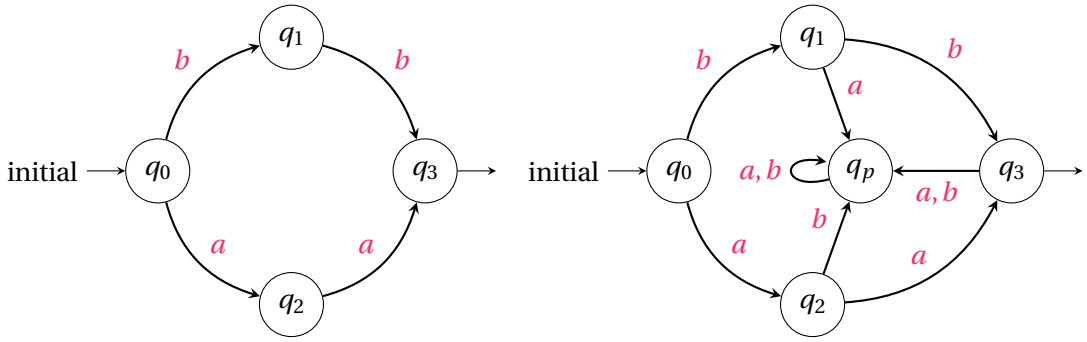


FIGURE 13.2 – Exemple d'automate fini déterministe non complet (à gauche) et complété (à droite)

M **Méthode 1 — Complété d'un AFD** Le complété d'un automate fini déterministe $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ noté $C(\mathcal{A})$ est l'automate

$$C(\mathcal{A}) = (Q \cup \{q_p\}, \Sigma, q_i, C(\delta), F) \tag{13.4}$$

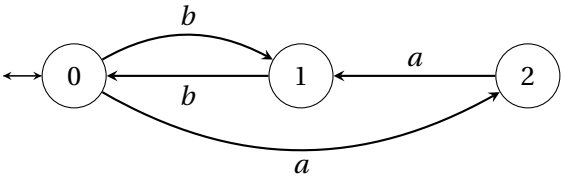
tel que :

- $q_p \notin Q$ est appelé l'état puits,
- $C(\delta)$ est l'application de $Q \cup \{q_p\} \times \Sigma$ dans $Q \cup \{q_p\}$ telle que :

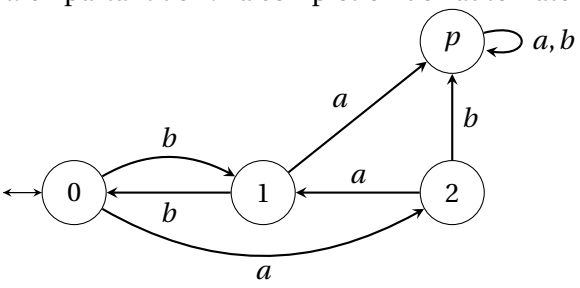
$$C(\delta)(q,a) = \begin{cases} \delta(q,a) \text{ si } \delta \text{ est définie pour } (q,a) \\ q_p \text{ sinon} \end{cases} \tag{13.5}$$

Cette méthode consiste donc à ajouter un état puits qui n'est pas co-accessible et à y faire converger toutes les transitions manquantes.

■ **Exemple 82 — Complétion d'un automate.** On considère l'automate fini déterministe suivant :



On observe que la fonction de transition n'est pas défini pour b en partant de l'état 2 ni pour a en partant de 1. La complétion de l'automate selon la méthode 1 donne :



Théorème 31 — Langage reconnu par un automate et son complété. Un automate $\mathcal{A}$ et son complété $C(\mathcal{A})$ reconnaissent le même langage :

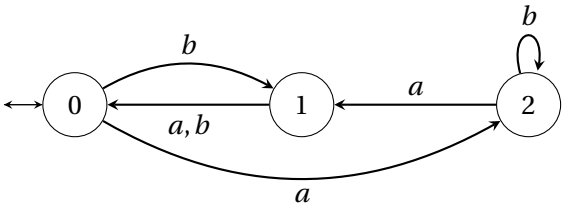
$$\mathcal{L}_{\text{rec}}(\mathcal{A}) = \mathcal{L}_{\text{rec}}(C(\mathcal{A})) \tag{13.6}$$

Démonstration. La fonction de transition pour un mot reconnu est la même sur les automates $\mathcal{A}$ et $C(\mathcal{A})$. Pour un mot non reconnu, elle diffère mais dans ce cas le mot n'appartient pas au langage. Donc les mots reconnus sont les mêmes. ■

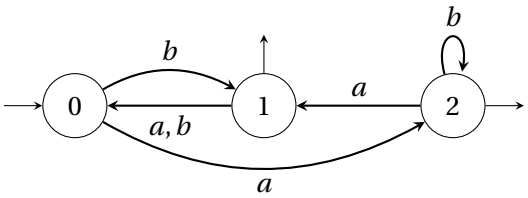
F Complémentaire d'un AFD

(M) **Méthode 2 — Complémentaire d'un AFD** Le complémentaire d'un automate fini déterministe **complet** $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ noté $\mathcal{A}^c$ est l'automate complet $\mathcal{A}^c = (Q, \Sigma, q_i, \delta, Q \setminus F)$.

■ **Exemple 83 — Complémentaire d'AFD.** On considère l'automate fini déterministe **complet** suivant :



On observe que seul l'état 0 est un état accepteur. Le complémentaire de l'automate selon la méthode 2 donne :



Théorème 32 — Langage reconnu par un automate et son complémentaire. Le langage reconnu par l'automate complémentaire d'un automate complet est le complémentaire du langage reconnu par cet automate :

$$\mathcal{L}_{\text{rec}}(\mathcal{A}^c) = \Sigma^* \setminus \mathcal{L}_{\text{rec}}(\mathcal{A}) \tag{13.7}$$

Démonstration. On procède par double inclusion.

(⊂) Si w est un mot reconnu par l'automate complémentaire, alors l'état accepteur de w n'appartient pas F, $\delta^*(q_i, w) \in Q \setminus F$. Donc w n'appartient pas à $\mathcal{L}_{\text{rec}}(\mathcal{A})$.

(⊃) Soit w un mot de l'ensemble $\Sigma^* \setminus \mathcal{L}_{\text{rec}}(\mathcal{A})$. En utilisant l'automate $\mathcal{A}$, le chemin emprunté en suivant les lettres de w mène donc à un état non accepteur de $\mathcal{A}$. Comme les états non accepteurs de $\mathcal{A}$ sont les états accepteurs de $\mathcal{A}^c$, w est donc un mot de $\mathcal{L}_{\text{rec}}(\mathcal{A}^c)$.

■

(R) Avant de complémenter un automate, il convient de vérifier que celui-ci est complet afin de ne rater aucun mot.

Théorème 33 — Stabilité des langages reconnaissables par complémentation. Les langages reconnaissables sont stable par complémentation : s'il existe un langage reconnaissable sur Σ par un automate $\mathcal{A}$, alors le complémentaire de ce langage est reconnaissable par $\mathcal{A}^c$.

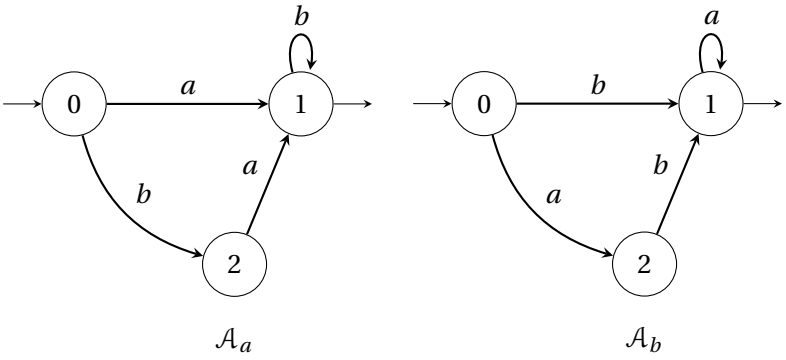
Démonstration. Ce théorème est une conséquence du théorème précédent et de la définition de l'automate complémentaire : si un langage est reconnaissable, alors son complémentaire l'est aussi puisqu'il existe un automate fini qui le reconnaît. ■

G Produit de deux AFD - Automates produit

■ **Définition 190 — Produit de deux automates.** Le produit de deux automates sur un même alphabet Σ, $\mathcal{A}_a = (Q_a, \Sigma, q_{i_a}, \delta_a, F_a)$ et $\mathcal{A}_b = (Q_b, \Sigma, q_{i_b}, \delta_b, F_b)$, noté $\mathcal{A}_a \times \mathcal{A}_b$ est l'automate $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ tel que :

- $Q = Q_a \times Q_b$
- $q_i = (q_{i_a}, q_{i_b})$
- $\delta : (Q_a \times Q_b) \times \Sigma \longrightarrow (Q_a \times Q_b)$ est définie par : $\delta((q_a, q_b), s) = (\delta_a(q_a, s), \delta_b(q_b, s))$
- $F = F_a \times F_b$

■ **Exemple 84 — Exemple d'automate produit.** On considère les deux automates suivants sur le même alphabet $\Sigma = \{a, b\}$:



Pour comprendre le fonctionnement de l'automate produit $\mathcal{A}_a \times \mathcal{A}_b$, il faut imaginer que lors du parcours lié à un mot w, on avance simultanément sur les deux automates. Si les deux s'arrêtent simultanément sur un état accepteur à la fin de la lecture de w, alors w est un mot de l'automate produit.

Par exemple, les mots ab et ba sont à la fois reconnus par $\mathcal{A}_a$ et $\mathcal{A}_b$: ce sont des mots de $\mathcal{L}_{\text{rec}}(\mathcal{A}_a \times \mathcal{A}_b)$. Par contre, aba est reconnu par $\mathcal{A}_b$ mais pas par $\mathcal{A}_a$.

Théorème 34 — Langage reconnu par un produit d'automates. Le langage reconnu par un produit d'automates est l'intersection des langages reconnus par ces automates :

$$\mathcal{L}_{\text{rec}}(\mathcal{A}_a \times \mathcal{A}_b) = \mathcal{L}_{\text{rec}}(\mathcal{A}_a) \cap \mathcal{L}_{\text{rec}}(\mathcal{A}_b) \tag{13.8}$$

Démonstration. On procède par double inclusion et on utilise la fonction de transition étendue aux mots.

($\subset$) Soit w un mot reconnu par l'automate produit. Alors, par définition de l'automate produit :

$\delta^*((q_a, q_b), w) = (\delta_a^*(q_a, w), \delta_b^*(q_b, w))$ et $\delta_a^*(q_a, w) \in F_a$ et $\delta_b^*(q_b, w) \in F_b$.
Cela signifie que w est reconnu à la fois par $\mathcal{A}_a$ et $\mathcal{A}_b$. w appartient donc aux deux langages reconnus par $\mathcal{A}_a$ et $\mathcal{A}_b$.
Donc $w \in \mathcal{L}_{\text{rec}}(\mathcal{A}_a) \cap \mathcal{L}_{\text{rec}}(\mathcal{A}_b)$.

(⊃) soit w un mot reconnu par $\mathcal{A}_a$ et par $\mathcal{A}_b$. Alors un existe un chemin dans $\mathcal{A}_a$ qui, d'après le mot w, mène à un état accepteur de F_a. De même pour F_b. Donc $(\delta_a(q_a, w), \delta_b(q_b, w)) \in F_a \times F_b$. w appartient à $\mathcal{L}_{\text{rec}}(\mathcal{A}_a \times \mathcal{A}_b)$.

■

Théorème 35 — Stabilité des langages reconnaissables par l'intersection. Les langages reconnaissables sont stables par l'intersection : l'intersection de deux langages reconnaissables est un langage reconnaissable.

Démonstration. Ce théorème est un corolaire du théorème précédent. ■

R La stabilité des langages reconnaissables est importante car nous montrerons par la suite que les langages reconnaissables sont les langages réguliers. Or, les langages réguliers ne sont pas stables par définition pour les opérations non régulières, tout comme les langages reconnaissables ne sont pas stables par définition pour les opérations régulières (union, concaténation et fermeture de Kleene). Le théorème de Kleene va nous permettre d'étendre ces résultats d'une représentation à une autre.

14

AUTOMATES FINIS NON DÉTERMINISTES

À la fin de ce chapitre, je sais :

- ☞ reconnaître un automate fini non déterministe (AFND)
- ☞ déterminiser un AFND
- ☞ construire une AFND reconnaissant un langage rationnel simple

A Automate fini non déterministe (AFND)

■ **Définition 191 — Automate fini non déterministe (AFND).** Un automate fini non déterministe est un quintuplet $(Q, \Sigma, Q_i, \Delta, F)$ tel que :

1. Q est un ensemble non vide et fini dont les éléments sont les états,
2. Σ est l'alphabet,
3. $Q_i \subseteq Q$ sont **les** états initiaux,
4. $\Delta \subseteq Q \times \Sigma \times Q$ est la **relation** de transition de l'automate,
5. $F \subseteq Q$ est l'ensemble des états accepteurs ou terminaux.

R Un AFND est par essence asynchrone. Son exécution nécessite l'exécution de toutes les transitions possibles au départ de chaque état traversé.

R Le non déterminisme d'un AFND est dû au fait que Δ n'est pas une fonction mais une relation : depuis un état q, une même lettre peut faire transiter l'AFND vers des états différents. Par exemple, (q, a, q') et (q, a, q''). Quelle transition choisir ? Là est le non déterminisme.

R Un AFND peut posséder plusieurs états initiaux, ce qui n'est pas le cas d'un AFD (q_i devient Q_i). On peut facilement se ramener à un seul état initial en utilisant des transitions spontanées. C'est pourquoi on considèrera souvent ce cas.

R Qui peut le plus peut le moins : un AFD est un cas particulier d'AFND pour lequel $\Delta = \{(q, a, q'), \delta(q, a) = q'\}$.

B Représentation d'un AFND

Les AFND peuvent être représentés de la même manière que les AFD sous la forme de tableaux ou de graphes comme le montre les figures 14.1 et 14.1.

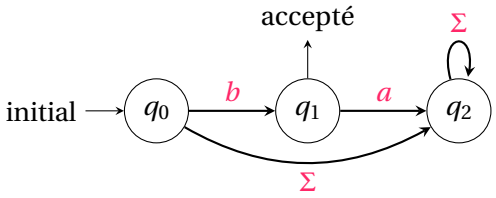


FIGURE 14.1 – Exemple d'automate fini non déterministe représenté sous la forme d'un graphe. On suppose que $\Sigma = \{a, b\}$ est l'alphabet

	$\downarrow q_0$	$\uparrow q_1$	q_2
a	q_2	q_2	q_2
b	q_1, q_2		q_2

	a	b
$\downarrow q_0$	q_2	q_1, q_2
$\uparrow q_1$	q_2	
q_2	q_2	q_2

TABLE 14.1 – Exemple d'automate fini non déterministe représenté sous la forme de tableaux : on peut choisir de représenter les états en ligne ou en colonne.

R Pour un même mot, il peut donc exister plusieurs exécutions possibles sur un AFND. La programmation des AFND n'est donc pas aussi simple que celles de AFD. Il faut être en mesure de tester tous les chemins possibles!

C Acceptation d'un mot

■ **Définition 192 — Relation de transition étendue aux mots.** La relation de transition peut être étendue aux mots par passages successifs d'un état à un autre en lisant les lettres d'un

mot. On la note Δ^* .

■ **Définition 193 — Langage reconnu par un AFND.** Le langage $\mathcal{L}_{rec}(\mathcal{A})$ reconnu par un automate fini non déterministe $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$:

$$\mathcal{L}_{rec}(\mathcal{A}) = \{w \in \Sigma^*, w \text{ est accepté par } \mathcal{A}\} \tag{14.1}$$

■ **Définition 194 — Langage reconnaissable par un AFND.** Un langage $\mathcal{L}$ sur un alphabet Σ est reconnaissable s'il existe un automate fini non déterministe $\mathcal{A}$ d'alphabet Σ tel que $\mathcal{L} = \mathcal{L}_{rec}(\mathcal{A})$.

D Déterminisé d'un AFND

M **Méthode 3 — Déterminisé d'un AFND** Le déterminisé d'un automate fini non déterministe $\mathcal{A} = (Q, \Sigma, Q_i, \Delta, F)$ est l'automate $\mathcal{A}_d = (\mathcal{P}(Q), \Sigma, q_i, \delta, \mathcal{F})$ défini par :

- $\mathcal{P}(Q)$ est l'ensemble des parties de Q,
- q_i est **l'ensemble des états initiaux**,
- $\forall \pi \in \mathcal{P}(Q), \forall a \in \Sigma, \delta(\pi, a) = \bigcup_{q \in \pi} \{q' \in Q, (q, a, q') \in \Delta\}$,
- $\mathcal{F} = \{\pi \in \mathcal{P}(Q), \pi \cap F \neq \emptyset\}$.

Ce qui signifie que :

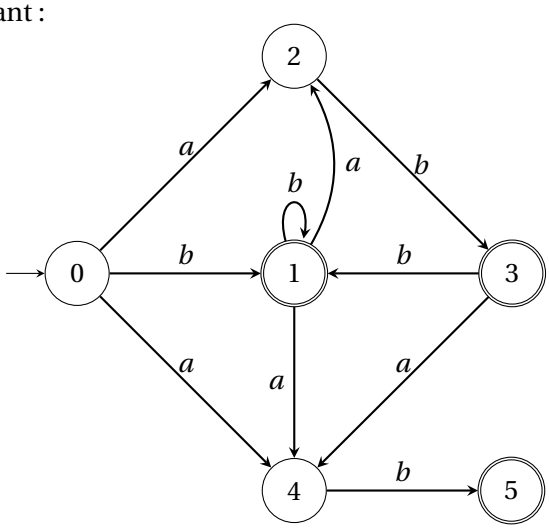
- l'état initial du déterminisé est l'état initial de l'AFND, ou bien, s'il possède plusieurs états initiaux, l'état initial constitué par la partition de tous les états initiaux de l'AFND.
- toute partie de Q est susceptible d'être un état. En pratique dans les exercices, vous construirez les états au fur et à mesure à partir de l'état initial comme dans l'exemple 85. L'ensemble des parties de Q n'a pas souvent besoin d'être explicité.
- les états accepteurs sont **les parties qui contiennent un état accepteur** de l'AFND.

L'algorithme 13 décrit cette méthode d'un point de vue opérationnel. Il s'agit de balayer un graphe en largeur et de mettre à jour les états et les transitions trouvées.

Algorithme 13 Algorithme de déterminisation d'un AFND

```
1 : Fonction DÉTERMINISER(A = (Q, Σ, Q_i, Δ, F))
2 :     transitions ← ∅
3 :     q_i ← la partition des états initiaux                    ▷ l'état initial
4 :     états ← q_i
5 :     file ← q_i
6 :     tant que file n'est pas vide répéter
7 :         q ← DÉFILER(file)
8 :         pour chaque lettre λ de l'alphabet Σ répéter
9 :             q' ← la partition des états possibles depuis (q, λ) d'après Δ
10 :            si q' n'est pas encore dans états alors
11 :                AJOUTER(états, q')
12 :                ENFILER(file,q')
13 :            AJOUTER(transitions, (q, λ, q'))
14 :    accepteurs ← ∅
15 :    pour chaque état e de états répéter
16 :        si e ∈F alors
17 :            AJOUTER(accepteurs, e)
18 :    renvoyer (états, Σ, q_i, transitions, accepteurs)
```

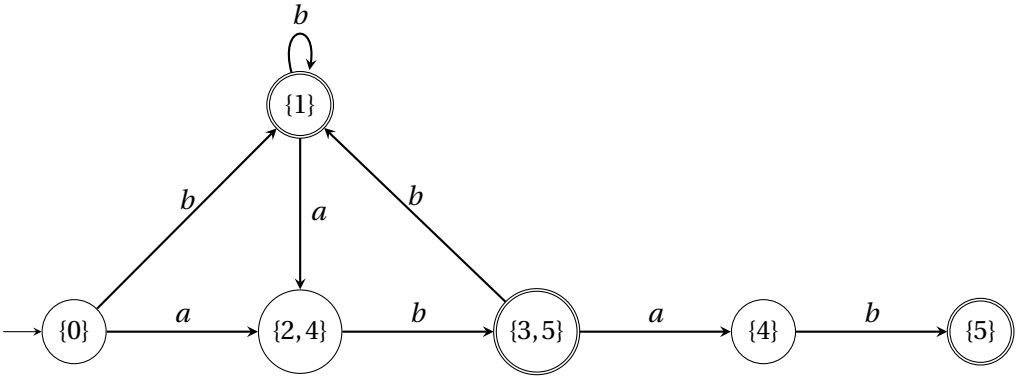
■ **Exemple 85 — Déterminiser un AFND.** On considère l'automate fini non déterministe suivant :



En construisant les parties $\mathcal{P}(Q)$ au fur et mesure à partir de l'état initial, on obtient la fonction de transition :

	$\downarrow\{0\}$	$\uparrow\{1\}$	$\{2,4\}$	$\uparrow\{3,5\}$	$\{4\}$	$\uparrow\{5\}$
a	$\{2,4\}$	$\{2,4\}$	$\emptyset$	$\{4\}$	$\emptyset$	$\emptyset$
b	$\{1\}$	$\{1\}$	$\{3,5\}$	$\{1\}$	$\{5\}$	$\emptyset$

On en déduit l'AFD suivant :



Il est possible de renommer les états arbitrairement.

Théorème 36 Un AFND $\mathcal{A}$ et son déterminisé $\mathcal{A}_d$ reconnaissent le même langage.

$$\mathcal{L}_{rec}(\mathcal{A}) = \mathcal{L}_{rec}(\mathcal{A}_d) \tag{14.2}$$

Démonstration. $\mathcal{A}_d$ est bien un automate fini déterministe car :

- il possède au plus $2^{|Q|}$ états et $|Q|$ est fini puisque $\mathcal{A}$ est fini.
- son état de départ est unique,

- d'après la définition de δ, s'il existait deux états tels que $\delta(\pi, a) = \pi_1$ et $\delta(\pi, a) = \pi_2$, alors on aurait : $\pi_1 = \pi_2 = \{q', (q, a, q') \in \Delta \text{ et } q \in \pi\}$, c'est-à-dire que ces deux états seraient égaux. Donc δ est bien une fonction de transition et $\mathcal{A}_d$ est déterministe.
- comme $\mathcal{A}$ possède au moins un état final, $\mathcal{F}$ n'est pas vide.

Ensuite, il nous faut montrer que $\mathcal{L}_{rec}(\mathcal{A}_d) = \mathcal{L}_{rec}(\mathcal{A})$.

Tout d'abord, dire que le mot vide ϵ est dans le langage $\mathcal{L}_{rec}(\mathcal{A})$ est équivalent à dire que $q_i \cap F \neq \emptyset$. Sur le déterminé, cela se traduit par $q_i \in \mathcal{F}$. Donc, si le mot vide appartient à l'un, il appartient à l'autre.

($\Longrightarrow$) Soit w un mot non vide sur Σ. Si $w = a_1 \dots a_n$ est accepté par $\mathcal{A}$, alors cela signifie qu'il existe une succession d'états $(q_i, q_1 \dots, q_n)$ de $\mathcal{A}$ telle que $q_n \in F$. Mais alors, comme les transitions sont traduites dans l'automate déterminisé, pour cette succession d'état de $\mathcal{A}$, on peut trouver une succession d'états $(q_i, \pi_1, \dots, \pi_m)$ de $\mathcal{A}_d$ telle que $\pi_m \in \mathcal{F}$ et que chaque q_j de la succession d'états de $\mathcal{A}$ fasse partie d'une partition π_k de la succession d'état de $\mathcal{A}_d$. Donc w est accepté par l'automate déterminisé.

($\Longleftarrow$) On procède de même dans l'autre sens. ■

R Le déterminisé d'un AFND $\mathcal{A}$ comporte donc plus d'états que $\mathcal{A}$. Dans le pire des cas, l'algorithme de détermination a une complexité exponentielle.

E ϵ-transitions

■ **Définition 195 — ϵ-transition.** Une ϵ-transition est une transition dans un automate non déterministe dont l'étiquette est le mot vide ϵ. C'est une transition spontanée d'un état à un autre.

R Une transition spontanée fait qu'un automate peut être considéré dans deux états simultanément, celui qui précède la transition et le suivant. C'est pourquoi un automate déterministe ne comporte pas de transitions spontanées.

R Les automates à transition spontanée ou automates asynchrones sont utilisés notamment par l'algorithme de Thompson pour passer d'une expression régulière à un automate. Ces transitions peuvent aussi servir à normaliser un automate. La plupart du temps on les insère dans un automate pour les éliminer par la suite.

Un exemple d'un tel automate est donné sur la figure 14.2. Cet automate reconnaît les mots commençant par un nombre de b quelconque suivi d'un a ou bien les mots commençant par un nombre quelconque de a suivi par b. En fait, on va voir qu'on peut très bien exprimer un tel automate de manière non déterministe sans transitions spontanées et même de manière déterministe. Les $\epsilon-$transitions n'apportent donc pas d'expressivité en plus en terme de langage.

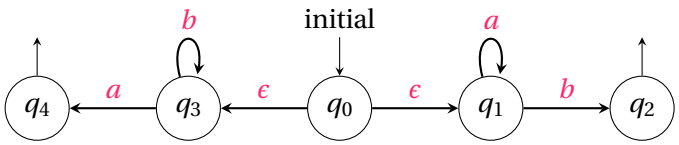


FIGURE 14.2 – Exemple d'automate fini non déterministe avec transition spontanée représenté sous la forme d'un graphe. On peut facilement les éliminer dans ce cas en créant plusieurs états initiaux.

15

DES EXPRESSIONS RÉGULIÈRES AUX AUTOMATES

À la fin de ce chapitre, je sais :

- ☞ expliquer le théorème de Kleene et ses conséquences
- ☞ transformer une expression régulière en automate
- ☞ montrer qu'un langage est local
- ☞ appliquer les algorithmes de Thompson et de Berry-Sethi
- ☞ décrire l'automate de Glushkov

A Théorème de Kleene

Les chapitres précédents ont permis de construire deux ensembles de langages :

1. l'ensemble des langages réguliers, c'est-à-dire dénotés par une expression régulière,
2. et l'ensemble des langages reconnaissables, c'est-à-dire reconnus par un automate fini.

Il s'agit maintenant d'établir une correspondance entre ces deux ensembles de langages.

Théorème 37 — Kleene. Un langage $\mathcal{L}$ sur un alphabet Σ est un langage régulier si et seulement s'il est reconnaissable.

Démonstration. ($\Longrightarrow$) Soit $\mathcal{L}$ un langage régulier. On utilise la définition inductive des langages réguliers pour montrer que ce langage est reconnaissable.

(Cas de base)
- l'ensemble vide est reconnu par un automate dont l'ensemble des états accepteurs F est vide. $\mathcal{L}_{ER}(\emptyset)$ est un langage reconnaissable.
- le mot vide est reconnu par un automate à un seul état dont l'état initial est accepteur. $\mathcal{L}_{ER}(\epsilon)$ est un langage reconnaissable.
- les lettres de l'alphabet sont des langages reconnaissables de la même manière.

(Pas d'induction) • (union) : soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages réguliers reconnus par deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Alors on a :

$$\begin{aligned}\mathcal{L}_1 \cup \mathcal{L}_2 &= \{w, w \in \mathcal{L}_1 \text{ ou } w \in \mathcal{L}_2\} \\ &= \{w, w \in \mathcal{L}_{rec}(\mathcal{A}_1) \text{ ou } w \in \mathcal{L}_{rec}(\mathcal{A}_2)\} \\ &= \mathcal{L}_{rec}(\mathcal{A}_1) \cup \mathcal{L}_{rec}(\mathcal{A}_2)\end{aligned}$$

D'après la loi de Morgan, on a $\mathcal{L}_{rec}(\mathcal{A}_1) \cup \mathcal{L}_{rec}(\mathcal{A}_2) = \overline{\overline{\mathcal{L}_{rec}(\mathcal{A}_1)} \cap \overline{\mathcal{L}_{rec}(\mathcal{A}_2)}}$. Or, les langages reconnaissables sont stables par intersection et passage au complémentaire. Ils sont donc stable pour l'union et $\mathcal{L}_{rec}(\mathcal{A}_1) \cup \mathcal{L}_{rec}(\mathcal{A}_2)$ est donc un langage reconnaissable.

- (concéténation) : soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages réguliers reconnus par deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Alors construisons l'automate $\mathcal{A}$ en reliant les états accepteurs de $\mathcal{A}_1$ à l'état initial de $\mathcal{A}_2$ par une transition spontanée (cf. figure 15.3). Cet automate $\mathcal{A}$ reconnaît alors le langage $\mathcal{L}_1.\mathcal{L}_2$.
- (fermeture de Kleene) : soit $\mathcal{L}$ un langage régulier reconnu par un automate $\mathcal{A}$. Construisons l'automate $\mathcal{A}$ associé comme sur la figure 15.4. Alors $\mathcal{A}$ reconnaît le langage $\mathcal{L}^*$.

(Conclusion) Un langage régulier $\mathcal{L}$ est un langage reconnaissable. Pour ce sens de la démonstration, on s'est appuyé sur l'algorithme de Thompson. Mais on peut aussi utiliser l'automate de Glushkov et l'algorithme de Berry-Sethi.

($\Longleftarrow$) L'algorithme de Mac Naughton-Yamada permet de calculer l'expression régulière associée à un automate fini (⇢ HORS PROGRAMME). On le montrera via l'élimination des états au prochain chapitre. ■

R Le théorème de Kleene permet d'affirmer que les langages réguliers sont stables par intersection, complémentation ce qui n'était pas évident d'après la définition des expressions régulières. Inversement, les langages reconnaissables sont stables par union, concaténation et passage à l'étoile de Kleene. Tout résultat sur un type de langage (reconnaissable ou régulier) peut se transposer à l'autre type grâce au théorème de Kleene.

Les sections qui suivent présentent les algorithmes au programme qui permettent de passer du formalisme d'une expression régulière à celui d'un automate.

B Algorithme de Thompson

L'algorithme de Thompson permet de construire un automate reconnaissant le langage dénoté par une expression régulière en utilisant des patrons de conception d'automate normalisé pour chaque cas de base et chaque opération (union, concaténation, étoile de Kleene). Cet algorithme porte également le nom de méthode compositionnelle car on compose des automates correspondants à des expressions simples.

a Patron de conception d'un cas de base

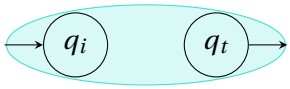
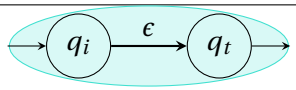
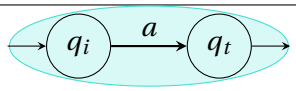
Expression régulière	Automate associé
$\emptyset$	
ϵ	
$a \in \Sigma$	

TABLE 15.1 – Automates associés aux cas de bases des expressions régulières.

b Patron de conception de l'union

On associe à l'union de deux expressions régulières $e_1|e_2$ l'automate décrit sur la figure 15.2. Au démarrage de la procédure, les deux expressions possèdent un automate équivalent comme le montre la figure 15.1.

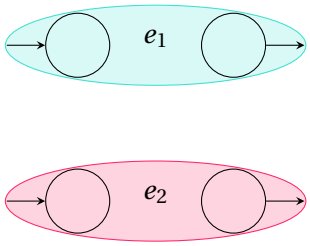


FIGURE 15.1 – Automates équivalents à e_1 et e_2 avant l'opération

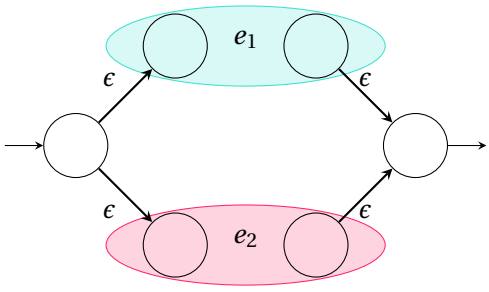


FIGURE 15.2 – Automate associé à l'union de deux expressions régulières $e_1|e_2$

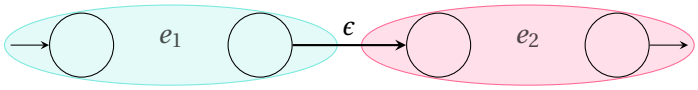


FIGURE 15.3 – Automate associé à la concaténation de deux expressions régulières $e_1 e_2$

c Patron de conception de la concaténation

On associe à la concaténation de deux expressions régulières l'automate décrit sur la figure 15.3.

d Patron de conception de l'étoile de Kleene

On associe à la fermeture de Kleene d'une expression régulière l'automate décrit sur la figure 15.4.

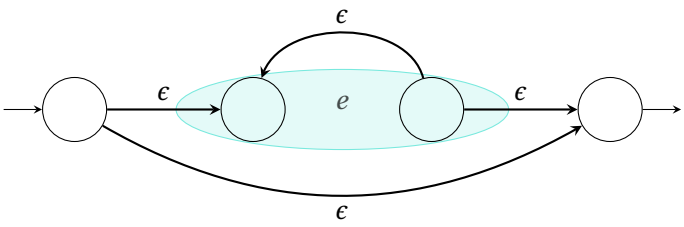
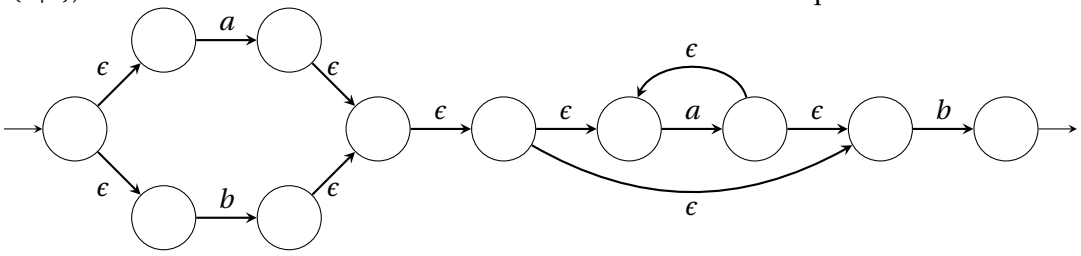


FIGURE 15.4 – Automate associé à la fermeture de Kleene de e

e Application

■ **Exemple 86** — $(a|b)a^*b$. On décompose l'expression régulière en éléments simples concaténés $(a|b)$, a^* et b et on enchaîne les automates associés. L' automate équivalent est :



f Élimination des transitions spontanées

L'automate de la figure 86 comporte un certain nombre de transitions spontanées. Si, parfois, ces transitions permettent de rendre plus lisible l'automate, celles-ci multiplient cependant les états ce qui n'est pas souhaitable, surtout dans l'optique de programmer cet automate

en le déterminisant... Il faut donc trouver une méthode pour éliminer ces transitions spontanées.

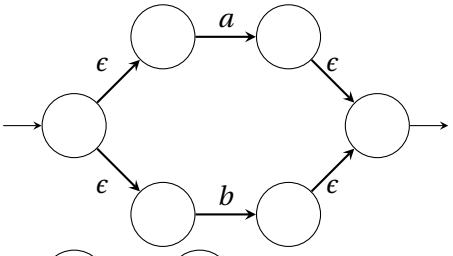
R Même s'il est possible d'éliminer les transitions spontanées une fois qu'on a construit tout l'automate associé à une expression régulière, il est souvent souhaitable de le faire à la volée afin de ne pas aboutir à un automate illisible.

M **Méthode 4 — Élimination des transitions spontanées** Il existe deux procédures similaires, une par l'avant, une par l'arrière.

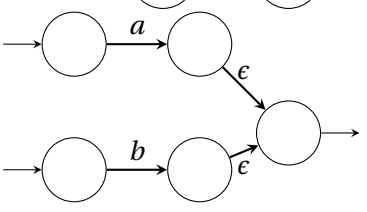
1. Fermeture par l'avant :
   - $\left(p \xrightarrow{a} q \xrightarrow{\epsilon} r\right) \rightsquigarrow \left(p \xrightarrow{a} r \text{ et } p \xrightarrow{a} q\right)$ Pour chaque transition d'un état p à un état q portant une lettre a et pour chaque transition spontanée de q à un état r, ajouter une transition de p à r portant la lettre a. Éliminer la transition spontanée.
   - $\left(\longrightarrow p \xrightarrow{\epsilon} q\right) \rightsquigarrow q \in Q_{init}$ Pour chaque transition spontanée d'un état p initial à un état q, ajouter q à l'ensemble des états initiaux. Éliminer la transition spontanée.
2. Fermeture par l'arrière :
   - $\left(p \xrightarrow{\epsilon} q \xrightarrow{a} r\right) \rightsquigarrow \left(p \xrightarrow{a} r \text{ et } q \xrightarrow{a} r\right)$ Pour chaque transition spontanée d'un état p à un état q et pour chaque transition de q à un état r portant la lettre a, ajouter une transition de p à r portant la lettre a. Éliminer la transition spontanée.
   - $\left(p \xrightarrow{\epsilon} q \longrightarrow\right) \rightsquigarrow p \in F$ Pour chaque transition spontanée d'un état p à un état accepteur q, ajouter p à l'ensemble des états accepteurs. Éliminer la transition spontanée.

■ **Exemple 87 — Élimination des transitions spontanées de l'automate de l'exemple 86.** Pour chaque automate associé, on peut déjà appliquer la fermeture avant ou arrière.

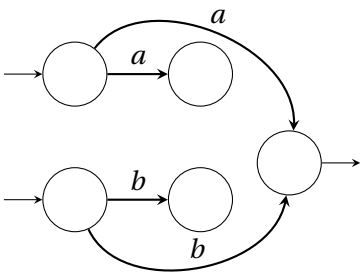
Pour l'union $a|b$, on trouve l'automate sans transitions spontanée en plusieurs étapes que voici :



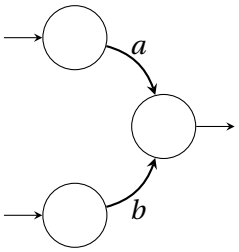
(i) état de départ



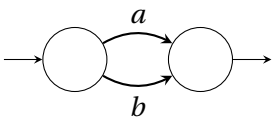
(ii) élmination par l'avant et l'état initial



(iii) élimination avant

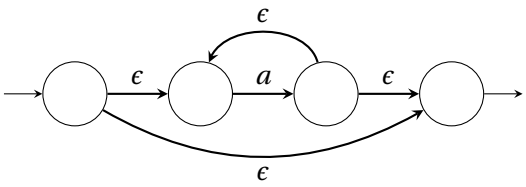


(iv) émondage

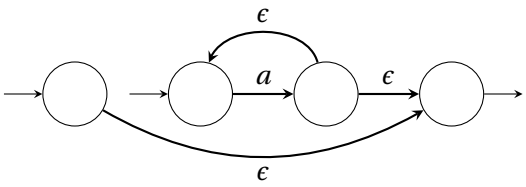


(v) un seul état initial

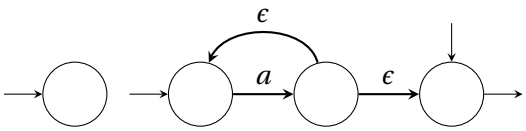
Pour la fermeture de Kleene, on procède de la même manière :



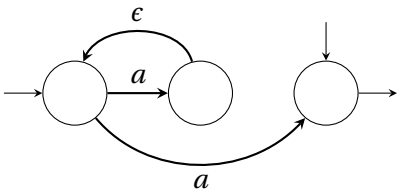
(i) état de départ



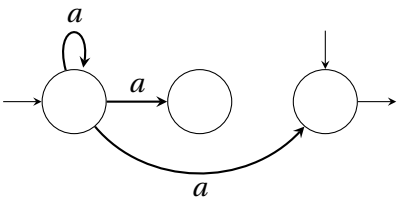
(ii) élimination par l'avant et l'état initial



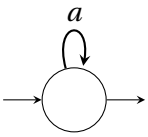
(iii) élimination par l'avant et l'état initial



(iv) émondage et élimination par l'avant

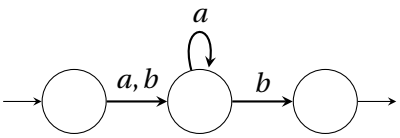


(v) élimination par l'avant



(vi) émondage et simplification

Finalement, on peut maintenant représenter l'automate normalisé (et déterministe) après élimination des transitions spontanées :



C Algorithme de Berry-Sethi et automate de Glushkov

Les notions de langage local et d'expression régulière linéaire sont introduites dans la seule perspective de construire l'automate de Glushkov[4] associé à une expression régulière linéaire par l'algorithme de Berry-Sethi[2], conformément au programme. C'est un long développement pour une procédure finale relativement simple.

a Langages locaux

■ **Définition 196 — Ensembles .** Soit $\mathcal{L}$ un langage sur Σ. On définit quatre ensembles de la manière suivante :

- les premières lettres des mots de $\mathcal{L}$

$$P(\mathcal{L}) = \{a \in \Sigma, \exists w \in \Sigma^*, aw \in \mathcal{L}\} \tag{15.1}$$

- les dernières lettres des mots de $\mathcal{L}$:

$$S(\mathcal{L}) = \{a \in \Sigma, \exists w \in \Sigma^*, wa \in \mathcal{L}\} \tag{15.2}$$

- les facteurs de longueur 2 des mots de $\mathcal{L}$:

$$F(\mathcal{L}) = \{v \in \Sigma^*, |v| = 2, \exists u, w \in \Sigma^*, uvw \in \mathcal{L}\} \tag{15.3}$$

- les facteurs de longueur 2 impossibles :

$$N(L) = \Sigma^2 \setminus F(L) \tag{15.4}$$

■ **Définition 197 — Langage local.** Un langage $\mathcal{L}$ sur Σ est local s'il existe deux parties P et S de Σ et une partie N de Σ^2 tels que :

$$\mathcal{L} \setminus \{\epsilon\} = (P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) \tag{15.5}$$

Dans ce cas, on a nécessairement $P = P(\mathcal{L})$, $S = S(\mathcal{L})$, $N = N(\mathcal{L})$.

R Cette définition signifie que l'appartenance d'un mot à un langage local peut être établie uniquement en regardant la première lettre, la dernière lettre et tous les blocs de deux lettres de ce mot. On peut imaginer que, pour vérifier, on fait glisser pour comparer tous les blocs de deux lettres non autorisés (N) sur le mot. D'où le nom local : on n'a pas besoin d'examiner dans sa globalité le mot, mais uniquement chaque lettre et sa voisine. P, S et N suffisent donc pour définir un langage local.

M **Méthode 5 — Montrer qu'un langage n'est pas local** Dans l'équation 15.5 de la définition 197, l'inclusion $\subset$ est toujours vraie. Il suffit donc de trouver un contre-exemple, c'est-à-dire un mot appartenant à $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*)$ mais pas à $\mathcal{L} \setminus \{\epsilon\}$.

■ **Exemple 88 — Langages locaux .** Sur l'alphabet $\Sigma = \{a, b\}$, on peut déterminer pour chacun des langages suivants les langages P, S et N :

1. $\mathcal{L}_{ER}(a^*)$: $P = \{a\}$, $S = \{a\}$ et $N = \{ab, ba, bb\}$.
2. $\mathcal{L}_{ER}((ab)^*)$: $P = \{a\}$, $S = \{b\}$ et $N = \{aa, bb\}$.

■ **Exemple 89 — Langages non locaux.** Sur l'alphabet $\Sigma = \{a, b\}$, on peut déterminer pour chacun des langages suivants les langages P, S et N et trouver un contre-exemple pour montrer qu'ils ne sont pas locaux :

1. $\mathcal{L}_{ER}(a^*(ab)^*)$: $P = \{a\}$, $S = \{a, b\}$ et $N = \{bb\}$. Mais on observe que le mot aba est dans $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*)$ mais pas dans $\mathcal{L}$.
2. $\mathcal{L}_{ER}(a^*|(ab)^*)$: idem

R Le dernier exemple montre que les langages locaux ne sont pas stables par union et concaténation.

Théorème 38 — L'ensemble des langages locaux est stable par intersection.

Démonstration. Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux langages locaux, et $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$. On pose $P = P(\mathcal{L})$, $S =$

$S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 196. Alors on a :

$$\begin{aligned}\mathcal{L} \setminus \{\epsilon\} &= (\mathcal{L}_1 \setminus \{\epsilon\}) \cap (\mathcal{L}_2 \setminus \{\epsilon\}) \\ &= ((P(\mathcal{L}_1)\Sigma^* \cap \Sigma^* S(\mathcal{L}_1)) \setminus (\Sigma^* N(\mathcal{L}_1)\Sigma^*)) \cap ((P(\mathcal{L}_2)\Sigma^* \cap \Sigma^* S(\mathcal{L}_2)) \setminus (\Sigma^* N(\mathcal{L}_2)\Sigma^*)) \\ &= (P(\mathcal{L}_1)\Sigma^* \cap \Sigma^* S(\mathcal{L}_1) \cap P(\mathcal{L}_2)\Sigma^* \cap \Sigma^* S(\mathcal{L}_2)) \setminus (\Sigma^* N(\mathcal{L}_1)\Sigma^* \cup \Sigma^* N(\mathcal{L}_2)\Sigma^*) \\ &= (P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N\Sigma^*).\end{aligned}$$

Donc $\mathcal{L}$ est bien local. ■

Théorème 39 — L'union de deux langages locaux définis sur deux alphabets disjoints est un langage local.

Démonstration. Soit Σ_1 et Σ_2 les alphabets **disjoints** sur lesquels sont définis $\mathcal{L}_1$ et $\mathcal{L}_2$, deux langages locaux et $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ défini sur $\Sigma = \Sigma_1 \cup \Sigma_2$. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 196.

On doit montrer l'inclusion $(P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*) \subset \mathcal{L}$ puisque l'inclusion réciproque est toujours vraie.

Considérons donc un mot $w \in (P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*)$ que l'on décompose en lettres $w = a_1 \dots a_n$. Montrons que $w \in \mathcal{L}$

- Soit $a_1 \in P(\mathcal{L}) = P(\mathcal{L}_1) \cup P(\mathcal{L}_2)$, on peut supposer sans perte de généralité que $a_1 \in P(\mathcal{L}_1)$, alors $a_1 \in \Sigma_1$.
- $a_1.a_2 \in \Sigma^2 \setminus N(\mathcal{L}) = F(\mathcal{L}_1) \cup F(\mathcal{L}_2)$, or $a_1 \in \Sigma_1$ et les alphabets Σ_1 et Σ_2 sont disjoints, donc nécessairement $a_1 a_2 \in F(\mathcal{L}_1)$ et $a_2 \in \Sigma_1$.
- De proche en proche, on montre que $a_i a_{i+1} \in F(\mathcal{L}_1)$ et $a_i \in \Sigma_1$ pour tout i.
- Enfin, $a_n \in S(\mathcal{L}) = S(\mathcal{L}_1) \cup S(\mathcal{L}_2)$ et $a_n \in \Sigma_1$ donc $a_n \in S(\mathcal{L}_1)$.
- Finalement $w \in (P(\mathcal{L}_1)\Sigma^* \cap \Sigma^* S(\mathcal{L}_1)) \setminus (\Sigma^* N(\mathcal{L}_1)\Sigma^*) = \mathcal{L}_1$ car $\mathcal{L}_1$ est local.

En partant de l'hypothèse que $a_1 \in \Sigma_2$, on en aurait conclu que $w \in \mathcal{L}_2$. On en déduit que $w \in \mathcal{L}_1 \cup \mathcal{L}_2$ et, donc, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ est un langage local. ■

Théorème 40 — La concaténation de deux langages locaux définis sur deux alphabets disjoints est un langage local.

Démonstration. Soit Σ_1 et Σ_2 les alphabets **disjoints** sur lesquels sont définis $\mathcal{L}_1$ et $\mathcal{L}_2$, deux langages locaux et $\mathcal{L} = \mathcal{L}_1\mathcal{L}_2$ défini sur $\Sigma = \Sigma_1\Sigma_2$. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 196.

On doit montrer l'inclusion $(P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*) \subset \mathcal{L}$ puisque l'inclusion réciproque est toujours vraie.

$$P(\mathcal{L}_1\mathcal{L}_2) = \begin{cases} \{a \in \Sigma_2, \exists w \in \Sigma_2^*, aw \in \mathcal{L}_2\} & \text{si } \epsilon \in \mathcal{L}_1 \\ \{a \in \Sigma_1, \exists w \in \Sigma_1^*\Sigma_2^*, aw \in \mathcal{L}_1\mathcal{L}_2\} & \text{sinon} \end{cases}$$

$$S(\mathcal{L}_1\mathcal{L}_2) = \begin{cases} \{a \in \Sigma_1, \exists w \in \Sigma_1^*, wa \in \mathcal{L}_1\} & \text{si } \epsilon \in \mathcal{L}_2^* \\ \{a \in \Sigma_2, \exists w \in \Sigma_1^*\Sigma_2^*, wa \in \mathcal{L}_1\mathcal{L}_2\} & \text{sinon} \end{cases}$$

$$F(\mathcal{L}_1\mathcal{L}_2) = \{v \in \Sigma_1^*\Sigma_2^*, |v| = 2, \exists u, w \in \Sigma_1^*\Sigma_2^*, uvw \in \mathcal{L}_1\mathcal{L}_2\}$$

On considère un mot $w \in (P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*)$ et on le décompose en lettres $w = a_0 \dots a_n$. On traite différents cas :

- Si $a_0 \in \Sigma_2$, alors $\epsilon \in \mathcal{L}_1$ et $a_2 \in P(\mathcal{L}_2)$. De proche en proche on montre que $a_i a_{i+1} \in F(\mathcal{L}_2)$, $a_i \in \Sigma_2$ pour tout i et $a_n \in S(\mathcal{L}_2)$, donc $w \in \mathcal{L}_2$ car $\mathcal{L}_2$ est local.
- Si $a_0 \in \Sigma_1$, alors $a_0 \in P(\mathcal{L}_1)$. Notons $a_0 \dots a_k$ le plus long préfixe de w qui soit dans Σ_1^*. On montre de proche en proche que $a_i a_{i+1} \in F(\mathcal{L}_1)$ pour $i < k$. Puis deux cas se présentent :
  - Si $k = n$ alors $a_n \in S(\mathcal{L}_1)$, donc $w \in \mathcal{L}_1$ car $\mathcal{L}_1$ est local.
  - Si $k < n$, on a $a_k a_{k+1} \in S(\mathcal{L}_1)P(\mathcal{L}_2)$ donc $a_k \in S(\mathcal{L}_1)$ et $a_{k+1} \in P(\mathcal{L}_2)$. On prouve alors que $a_0 \dots a_k \in \mathcal{L}_1$ et $a_{k+1} \dots a_n \in \mathcal{L}_2$ avec les mêmes arguments.

  Finalement, $w \in \mathcal{L}_1\mathcal{L}_2$ et $\mathcal{L}$ est local.

■

Théorème 41 — La fermeture de Kleene d'un langage local est un langage local.

Démonstration. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 196.
On a :

$$P(\mathcal{L}^*) = \{a \in \Sigma, \exists w \in \Sigma^*, aw \in \mathcal{L}^*\}$$

$$S(\mathcal{L}^*) = \{a \in \Sigma, \exists w \in \Sigma^*, wa \in \mathcal{L}^*\}$$

$$F(\mathcal{L}^*) = \{v \in \Sigma^*, |v| = 2, \exists u, w \in \Sigma^*, uvw \in \mathcal{L}^*\}$$

On considère un mot $u = a_0 \dots a_n \in (P(\mathcal{L}^*)\Sigma^* \cap \Sigma^* S(\mathcal{L}^*)) \setminus (\Sigma^* N(\mathcal{L}^*)\Sigma^*)$ et on cherche à montrer qu'il appartient à $\mathcal{L}^*$

De la même manière que précédemment, si $a_0 \in \Sigma$ alors $a_0 \in P(\mathcal{L})$. Les facteurs de longueur 2 de w sont dans $F(\mathcal{L})$ et $a_n \in \mathcal{L}$. On en déduit une décomposition de w en mots dans $(P(\mathcal{L})\Sigma^* \cap \Sigma^* S(\mathcal{L})) \setminus (\Sigma^* N(\mathcal{L})\Sigma^*)$ donc dans $\mathcal{L}$ car $\mathcal{L}$ est local. Finalement, $w \in \mathcal{L}^*$, car qui peut le plus peut le moins et $\mathcal{L}^*$ est local. ■

b Expressions régulières linéaires

Les langages définis par des expressions régulières ne sont donc pas toujours locaux. En revanche les langages définis par des expressions régulières **linéaires** le sont.

■ **Définition 198 — Expression régulière linéaire.** Une expression régulière e sur Σ est linéaire si toute lettre de Σ apparaît **au plus une fois** dans e.

■ **Exemple 90 — Expression régulière linéaire.** L'expression $(ab)^*$ est linéaire mais pas $(ab)^*a^*$.

Théorème 42 — Toute expression régulière linéaire dénote un langage local.

Démonstration. On procède par induction structurelle et en utilisant les propriétés des langages locaux.

Cas de base : $\emptyset$, ϵ et $a \in \Sigma$ sont des expressions régulières linéaires.

- $\mathcal{L}_{ER}(\emptyset) = \emptyset$ et on a bien $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) = \emptyset \setminus (\Sigma^* \Sigma^2 \Sigma^*) = \emptyset$
- $\mathcal{L}_{ER}(\epsilon) = \{\epsilon\}$ et on a bien $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) = \epsilon \setminus (\Sigma^* \Sigma^2 \Sigma^*) = \{\epsilon\}$
- $\mathcal{L}_{ER}(a) = \{a\}$ et on a bien $(P\Sigma^* \cap \Sigma^* S) \setminus (\Sigma^* N \Sigma^*) = \{a\} \setminus (\Sigma^* \Sigma^2 \Sigma^*) = \{a\}$

Pas d'induction :

(union) Soit Σ_1 et Σ_2 des alphabets **disjoints**. Soient e_1 et e_2 deux expressions régulières linéaires définies respectivement sur Σ_1 et Σ_2. Alors $e_1|e_2$ est régulière et linéaire et la sémantique des expressions régulières permet d'affirmer que $\mathcal{L}_{ER}(e_1|e_2) = \mathcal{L}_{ER}(e_1) \cup \mathcal{L}_{ER}(e_2)$. Or, l'union de deux langages locaux dont les alphabets sont disjoints est un langage local. Donc $\mathcal{L}_{ER}(e_1|e_2)$ est un langage local.

(concaténation) on procède de même en utilisant la concaténation de deux langages locaux.

(union) on procède de même en utilisant la fermeture de Kleene de deux langages locaux.

Finalement, les expressions régulières linéaires dénotent des langages locaux. ■

R La réciproque de ce théorème est fausse : par exemple, le langage $L(aa^*)$ est local mais aa^* n'est pas une expression régulière linéaire.

c Automate locaux

Automates locaux

■ **Définition 199 — Automate local.** Un automate fini déterministe $\mathcal{A} = (Q, \Sigma, q_i, \delta, F)$ est local si pour toute lettre $a \in \Sigma$, il existe un état $q \in Q$ tel que toutes les transitions étiquetées par a arrivent dans q.

Théorème 43 — Tout langage local $\mathcal{L}$ est reconnaissable par un automate local. De plus, si $\mathcal{L}$ ne contient pas le mot vide ϵ, alors l'automate est normalisé.

Démonstration. Soit $\mathcal{L}$ un langage local. On pose $P = P(\mathcal{L})$, $S = S(\mathcal{L})$ et $N = N(\mathcal{L})$, les ensembles définis comme en 196.

On considère l'automate $\mathcal{A} = (Q, \Sigma \cup \{\epsilon\}, q_0, \delta, S \cup \{\epsilon\})$ et la fonction δ définie par :

$$\forall a \in P, \delta(q_0, a) = q_a \tag{15.6}$$

$$\forall a_1 a_2 \in F, \delta(q_{a_1}, a_2) = q_{a_2} \tag{15.7}$$

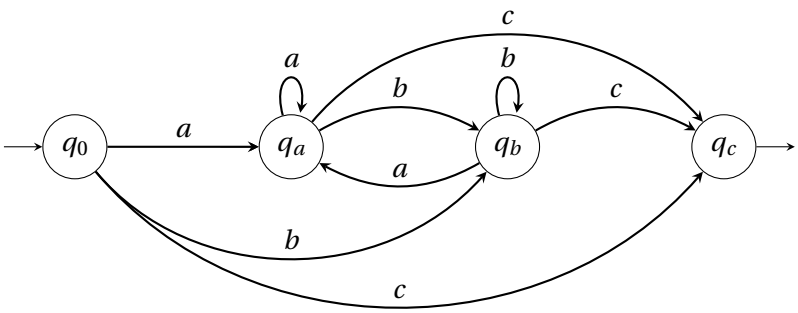
$$\tag{15.8}$$

Par construction de cet automate, un mot $w = a_0 a_1 \dots a_n$ est reconnu si et seulement si :

- $a_0 \in P$,
- $\forall i \in [\![0, n-1]\!], a_i a_{i+1} \in F$
- et $a_n \in S$.

Comme $\mathcal{L}$ est local, on a bien $\mathcal{L}_{rec}(\mathcal{A}) = \mathcal{L}$. Si $\mathcal{L}$ ne contient pas le mot vide, il suffit de l'exclure des états accepteurs et on obtient un automate normalisé. ■

■ **Exemple 91 — Automate associé à l'expression régulière linéaire** $(a|b)^*c$ **.** Le langage dénoté par cette expression régulière est local (le montrer !). On construit l'automate défini lors de la démonstration du théorème 43. Comme ce langage ne comporte pas le mot vide, l'automate est normalisé.



d Automate de Glushkov et algorithme de Berry-Sethi

On cherche maintenant un algorithme pour transformer une expression régulière (pas nécessairement linéaire) en un automate fini.

M **Méthode 6 — Algorithme de Berry-Sethi** Pour obtenir un automate fini local reconnaissant le langage $\mathcal{L}_{ER}(e)$ à partie d'une expressions régulière e sur un alphabet Σ :

1. Linéariser l'expression e : cela consiste à numéroter toutes les lettres qui apparaissent afin de créer une expression rationnelle linéaire e'.
2. Déterminer les ensembles P, S et F associés au langage local $\mathcal{L}(e')$.
3. Déterminer un automate **local** $\mathcal{A}$ reconnaissant $\mathcal{L}_{ER}(e')$ à partir de P, S et F. On peut associer ses états aux les lettres de l'alphabet de e'. L'état initial est relatif au mot vide : s'il appartient au langage, on fait de cet état un état accepteur. Toutes les transitions qui partent de l'état initial conduisent à un état associé à une lettre de P. Les états accepteurs sont associés aux éléments de S. Les facteurs de deux lettres déterminent les autres transitions.
4. Supprimer les numéros sur les transitions et faire réapparaître l'alphabet initial Σ.

L'automate obtenu est nommé automate de Glushkov[4]. C'est un automate **local et sans tran-**

sitions spontanées. Il n'est pas nécessairement déterministe mais on peut le déterminiser facilement en utilisant la procédure de déterminisation d'un AFND (cf méthode 3). Il possède $|\Sigma_e|$ états où Σ_e est l'alphabet étendu obtenu en faisant le marquage. $|\Sigma_e|$ correspond donc au nombre total de lettres dans e en comptant les répétitions. Dans le pire des cas, le nombre de transitions de l'automate est en $O(|\Sigma_e^2|)$.

M **Méthode 7 — Linéarisation de l'expression régulière** À partir de l'expression régulière de départ, on numérote à partir de 1 chaque lettre de l'expression dans l'ordre de lecture.

Si une lettre apparaît plusieurs fois, elle est numérotée autant de fois qu'elle apparaît avec un numéro différent.

Par exemple, la linéarisation de $ab|(ac)^*$ est $a_1b_1|(a_2c_1)$.

R La linéarisation d'une expression est en fait un marquage par une fonction

$$m : \{a_1, a_2, \ldots, b_1, \ldots,\} \longrightarrow \Sigma \tag{15.9}$$

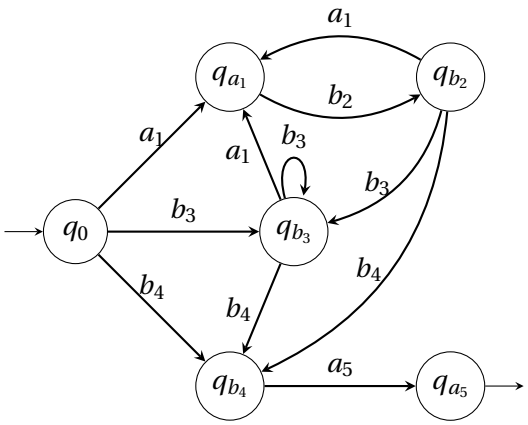
que l'on utilise également à la fin de l'algorithme de Berry-Sethi 6 pour supprimer les numéros.

La complexité de l'algorithme de Berry-Sethi est quadratique dans le pire des cas. Si n est le nombre de lettres recontrées dans l'expression linéarisée, on a :

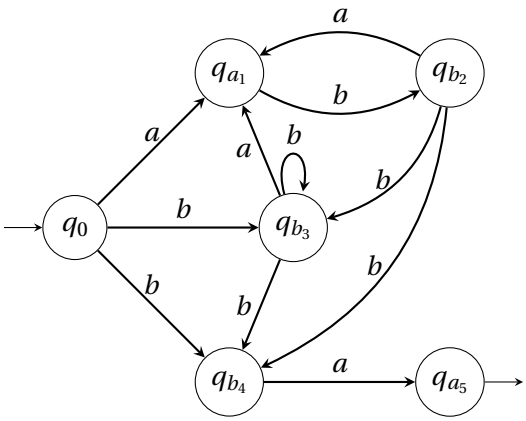
- la linéarisation est en $O(n)$,
- la construction des ensemble P, S, et F est linéaire en $O(n)$,
- la construction de l'automate avec les ensembles précédents est au pire quadratique en $O(n^2)$ (on construit n chemins qui peuvent être de longueur n),
- la suppression des marquages est linéaire en $O(n)$.

■ **Exemple 92 — Construire l'automate de Glushkov associé à l'expression régulière** $(ab|b)^*ba$**.** On applique l'algorithme de Berry-Sethi :

1. Linéarisation : $(ab|b)^*ba \longrightarrow (a_1b_2|b_3)^*b_4a_5$,
2. Construction des ensembles associés au langage local de l'expression linéarisée :
   - $P = \{a_1, b_3, b_4\}$
   - $S = \{a_5\}$
   - $F = \{a_1b_2, b_2a_1, b_2b_4, b_3b_3, b_3b_4, b_3a_1, b_4a_5, b_2b_3\}$
3. Construction de l'automate local associé :



Suppression des marquages des transitions (numéros) :

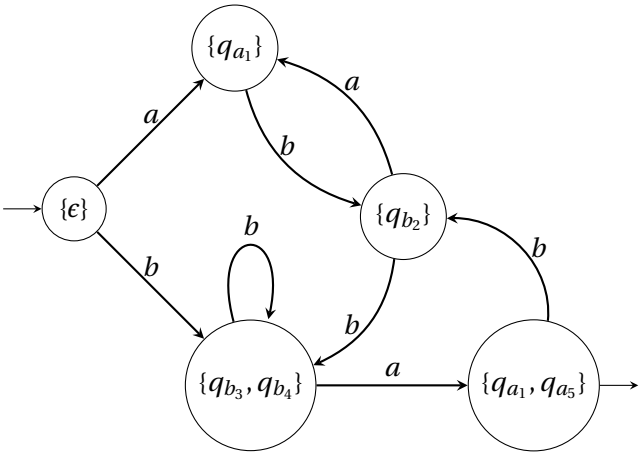


Déterminisation :

On construit la fonction de transition de proche en proche à partir de l'état initial :

	$\downarrow q_0$	$\{q_{a_1}\}$	$\{q_{b_2}\}$	$\{q_{b_3}, q_{b_4}\}$	$\uparrow\{q_{a_1}, q_{a_5}\}$
a	$\{q_{a_1}\}$		$\{q_{a_1}\}$	$\{q_{a_1}, q_{a_5}\}$	
b	$\{q_{b_3}, q_{b_4}\}$	$\{q_{b_2}\}$	$\{q_{b_3}, q_{b_4}\}$	$\{q_{b_3}, q_{b_4}\}$	$\{q_{b_2}\}$

ce qui se traduit par l'AFD :



D Comparaison Thompson / Berry-Sethi

Étape	Thompson	Berry-Sethi
Préparation	Aucune	Linéariser l'expression Construire les ensembles P, S et F
Automate	Construction directe en $O(n)$	Construction en $O(n^2)$
Finition	Supprimer les transitions spontanées	Aucun

TABLE 15.2 – Comparaison des algorithmes de Thompson et Berry-Sethi.

16

DES AUTOMATES AUX EXPRESSIONS RATIONNELLES

À la fin de ce chapitre, je sais :

- ☞ expliquer ce qu'est un automate généralisé
- ☞ fusionner des transitions multiples
- ☞ éliminer des états
- ☞ passer d'un automate à une expression régulière

Ce chapitre permet d'apporter un début de réponse à la démonstration de la réciproque du théorème de Kleene, à savoir qu'un langage reconnaissable est un langage régulier. On s'appuie pour cela sur l'algorithme d'élimination des états qui nécessite le concept d'automate généralisé.

A Automate généralisé

■ **Définition 200 — Automate généralisé.** Soit Σ un alphabet et $\mathcal{E}_R$ l'ensemble des expressions régulières sur un Σ. Un automate généralisé est un automate normalisé tel que $\mathcal{A} = (Q, \mathcal{E}_R, Q_i, \Delta, F)$, c'est-à-dire un automate normalisé dont les étiquettes des arcs sont des expressions régulières.

B D'un automate généralisé à une expression régulière

Soit un automate $\mathcal{A}$ **normalisé**. On souhaite trouver une expression régulière e telle que $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{ER}(e)$, c'est-à-dire que le langage reconnu par l'automate $\mathcal{A}$ est le même que celui dénoté par e.

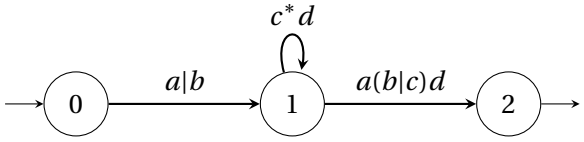


FIGURE 16.1 – Exemple d'automate généralisé sur l'ensemble des expressions régulières sur $\Sigma = \{a, b, c, d\}$

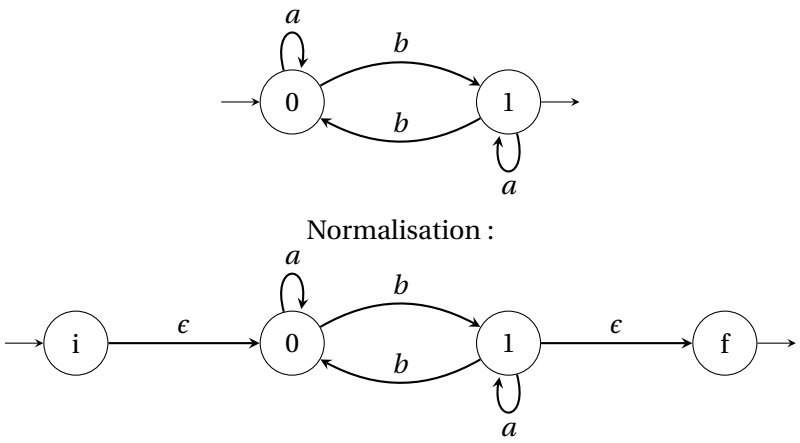


FIGURE 16.2 – Normalisation d'un automate

■ **Définition 201 — Automate normalisé.** Un automate est normalisé s'il ne possède pas de transition entrante sur son état initial et s'il possède un seul état final sans transition sortante.

R Si l'automate n'est pas normalisé, on ajoute un état initial avec une transition spontanée vers l'état initial et un état final relié par des transitions spontanées depuis les états accepteurs.

■ **Exemple 93 — Normalisation d'un automate.** La figure 16.2 montre un automate et sa version normalisée. Il faut noter que si l'automate possède plusieurs états accepteurs, il faut relier tous ces états accepteurs au nouvel état final.

La construction de Brzozowski et McCluskey est intuitive et facile à programmer. Il s'agit d'éliminer un à un les états de l'automate généralisé associé à $\mathcal{A}$. À la fin de la procédure, il ne reste plus que deux états reliés par un seul arc étiqueté par une seule expression régulière e. Le langage dénoté par cette dernière est le langage reconnu par l'automate.

M **Méthode 8 — Construire l'expression régulière équivalent à un automate normalisé**
Deux grandes étapes sont nécessaires pour construire l'expression régulière équivalent à un automate. **Pour chaque état q à éliminer**, c'est-à-dire les états autres que l'état initial ou l'état final,

1. **fusionner** les expressions régulières des transitions au départ de q_s et à destination du même état q_n comme illustré sur la figure 16.3. Formellement, si on a les transitions (q_s, e_1, q_n) et (q_s, e_2, q_n), alors on fusionne les deux expressions en faisant leur somme : $(q_s, e_1|e_2, q_n)$. On ne conserve ainsi qu'une seule expression par destination au départ de q_s.
2. **éliminer l'état** q_s en mettant à jour les transitions au départ des états précédents comme l'illustre la figure 16.4. Considérons chaque transition de type (q_p, e_1, q_s) et (q_s, e_2, q_n), c'est-à-dire les transitions pour lesquelles q_s intervient. Si on souhaite éliminer q_s, il faut considérer à chaque fois deux cas :
   (a) une transition boucle (q_s, e_b, q_s) existe : alors il est nécessaire d'ajouter la transition $(q_p, e_1 e_b^* e_2, q_n)$,
   (b) dans le cas contraire, il suffit d'ajouter la transition $(q_p, e_1 e_2, q_n)$.

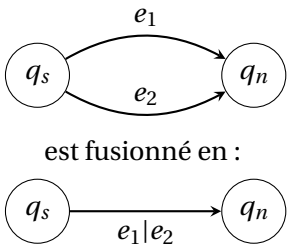
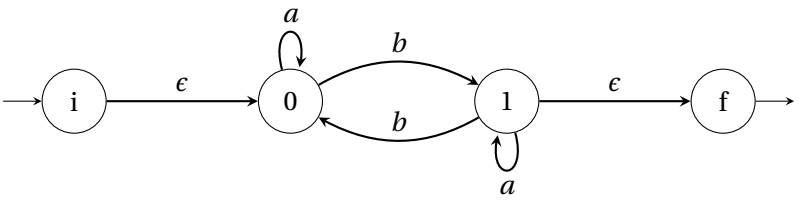
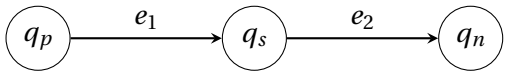


FIGURE 16.3 – Fusion de deux arcs au départ d'un état q_s et à destination du même état q_n

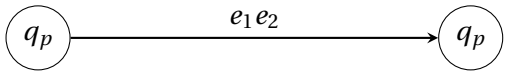
■ **Exemple 94 — Élimination des états d'un automate généralisé et normalisé .** Considérons l'automate normalisé de la figure 16.2.



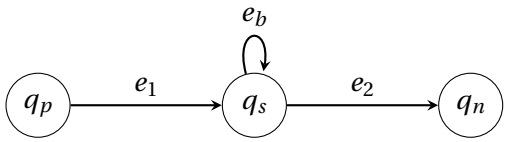
Élimination de l'état 1 :



après élimination de q_s, on obtient :



Dans le cas où il existe une boucle :



après élimination de q_s, on obtient :

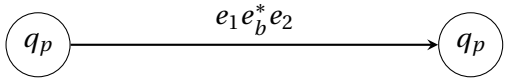
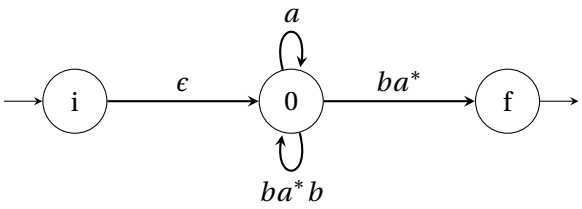
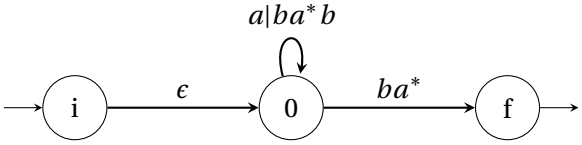


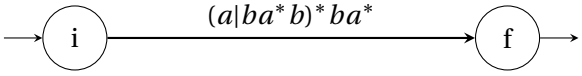
FIGURE 16.4 – Élimination d'un état q_s.



Fusion des arcs partant de 0 à destination de 0 :



Élimination de l'état 0 :



L'expression régulière équivalente à l'automate est donc $(a|ba^*b)^*ba^*$.

17

AU-DELÀ DES LANGAGES RÉGULIERS

À la fin de ce chapitre, je sais :

- ☞ expliquer les limites des langages réguliers
- ☞ montrer qu'un langage n'est pas régulier

A Limites des expressions régulières

Les langages réguliers permettent de reconnaître un motif dans un texte. Néanmoins, ils ne permettent pas de mettre un sens sur le motif reconnu : celui-ci est reconnu par l'automate mais en quoi est-il différent d'un autre mot reconnu par cet automate ? Par exemple, on peut reconnaître les mots qui se terminent par *tion* mais on ne saura pas faire la différence sémantique entre *révolution* et *abstention*.

Un autre exemple classique est l'interprétation des expressions arithmétiques : comment comprendre que $a \times b - c$ se calcule $(a \times b) - c$ et pas $a \times (b - c)$. Les deux motifs sont des expressions arithmétiques valides mais elle ne s'interprètent pas de la même manière. C'est là une des limites des langages réguliers : une fois motif reconnu, on ne peut pas l'interpréter. Pour la dépasser, il faut utiliser les notions de grammaires --→ HORS PROGRAMME .

Une autre question se pose : comment savoir si un langage est régulier sans pour autant exhiber un automate ? Comment caractériser formellement un langage régulier ?

B Caractériser un langage régulier

R Si un langage est de cardinal fini, alors il est régulier. En effet, chaque mot du langage peut être décrit par une expression rationnelle composée des symboles mêmes des mots. On peut alors construire l'automate qui reconnaît ce langage par la méthode compositionnelle (construction de Thompson). La question de la caractérisation d'un langage régulier concerne donc celle d'un langage de cardinal **infini**.

Théorème 44 — Lemme de l'étoile. Pour tout langage **régulier** $\mathcal{L}$ sur une alphabet Σ, on a :

$$\exists n \geq 1, \forall w \in \mathcal{L}, |w| \geq n \Rightarrow \exists x, y, z \in \Sigma^*, w = xyz \wedge \left(y \neq \epsilon \wedge |xy| \leq n \wedge \mathcal{L}_{ER}(xy^*z) \subseteq \mathcal{L}\right) \quad (17.1)$$

Démonstration. Soit $\mathcal{L}$ un langage régulier sur un alphabet Σ. D'après le théorème de Kleene, il existe un automate fini $\mathcal{A}$ à n états qui reconnaît $\mathcal{L}$. Soit w un mot reconnu par l'automate $\mathcal{A}$ à n états de longueur m. Il existe un chemin dans $\mathcal{A}$ qui part de l'état initial q_0 et s'achève sur un état accepteur q_m.

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} q_m$$

En numérotant de manière incrémentale les états de 0 à m, on a nécessairement $m > n$. D'après le principe des tiroirs, comme l'automate ne possède que n états, ce chemin repasse par certains états. Prenons le premier état par lequel le chemin repasse et notons le i. Il existe donc deux entiers i et j tels que $0 < i < j \leqslant n < m$ et $q_i = q_j$, c'est-à-dire il existe un cycle de longueur $j - i$ sur le chemin. Comme il s'agit du premier état par lequel on repasse, les état q_0 jusqu'à q_{j-1} sont tous distincts.

On choisit alors de poser $x = a_1 \dots a_{i-1}$, $y = a_i \dots a_{j-1}$ et $z = a_j \dots a_m$. On remarque que $w = xyz$ et que x et xy vérifient les propriétés du lemme de l'étoile car y n'est pas vide et $|xy| \leqslant n$. Il reste à montrer que $xy^*z \subseteq \mathcal{L}$. Comme le chemin reconnaissant y est un cycle (cf. figure 17.1), on peut le parcourir autant de fois que l'on veut, 0 ou k fois, le mot sera toujours reconnu par l'automate. ■

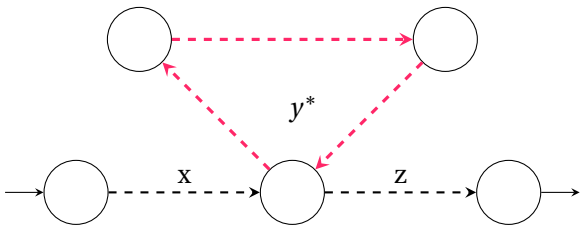


FIGURE 17.1 – Illustration du lemme de l'étoile : si le nombre de lettres d'un mot reconnu w est plus grand que le nombre d'états de l'automate n, alors il existe une boucle sur laquelle on peut itérer.

Théorème 45 — Principes des tiroirs. Si $n+1$ éléments doivent être placés dans n ensembles, alors il existe au moins un ensemble qui contient au moins 2 éléments. Autrement dit, si E et F sont deux ensembles finis tels que $|E| > |F|$, alors il n'existe aucune application injective de E dans F.

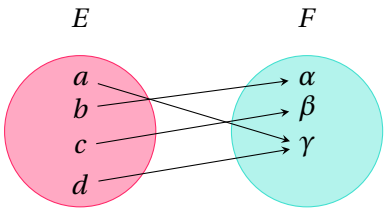


FIGURE 17.2 – Illustration du principe des tiroirs : on ne peut pas ranger les éléments de E dans les tiroirs de F sans en mettre deux dans un tiroir.

R Le lemme de l'étoile est parfois appelé le lemme de l'itération car on peut itérer autant de fois que l'on veut y.

Vocabulary 9 — Pumping lemma ⇜⇝ Lemme de l'étoile

R Il faut remarquer que le lemme de l'étoile peut être vérifié par un langage non régulier : il s'agit d'une condition **nécessaire pour être régulier mais pas suffisante**. C'est pourquoi, la plupart du temps, on utilise le lemme de l'étoile dans sa forme contraposée pour montrer qu'un langage n'est pas régulier : **s'il ne le vérifie pas, il n'est pas régulier**.

C Les langages des puissances

■ **Définition 202 — Langage des puissances.** On appelle langage des puissances le langage défini par :

$$\mathcal{L}_p = \{a^n b^n, n \in \mathbb{N}\} \tag{17.2}$$

Théorème 46 — Le langage des puissances n'est pas régulier.

Démonstration. Par l'absurde en utilisant le lemme de l'étoile.

Supposons que $\mathcal{L}_p$ soit régulier. Alors il vérifie le lemme de l'étoile. **Soit $\mathcal{A}$ un automate à n état qui reconnaît** $\mathcal{L}$. Considérons le mot $w = a^n b^n \in \mathcal{L}_p$. On a bien $|w| = 2n \geqslant n$. On peut donc appliquer le lemme de l'étoile à w.

D'après ce lemme, il existe une décomposition de w en xyz qui vérifie $|xy| \leqslant n$ et $y \neq \epsilon$. Soit i et j deux entiers naturels tels que $i + j \leqslant n$ et $j > 0$. Cette décomposition de w est nécessairement de la forme générale $w = a^i a^j a^{n-i-j} b^n = xyz$, avec $x = a^i$, $y = a^j$ et $z = a^{n-i-j} b^n$.

Les conditions du lemme sont vérifiées et il est donc possible d'itérer sur y : un tel mot appartient toujours au langage. Donc le mot $xy^2z = a^i a^{2j} z = a^i a^{2j} a^{n-i-j} b^n$ devrait appartenir à $\mathcal{L}_p$. Or ce n'est manifestement pas le cas car $i+2j+n-i-j = n+j > n$ car $j > 0$. C'est pourquoi $\mathcal{L}_p$ n'est pas un langage régulier. ■

R Le théorème 46 un résultat théorique important à connaître car :

- on peut s'en servir pour démontrer la non régularité d'autres langages en utilisant la stabilité de l'intersection pour les langages réguliers.
- la démonstration est canonique, c'est-à-dire typique de l'utilisation du lemme de l'étoile.

18

À SAVOIR POUR LES CONCOURS

A Algorithmes à savoir écrire en OCaml en moins de 5 minutes

- recherche du minimum d'un tableau,
- algorithme d'Euclide,
- produit matriciel naïf,
- évaluation de polynôme via la méthode de Horner,
- exponentiation rapide,
- décomposition en base b,
- suite récurrente linéaire en temps constant (impératif et récursif) par exemple Fibonacci,
- insertion d'un élément dans une liste triée,
- tri par insertion,
- coder le miroir d'une liste en temps linéaire en sa taille,
- recherche par dichotomie dans un tableau trié,
- calculer la taille et la hauteur d'un arbre.

B Algorithmes classiques à savoir implémenter en 10 minutes

- tri rapide,
- tri fusion,
- parcours en largeur et en profondeur d'un graphe,
- parcours préfixe, infixe, postfixe d'un arbre binaire,
- recherche, insertion et suppression dans un ABR,
- création, extraction et ajout dans un tas min ou tas max,
- tri par tas,
- les opérations sur une file de priorités implémentées par un tas,
- les opérations sur une structure union-find.

C Algorithmes dont la connaissance aide à finir plus vite

- algorithme de Quine,
- algorithme de Huffman,
- algorithme de Floyd-Warshall,
- algorithme de Dijkstra,
- algorithme A*,
- algorithme de Kruskal,
- algorithme de Prim,
- tri topologique,
- détection de cycles dans un graphe orienté,
- calcul de couplage maximum dans graphe biparti (chemin augmentant),
- les algorithmes classiques de programmation dynamique : rendu de monnaie, plus longue sous séquence commune, distance d'édition, sac à dos.

D Automates et langages réguliers

Les éléments théoriques à connaître :

- définition inductive des mots,
- lemme de Lévi,
- définition inductive des expressions régulières,
- sémantique des expressions régulières,
- définition inductives des langages réguliers,
- théorème de Kleene,
- stabilité des langages réguliers (union, concaténation, fermeture de Kleene),
- stabilité des automates finis (intersection (produit), complémentation),
- constructions de Thompson,
- lemme de l'étoile.

À savoir faire :

1. compléter, complémenter ou émonder un automate,
2. construire un automate pour l'union, le complémentaire, l'intersection, la concaténation,
3. exhiber un automate ou une expression régulière associés à un langage simple,
4. déterminiser un automate fini non déterministe (via l'automate des parties),
5. trouver l'automate correspondant à une expression régulière en utilisant l'algorithme de Berry-Sethi (automate local, automate de Glushkov),

6. trouver l'expression régulière correspondant à un automate par élimination des états (automates généralisés),
7. éliminer les ε-transitions d'un automate,
8. utiliser le lemme de l'étoile pour montrer qu'un langage n'est pas rationnel,

E Arbres et graphes

- maîtriser les différentes représentations des graphes et leurs intérêts (listes et matrices d'adjacence),
- connaître le lien entre les puissances de la matrice d'adjacence et l'existence de chemins dans un graphe,
- savoir démontrer la correction et la terminaison du parcours en largeur d'un graphe,
- connaître et savoir montrer les différentes caractérisations d'un arbre,
- savoir montrer les encadrements classiques entre hauteur et taille d'un arbre binaire,
- savoir transformer un arbre d'arité quelconque en arbre binaire.

F Logique

- savoir modéliser des propositions en langage naturel par des propositions logiques,
- savoir établir rapidement une table de vérité,
- savoir mettre une formule sous forme normale disjonctive ou conjonctive, d'après un table de vérité ou en calculant,
- connaître les lois de la logique des propositions,
- savoir évaluer une formule logique étant donnée une valuation entière binaire,
- maîtriser le vocabulaire : satisfaisable, tautologie, antilogie, valuation, modèle, équivalence, conséquence sémantique, séquent, prémisse, conclusion, règle d'inférence.
- connaître les règles de la déduction naturelle et s'entraîner à les appliquer sur des exemples.

G Techniques algorithmiques

- Savoir calculer la complexité d'un programme impératif,
- Savoir calculer la complexité d'un programme récursif d'après la relation de récurrence,
- Connaître les complexités associées aux relations :
  - $T(n) = 1 - T(n-1)$
  - $T(n) = 1 + T(n/2)$
  - $T(n) = n + 2C(n/2)$
  - $T(n) = 1 + 2T(n-1)$
- savoir démontrer la terminaison d'un algorithme impératif ou récursif,
- savoir démontrer la correction d'un algorithme,

H Techniques mathématiques

Savoir démontrer une propriété :

- par récurrence,
- par induction structurelle,
- par double inclusion,
- par contraposition,
- par l'absurde,
- par analyse-synthèse.

Septième partie

Annexes

BIBLIOGRAPHIE

Articles

[1] Walter William Rouse BALL. “The eight queens problem”. In : *Mathematical recreations and essays* (1960), pages 97-102 (cf. page 110).

[2] Gerard BERRY et Ravi SETHI. “From regular expressions to deterministic automata”. In : *Theoretical computer science* 48 (1986). Publisher : Elsevier, pages 117-126 (cf. page 187).

[3] David GALE et Lloyd S SHAPLEY. “College admissions and the stability of marriage”. In : *The American Mathematical Monthly* 69.1 (1962). Publisher : Taylor & Francis, pages 9-15 (cf. page 143).

[4] Victor Mikhaylovich GLUSHKOV. “The abstract theory of automata”. In : *Russian Mathematical Surveys* 16.5 (1961). Publisher : IOP Publishing, page 1 (cf. pages 187, 192).

[5] Kurt GÖDEL. “Die Vollständigkeit der Axiome des logischen Funktionenkalküls”. In : *Monatshefte für Mathematik und Physik* 37.1 (1er déc. 1930), pages 349-360. ISSN : 1436-5081. DOI : 10.1007/BF01696781. URL : https://doi.org/10.1007/BF01696781 (visité le 15/03/2019) (cf. page 59).

[6] Donald E KNUTH. “Dynamic huffman coding”. In : *Journal of Algorithms* 6.2 (1er juin 1985), pages 163-180. ISSN : 0196-6774. DOI : 10.1016/0196-6774(85)90036-7. URL : https://www.sciencedirect.com/science/article/pii/0196677485900367 (visité le 17/01/2022) (cf. page 102).

[7] Joseph B KRUSKAL. “On the shortest spanning subtree of a graph and the traveling salesman problem”. In : *Proceedings of the American Mathematical society* 7.1 (1956). Publisher : JSTOR, pages 48-50 (cf. page 125).

[8] Robert Clay PRIM. “Shortest connection networks and some generalizations”. In : *The Bell System Technical Journal* 36.6 (1957). Publisher : Nokia Bell Labs, pages 1389-1401 (cf. page 125).

[9] Lloyd SHAPLEY et Herbert SCARF. “On cores and indivisibility”. In : *Journal of mathematical economics* 1.1 (1974). Publisher : Elsevier, pages 23-37 (cf. page 143).

[10] Jeffrey Scott VITTER. “Design and analysis of dynamic Huffman codes”. In : *Journal of the ACM* 34.4 (1er oct. 1987), pages 825-845. ISSN : 0004-5411. DOI : 10.1145/31846.42227. URL : https://doi.org/10.1145/31846.42227 (visité le 17/01/2022) (cf. page 102).

[11] Terry A. WELCH. “A technique for high-performance data compression”. In : *Computer* 17.6 (1984). Publisher : IEEE Computer Society, pages 8-19 (cf. page 103).

[12] Jacob ZIV et Abraham LEMPEL. “A universal algorithm for sequential data compression”. In : *IEEE Transactions on information theory* 23.3 (1977). Publisher : IEEE, pages 337-343 (cf. page 103).

[13] Jacob ZIV et Abraham LEMPEL. “Compression of individual sequences via variable-rate coding”. In : *IEEE transactions on Information Theory* 24.5 (1978). Publisher : IEEE, pages 530-536 (cf. page 103).

Livres

Sites web