Langages & Automates Finis

-

Cours IUP-MIAGE 2ème année

Version 1.0

Emmanuel Desmontils
Sophie Coudert
Département Informatique
Faculté des sciences et des techniques de Nantes

24 mars 1999

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Mise en garde sur ce polycopié

Ce polycopié est une première version probablement imparfaite. Aussi, nous demandons à tout lecteur de ne pas hésiter à prodiguer remarques et conseils aussi bien sur le fond que sur la forme.

Notez qu'il constitue un *complément* du cours d'automates dispensé en licence IUP-MIAGE. A ce titre, il ne prétend pas remplacer le cours mais présente, de manière plus rigoureuse et formelle, les notions introduites.

Il existe des pages Web concernant ce module [1]. Le lecteur peut y trouver :

- la version la plus récente de ce document ;
- les transparents utilisés durant le cours ;
- les feuilles d'exercices de TD et de TP ;
- des références sur les documents issus du Web utilisés pour construire ce cours (voir la bibliographie).

1.2 Présentation du cours

Le module «Automates» vise à introduire les bases de la théorie des langages et des automates. Il permet aussi d'introduire des notions utiles au module de Compilation en maîtrise IUP-MIAGE, particulièrement en ce qui concerne l'analyse lexicographique d'un programme.

On y définit les notions suivantes : vocabulaire, langage, grammaires, classification de Chomsky, langages relationnels, expressions régulières, machine de Turing, automates déterministes et non-déterministes. On présente aussi bien les bases théoriques nécessaires à la bonne compréhension de ces notions que des algorithmes de base permettant de les manipuler. En particulier, nous nous attachons à présenter les outils permettant de construire un analyseur de chaînes de symboles à partir d'une ou plusieurs expressions décrivant leur construction (le langage) [2].

1. «http://www.sciences.univ-nantes.fr/info/perso/permanents/desmontils/Automates.html».
2. Les séances de travaux dirigés et de travaux pratiques permettent aux étudiants de développer les différents concepts définis en cours.

1.3 Plan de ce document

Nous débuterons ce cours (Chapitre 2) par une présentation des bases sur les langages formels en introduisant différentes notions importantes comme celles de vocabulaire, langage et grammaire. Nous insisterons notamment sur une classe de langage particulière : les langages réguliers. Nous en donnerons les définitions et les outils permettant de les décrire (les expressions régulières) et de les manipuler.

Ensuite, nous introduirons rapidement les machines de Turing (Chapitre 3) avant de nous attarder sur un cas particulier : les automates finis (Chapitre 4). Nous donnerons toutes les définitions nécessaires à leur utilisation.

Nous proposerons ensuite une succession d'algorithmes permettant de transformer ces automates finis (Chapitre 5). Nous présenterons aussi une méthode permettant de passer d'une expression régulière (décrivant un langage régulier) à un automate fini déterministe et minimal capable de reconnaître des mots de ce langage.

Enfin, nous exposerons des applications possibles des automates finis (Chapitre 6). Nous verrons en particulier leur utilisation dans le processus de compilation d'un langage informatique et dans le traitement du langage naturel.

Chapitre 2

Langages et grammaires

2.1 Introduction

Les langages peuvent prendre plusieurs types de formes : naturel, mathématique, langage de programmation (Pascal, C...)...

En fin des années 50, le linguiste Noam CHOMSKY propose une première formalisation de la description des langages formels. Un langage formel est constitué d'un vocabulaire et de règles de grammaires entièrement connus. La théorie des langages formels s'occupe soit de générer l'ensemble des phrases d'un langage soit de déterminer si une phrase appartient ou non au langage.

Nous allons faire une introduction à la théorie des langages. D'abord nous présenterons les notions de base tournant autour des langages (alphabet, chaîne, langage...) ainsi que les opérations associées. Nous nous intéresserons ensuite à un cas particulier de langage appelé *langage rationnel*. Nous donnerons des outils permettant de les décrire dont principalement *les expressions régulières*. Nous présenterons aussi une méthode permettant de déterminer si un langages n'est pas rationnel basée sur le *lemme de la pompe*. Enfin, nous introduirons les grammaires permettant de définir un langage. Une grammaire (ou syntaxe) est un ensemble de règles applicables à un vocabulaire définissant les phrases correctes du langage. Cette analyse se situe à un niveau purement syntaxique. Une phrase appartenant à un langage peu ne pas avoir de sens d'un point de vue sémantique mais en avoir du point de vue syntaxique. Une grammaire a deux fonctions : produire (des phrases syntaxiquement correctes) et reconnaître (des phrases comme syntaxiquement correctes).

2.2 Langages & Opérations associées

Dans cette section nos présentons brièvement la structure de monoïde qui est la base de la représentation mathématique des mots. Ensuite nous définissons les mots comme les éléments de monoïdes particuliers construits sur des ensembles de base appelés alphabets. Et enfin, nous définissons les langages comme des ensembles de mots précis. Et sur les langages on définit des opérations permettant, en les combinant entre eux, d'en obtenir de nouveaux.

2.2.1 Notion de monoïde

Définition 2.2.1 *(monoïde) Un monoïde est un ensemble M muni d'une opération binaire interne associative $\circ$ et d'un élément neutre ε. Il est parfois noté $< M, \circ, \varepsilon >$.*

Par exemple, $< \mathbb{N}, +, 0 >$ est un monoïde, où $\mathbb{N}$ est l'ensemble des entiers naturels. En effet, $+$ est associative et 0 est neutre pour $+$. De même, $< \mathbb{N}, *, 1 >$ est un monoïde.

Définition 2.2.2 *(Sous-monoïde) Soit $< M, \circ, \varepsilon >$ un monoïde et T une partie de M. $< T, \circ, \varepsilon >$ est un sous-monoïde de $< M, \circ, \varepsilon >$ si c'est un monoïde, c'est-à-dire si $\varepsilon \in T$ et $\forall t, t' \in T,\ t \circ t' \in T$*

Par exemple, $< \mathbb{P}, +, 0 >$ est un sous-monoïde de $< \mathbb{N}, +, 0 >$, où $\mathbb{P}$ est l'ensemble des entiers naturels pairs.

Définition 2.2.3 *(monoïde engendré) Soit $\mathcal{M} =< M, \circ, \varepsilon >$ un monoïde. Pour toute partie A de M on peut définir le plus petit sous-monoïde de $\mathcal{M}$ contenant A. On l'appelle le sous-monoïde de $\mathcal{M}$ engendré par A.*

Par exemple, le sous-monoïde de $< \mathbb{N}, +, 0 >$ engendré par $\{3\}$ est l'ensemble des naturels multiples de 3 (de même que celui engendré par $\{3, 6\}$).

Remarque *Soit $\mathcal{M} =< M, \circ, \varepsilon >$ un monoïde et A une partie de M. On peut montrer que le sous-monoïde de $\mathcal{M}$ engendré par A est $\bigcup_{n=0}^{\infty} A^n$, où*

$$A^0 = \{\varepsilon\} \quad et \quad \forall n \in \mathbb{N},\ T^{n+1} = T \circ T^n = \{x \circ y \mid x \in T, y \in T^n\}$$

2.2.2 Mots & monoïde libre

Intuitivement, un mot est un ensemble de symboles d'un alphabet placés les uns après les autres donc une suite de lettre (ou chaîne de caractères, dans une terminologie plus informatique).

Définition 2.2.4 *(Alphabet, Mots & Mot vide)*

- *Un alphabet $\mathcal{A}$ est un ensemble fini et non vide de symboles appelés lettres ou caractères.*
- *Un mot (ou* chaîne*) sur un alphabet $\mathcal{A}$ est une suite finie de lettres de $\mathcal{A}$. Si $[p]$ désigne la suite d'entiers $(1, 2, ..., p)$, un mot m est une application de $[p]$ dans $\mathcal{A}$ ($m : [p] \rightarrow \mathcal{A}$) pour un certain p de $\mathbb{N}$. p est appelé* la longueur *du mot m, notée $|m|$.*
- *Le mot vide est le mot de longueur 0, c'est-à-dire ne contenant aucun caractère. Il est souvent noté ε.*

Par exemple, $\mathcal{A}_1 = \{1\}$, $\mathcal{A}_2 = \{0,1\}$, $\mathcal{A}_3 = \{., -, /\}$, $\mathcal{A}_4 = \{0,1,2,...,9\}$, $\mathcal{A}_5 = \{a,b,...,z\}$ et $\mathcal{A} = \{a,b,c\}$ sont des alphabets; et $abbc, aaaa, bc, ...$ sont des mots sur $\mathcal{A}$.

Par convention (valable dans la suite du document), nous posons :

- Que $a, b, c, d, e, f, 0, 1...$ désignent des caractères.
- Que t, u, v, x, y, z désignent des chaînes.
- La chaîne $aaaa$ est aussi notée a^4. Par conséquent, $a^0 = \varepsilon$.
- Le caractère $m(i)$ désigne le $i$èmes caractère de la chaîne m.

Dans la suite, nous définissons les mots comme les éléments de monoïdes particuliers. On peut définir sur les mots (ou chaînes) une opération particulière : la concaténation. Intuitivement, si x et y sont des chaînes, alors le résultat de la concaténation de x et y est la chaîne xy. Cette opération est associative : $(xy)z = x(yz) = xyz$ (Les parenthèses sont ici des symboles mathématiques et n'appartiennent pas à l'alphabet de référence). De plus, la chaîne vide ε est neutre pour la concaténation. L'ensemble des mots sur un alphabet donné peut ainsi être muni d'une structure de monoïde. Ce type de monoïde, un peu particulier, est appelé monoïde libre.

Définition 2.2.5 *(Concaténation & monoïde libre) étant donné un alphabet $\mathcal{A}$,*

- *On note $\mathcal{A}^*$ l'ensemble de tous les mots sur $\mathcal{A}$. Dans $\mathcal{A}^*$, on définit une opération dite* produit de concaténation *ou simplement* concaténation. *Elle est noté $\oplus$ ou plus souvent implicite (simple juxtaposition des opérandes). Cette opération est une loi de composition interne associative définie de la façon suivante : soit u et v deux mots de longueur p et q, la concaténation de u et v (dans cet ordre) est le mot w de longueur $p+q$ défini par $w(i) = u(i)$ si $1 \leq i \leq p$ et $w(i) = v(i)$ si $p+1 \leq i \leq q$. Autrement dit, w est construit par juxtaposition des mots u et v, c'est-à-dire $w = u \oplus v = uv$.*
- *Le monoïde $(\mathcal{A}^*, \oplus, \varepsilon)$ est appelé le* monoïde libre *engendré par $\mathcal{A}$.*

Par exemple, si $x = ab$ et $y = cd$, alors $x \oplus y = xy = abcd$. De plus, $x \oplus \varepsilon = \varepsilon \oplus x = x$ et, par exemple, $(ab \oplus cd) \oplus ef = abcd \oplus ef = abcdef = ab \oplus (cdef) = ab \oplus (cd \oplus ef)$. L' ensemble $\mathcal{A}^*$ des mots ainsi construits est infini dénombrable.

Théorème 2.2.1 *Tout mot x (de longueur n) sur un alphabet $\mathcal{A}$ se décompose de façon unique en $x_1 \oplus \cdots \oplus x_n$, également noté $x_1 \cdots x_n$, où x_1 et $\cdots$ et x_n sont des lettres de $\mathcal{A}$.*

Remarque *Une chaîne sur un alphabet $\mathcal{A}$ peut également être définie récursivement par :*

- *la chaîne vide ε est une chaîne sur $\mathcal{A}$;*
- *si x est une chaîne sur $\mathcal{A}$ et $a \in \mathcal{A}$ alors la chaîne ax est une chaîne sur $\mathcal{A}$;*
- *aucun autre objet n'est une chaîne sur $\mathcal{A}$.*

Cette définition sert de base pour la démonstration de nombreuses propriétés à propos des mots, par récurrence sur leur longueur. L'idée générale est que pour montrer qu'une propriété P est vraie de tous les mots, il suffit de montrer que P est vraie de ε (cas de base) et que si P est vraie d'un mot x (hypothèse de récurrence), alors elle est vraie de ax, pour toute lettre a.

Notations

- L'ensemble des mots (ou chaînes) de longueur n (pour $n \in \mathbb{N}$) construits sur $\mathcal{A}$ est noté $\mathcal{A}^n$ ($\mathcal{A}^n = \{x_1 x_2 ... x_n \mid \forall i,\ 1 \leq i \leq n,\ x_i \in \mathcal{A}\}$. Par conséquent, toutes les chaînes y de l'ensemble $\mathcal{A}^3\}$) ont pour longueur $|y| = 3$.
- L'ensemble des chaînes de longueur finie et non nulle définies sur $\mathcal{A}$ est noté $\mathcal{A}^+ = \bigcup_{n=1}^{\infty} \mathcal{A}^n = \{x \mid \exists n \in \mathbb{N}, n \geq 1 \text{ et } x \in \mathcal{A}^n\}$. Notons que $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$.
- Par convention, $\mathcal{A}^0$ est $\{\varepsilon\}$ et donc $\mathcal{A}^* = \bigcup_{n=0}^{\infty} \mathcal{A}^n$

Par exemple, si l'on considère l'alphabet $\mathcal{A} = \{0, 1\}$ alors :

- $\mathcal{A}^0 = \{\varepsilon\}$
- $\mathcal{A}^1 = \mathcal{A} = \{0, 1\}$;
- $\mathcal{A}^2 = \{0, 1\}^2 = \{00, 01, 10, 11\}$;
- $\mathcal{A}^3 = \{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$;
- ...
- $\mathcal{A}^* = \{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, ..., 11111, ..., 1001101110...\}$.
- $\mathcal{A}^+ = \{0, 1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, ..., 11111, ..., 1001101110...\}$.
- si $S = \{\varepsilon, 0, 00, 000, 0000, ...\} = \{0^i\} \subset \mathcal{A}^+$ (avec $\varepsilon = 0^0$) alors $< S, \oplus, \varepsilon >$ est le sous-monoïde de $< \mathcal{A}^*, \oplus, \varepsilon >$ engendré par $\{0\}$.

Remarque *Si $c_1 \in \mathcal{A}^n$ et $c_2 \in \mathcal{A}^m$ alors $c_1 \oplus c_2 \in \mathcal{A}^{n+m}$.*

2.2.3 Relations entre mots

Ici, nous présentons quelques relations classiques entre mots. Tout d'abord, intuitivement, un mot est préfixe (respectivement suffixe puis sous-chaîne) d'un autre mot, s'il en est un début (respectivement une fin, puis un morceau).

Définition 2.2.6 *(Préfixe, suffixe et sous-chaîne) Si x et y sont deux chaînes quelconques sur $\mathcal{A}$, alors*

- *x est un préfixe de y s'il existe z dans $\mathcal{A}^*$ tel que $y = xz$.*
- *x est un suffixe de y s'il existe z dans $\mathcal{A}^*$ tel que $y = zx$.*
- *x est une sous-chaîne de y s'il existe z et z dans $\mathcal{A}^*$ tel que $y = wxz$.*

Si $x \neq y$ (c'est-à-dire si $z \neq \varepsilon$), alors le préfixe ou le suffixe est dit propre.

Par exemple, dans la chaîne 001110, 00 est un suffixe, 10 est un préfixe et 0111 est une sous-chaîne.

Remarque *De manière plus générale, x est un préfixe de $x \oplus y \oplus z$, z en est un suffixe et y en est une sous-chaîne. x, comme y et z peuvent très bien être nuls et donc, ε est préfixe, suffixe et sous-chaîne de toute chaîne.*

Définition 2.2.7 *(Occurrence) Une occurrence d'une chaîne x dans une chaîne y est l'apparition de x à un endroit précis dans la chaîne y.*

Par exemple, avec $\mathcal{A} = \{a, b\}$, le mot $y = abababababa$ comporte quatre occurrences du mot $x = aba$.

Par ailleurs, notons que dans $\mathcal{A}^*$ définit sur $\mathcal{A} = \{x_1, x_2, ..., x_n\}$, il est possible de définir plusieurs relations d'ordre :

- l'ordre préfixiel, ordre partiel définit par : $u < v$ si et seulement si u est un préfixe de v ;
- l'ordre lexicographique (ordre du dictionnaire), ordre total définit par : $x_i \leq x_j$ si $i \leq j$ et $u < v$ si et seulement si $u = u_1yu_2$ et $v = v_1zv_2$ tels que $u_1 = v_1$ et $y < z$ avec y et $z \in \mathcal{A}$;
- l'ordre hiérarchique, ordre total pour lequel les mots sont classés en premier lieu par longueur, puis pour les mots de même longueur, par ordre lexicographique.

2.2.4 Langages & Opérations sur les langages

Définition 2.2.8 *(Langage - version 1 -) Un langage L sur un alphabet $\mathcal{A}$ est un ensemble de chaînes (ou ensemble de mots) sur $\mathcal{A}$. L est donc un sous-ensemble de $\mathcal{A}^*$ et, autrement dit, l'ensemble des langages sur $\mathcal{A}$ est l'ensemble $\mathcal{P}(\mathcal{A}^*)$ des parties de $\mathcal{A}^*$.*

Par exemple, si l'on considère l'alphabet $\mathcal{A} = \{0, 1\}$ alors $L_1 = \{0, 00, 1, 01, 11, 10\}$ est un langage sur $\mathcal{A}$. De même, pour tout entier naturel n, $\mathcal{A}^n$ est un langage sur $\mathcal{A}$, où, rappelons le, $\mathcal{A}^n$ est l'ensemble des mots de longueur n sur $\mathcal{A}$. Si de plus, pour toute lettre a, a^n désigne le mot $a \overset{n \text{ fois}}{\ldots} a$, formé de n a consécutifs, alors $L_2 = \{0^n1^n \mid n \geq 0\}$, $L_3 = \{0^n10^m \mid n \geq 0, m \geq 1\}$, $L_4 = \{1^n \mid n \geq 2\}$ et $L_5 = \{0^i \mid i \geq 0\} = \{0\}^*$ sont d'autres exemples de langages sur $\mathcal{A}$. Parmi tous les langages sur un alphabet $\mathcal{A}$ donné, on en distingue quelques uns, particuliers, dont par exemple les suivants :

Définition 2.2.9 *étant donné un alphabet $\mathcal{A}$, parmi tous les langages L de $\mathcal{P}(\mathcal{A}^*)$,*

- *Le* langage neutre *est celui dont le seul mot est la chaîne vide : $L = \{\varepsilon\}$.*
- *Le* langage vide *est celui qui ne contient aucun mot, soit $L = \emptyset$.*
- *Un* langage fini *est un langage qui contient un nombre fini de mots.*
  Un langage infini *est un langage non vide et non fini.*
- *Un langage L est dit posséder* la propriété préfixe (resp. suffixe) *si aucune chaîne de L n'est préfixe propre (resp. suffixe propre) d'une autre chaîne de L .*

Par exemple, $L = \{a^ib \mid i \geq 0\} = \{b, ab, aab, aaab, ...\}$ est dit préfixe mais n'est pas suffixe. $L = \{a^n \mid n \in \mathbb{N}\}$ n'est ni l'un ni l'autre. Notons par ailleurs que $\emptyset \neq \{\varepsilon\}$.

Comme les langages sont des ensembles, on peut leur appliquer les opérations ensemblistes classiques : union, intersection, complémentation, etc. De plus, par extension de la concaténation des mots aux langages, on peut définir quelques autres opérateurs.

Définition 2.2.10 *Soit $\mathcal{A}$ un alphabet. On définit sur les langages de $\mathcal{P}(\mathcal{A}^*)$ les opérateurs suivants : soient L et M deux langages sur $\mathcal{A}$,* *Opérateurs ensemblistes classiques :*

- union : $L \cup M = \{x \mid x \in L \text{ ou } x \in M\}$;
- intersection : $L \cap M = \{x \mid x \in L \text{ et } x \in M\}$;
- exclusion : $L/M = \{x \mid x \in L \text{ et } x \notin M\}$;
- complémentaire sur $\mathcal{A}^*$: $\mathcal{A}^*/L = \{x \mid x \in \mathcal{A}^* \text{ et } x \notin L\}$;

Opérateurs induits par la concaténation des mots :

- concaténation des langages : $LM = L \oplus M = \{xy \mid x \in L \text{ et } y \in M\}$.
- fermeture de Kleene : $L^* = \bigcup_{i=0}^{\infty} L^i$ *où* $L^0 = \{\varepsilon\}$ *et* $L^n = LL^{n-1}$;
- fermeture positive : $L^+ = \bigcup_{i=1}^{\infty} L^i$.

Intuitivement, la concaténation[1] de deux langages est l'ensemble des mots obtenus en concaténant un mot du premier langage avec un mot du second. Par exemple, si $L_1 = \{a, bc\}$ et $L_2 = \{de, f\}$ alors $L_1 \oplus L_2 = \{ade, af, bcde, bcf\}$. Sur ces opérateurs entre langages, on a entre autres les quelques propriétés suivantes :

- Le langage vide est absorbant pour la concaténation des langages : $\emptyset L = \emptyset = \emptyset L$
- $(\mathcal{P}(\mathcal{A}^*), \oplus, \{\varepsilon\})$ est un monoïde:
  - Le langage neutre est neutre pour la concaténation des langages : $\{\varepsilon\}L = L = L\{\varepsilon\}$
  - La concaténation des langages est associative : $(L_1L_2)L_3 = L_1(L_2L_3)$
- $L^+ = LL^* = L^*L \quad \text{et} \quad L^* = \{\varepsilon\} \cup L^+$
- $\emptyset^* = \{\varepsilon\} \quad \text{et} \quad \{\varepsilon\}^* = \{\varepsilon\}$

Jusqu'ici, un langage sur un alphabet $\mathcal{A}$ pouvait être n'importe quelle partie dans $\mathcal{P}(\mathcal{A}^*)$, ce qui est très vaste. En particulier, au moment de vérifier si un mot appartient à un langage, il est utile d'avoir une caractérisation précise de ce langage, ce qui n'est pas toujours évident si l'on considère une partie infinie quelconque dans $\mathcal{P}(\mathcal{A}^*)$. On va donc, d'un point de vue pragmatique, s'intéresser à des classes de langages particuliers pour lesquels on a des descriptions finies, utilisables pour décider si oui ou non, un mot est dans un langage. Parmi ces types de langages, on s'intéresse en premier lieu aux langages rationnels.

2.2.5 Langages & Expressions rationnels

Parmi tous les langages constitués par les parties d'un monoïde libre $\mathcal{A}^*$ engendrés par un alphabet $\mathcal{A}$, on distingue une classe particulière : la classe $Rat(\mathcal{A}^*)$ des *langages rationnels* sur $\mathcal{A}$. L'intérêt de cette classe est qu'il existe une méthode simple permettant de décider pour chacun de ses langages si un mot lui appartient ou non : les automates finis sur lesquels nous reviendrons plus loin dans ce polycopié (cf. Chapitre 4).

1. Par extension, cette concaténation est notée $\oplus$ comme celle des mots (et le symbole $\oplus$ est souvent omis), mais il s'agit bien de deux concaténations différentes : l'une entre mots et l'autre entre ensembles de mots.

Une partie de $\mathcal{A}^*$ est un langage rationnel si on peut l'obtenir à partir des parties finies de $\mathcal{A}^*$ (langages finis sur $\mathcal{A}$) en utilisant uniquement les trois opérations $\cup, \oplus$ et *. Et comme toute partie finie peut être vue comme la réunion de singletons, on peut définir les langages rationnels de la façon suivante :

Définition 2.2.11 *(Langage rationnel) Soit $\mathcal{A}$ un alphabet. Les* langages rationnels sur $\mathcal{A}$ *sont les éléments de la classe $Rat(\mathcal{A}^*)$ définie inductivement de la façon suivante :*
$Rat(\mathcal{A}^*)$ est le plus petit sous-ensemble de $\mathcal{P}(\mathcal{A})$ tel que :

1. *$\emptyset \in Rat(\mathcal{A}^*)$;*
2. *$\{\varepsilon\} \in Rat(\mathcal{A}^*)$;*
3. *$\forall a \in \mathcal{A},\ \{a\} \in Rat(\mathcal{A}^*)$;*
4. *Si $L_1 \in Rat(\mathcal{A}^*)$ et $L_2 \in Rat(\mathcal{A}^*)$, alors $L_1 \cup L_2 \in Rat(\mathcal{A}^*)$;*
5. *Si $L_1 \in Rat(\mathcal{A}^*)$ et $L_2 \in Rat(\mathcal{A}^*)$, alors $L_1 \oplus L_2 \in Rat(\mathcal{A}^*)$;*
6. *Si $L_1 \in Rat(\mathcal{A}^*)$ alors $L_1^* \in Rat(\mathcal{A}^*)$.*

Par exemple, si l'on considère l'alphabet $\mathcal{A} = \{a, b, c\}$, alors l'ensemble des mots qui ne sont formés que de a est dans $Rat(\mathcal{A}^*)$. Ce langage est $\{a\}^*$. On peut l'obtenir à partir des points 3, puis 6 de la définition qui précède. Toute partie finie L de $\mathcal{A}^*$ est dans $Rat(\mathcal{A}^*)$: L peut être obtenue par un certain nombre d'applications des points 1, 2, 3 et 4 (démonstration par récurrence sur le cardinal de L). L'ensemble des mots sur $\mathcal{A}$ commençant par un b est également un langage de $Rat(\mathcal{A}^*)$. Ce langage est $b \oplus (\{a\} \cup \{b\} \cup \{c\})^*$. Il est obtenu en applicant successivement les points 3 (3 fois), 4 (2 fois), 6 (1 fois) et 5 (1 fois).
On peut donc construire à l'aide des opérateurs de la définition précédente de nombreux langages désignés par des expressions mathématiques comme par exemple $L = \{((\{a\} \cup \{b\})^* \oplus \{a\})^* \oplus (\{b\} \oplus \{c\})^*\}$, ce qui, en appliquant les définitions peut se simplifier en $L = \{(\{a, b\}^* \oplus \{a\})^* \oplus \{ac\}^*\}$. Cette écriture toutefois reste lourde, en particulier à cause des nombreuses accolades qu'elle comporte. Il existe donc pour les langages rationnels une écriture simplifiée, plus compacte et pratique : c'est ce qu'on appelle les expressions rationnelles. Chaque expression rationnelle désigne un langage rationnel précis.

Définition 2.2.12 *(Expression rationnelle) Les expressions rationnelles sur $\mathcal{A}$ décrivent les langages rationnels. Elles sont définies de la façon suivante :*

- *$\emptyset$ est une expression rationnelle qui décrit le langage rationnel $\emptyset$;*
- *ε est une expression rationnelle qui décrit le langage rationnel $\{\varepsilon\}$;*
- *pour tout a de $\mathcal{A}$, a est une expression rationnelle qui décrit le langage rationnel $\{a\}$;*
- *Si l_1 et l_2 sont des expressions rationnelles qui décrivent L_1 et $L_2 \in Rat(\mathcal{A}^*)$ alors :*
  - *$(l_1 \mid l_2)$ est une expression rationnelle qui décrit le langage rationnel $L_1 \cup L_2 \in Rat(\mathcal{A}^*)$,*
  - *$(l_1 l_2)$ est une expression rationnelle qui décrit le langage rationnel $L_1 \oplus L_2 \in Rat(\mathcal{A}^*)$,*
  - *$(l_1)^*$ est une expression rationnelle qui décrit le langage rationnel $(L_1)^*$.*

Autrement dit, par rapport à la précédente, l'écriture sous forme d'expression rationnelle consiste essentiellement à omettre les accolades, à systématiser l'omission du symbole $\oplus$, à remplacer "$\cup$" par[2] "|" et à utiliser les parenthèses pour lever les ambigüités. Comme la réunion et la concaténation sont associatives, on omettra généralement les parenthèses inutiles. On écrira par exemple $(u \mid v \mid w)$ plutôt que $((u \mid v) \mid w)$ ou $(u \mid (v \mid w))$ et de même pour la concaténation. Bien sûr, il est sous-entendu qu'aucun des symboles "|", "$*$", "ε", "(" et ")" n'appartient à l'alphabet $\mathcal{A}$. Sinon, il faut les différencier en renomant les symboles.

Théorème 2.2.2 *Tout langage dénoté par une expression rationnelle est un langage rationnel.*

En résumé, étant donné un alphabet $\mathcal{A}$ et $\mathcal{C} = \{|, *, \varepsilon, (,)\}$, tel que $\mathcal{A} \cap \mathcal{C} = \emptyset$,

- Un élément ω de $(\mathcal{A} \cup C)^*$ est une expression rationnelle si et seulement si :
  - $\omega \in \mathcal{A}^*$;
  - ω est de la forme ω_1^*, ou $\omega_1\omega_2$ ou $(\omega_1 \mid \omega_2 \mid \omega_3 \mid ... \mid \omega_n)$ avec ω_i une expression rationnelle $\forall i$ et $\forall n \geq 1$.
- Intuitivement une expression rationnelle désigne un langage rationnel de la façon suivante :
  - $\omega \in \mathcal{A}^*$ désigne le langage dont le seul mot est w (w peut être ε)
  - $\omega_1\omega_2$ désigne le langage dont chaque mot est obtenu en concaténant (juxtaposant) un mot du langage désigné par ω_1 avec un mot du langage désigné par ω_2 (dans l'ordre).
  - ω^* désigne le langage dont chaque mot est obtenu en concaténant un nombre quelconque (éventuellement nul) de mots du langage désigné par ω.
  - $\omega_1 \mid \omega_2$ désigne la réunion des langages désignés par ω_1 et ω_2. Les mots de $\omega_1 \mid \omega_2$ sont ceux qui appartiennent à ω_1 *ou* à ω_2.
- Et pour des raisons pratiques, on se donne quelques notations complémentaires :
  - $\omega^+ = \omega\omega^* = \omega^*\omega$;
  - $\omega?$ signifie 0 ou 1 instance de ω. Donc $\omega?$ est une abréviation pour $\omega \mid \varepsilon$;
  - la notation $[a, b, c]$, où a, b et c sont des symboles, abrège l'expression régulière $a \mid b \mid c$. Une classe de caractères comme $[a - z]$ abrège l'expression rationnelle $a \mid b \mid ... \mid z$.
  - ω^n, pour un $n \geq 0$ désigne la chaîne $\omega \overset{n \text{ fois}}{\ldots} \omega$.

Par exemple, 01 désigne le langage $\{01\}$, 0^* désigne le langage $\{0\}^*$ et $(0+1)^*011$ désigne le langage ne comportant que des chaînes se terminant par 011 sur l'alphabet $\{0, 1\}$. Comme exemple plus complexe, on peut donner $[A-Z, a-z][A-Z, a-z, 0-9]^*$ qui décrit la syntaxe des identificateurs : des mot commençant par une lettre, majuscule ou minuscule ($[A - Z, a - z]$) et suivit d'une suite quelconque de caractères parmi les lettres et les chiffres ($[A - Z, a - z, 0 - 9]^*$). Notons enfin que si ω^n désigne bien un langage régulier quand n est un entier fixé, par contre, ce n'est pas toujours vrai dans le cas général, quand n est une variable. En effet, par exemple, $(ab)^3$ désigne $ababab$ qui est une expression régulière, de même que $[a, b]^2$ qui désigne $(a \mid b)(a \mid b)$. Par contre le langage désigné par a^nb^n n'est pas régulier : il s'agit des mots commençant par un certain nombre de a

2. Certains auteurs utilisent parfois "+" plutôt que "|"

et suivis *du même* nombre de b, et ce langage ne peut pas être obtenu à partir des parties finies d'un alphabet $\mathcal{A}$ par simple application des opérations $\oplus, \cup, *$. Ceci illustre bien le fait que tout langage sur $\mathcal{A}$ n'est pas nécessairement régulier.

Les langages rationnels sont exactement les langages reconnaissables par les automates finis (qui sont présentés plus loin aux chapitres 4 et 5) et donc, pour montrer qu'un langage est régulier, la méthode la plus courante consiste à produire soit une expression rationnelle qui lui correspond (en vertu du théorème 2.2.2), soit un automate fini qui le reconnaît (en vertu du théorème 5.4.1, chapitre 5).

De manière générale, par contre, il n'est pas toujours facile de montrer qu'un langage n'est pas rationnel. Pour cela, on utilise souvent le théorème suivant :

Théorème 2.2.3 *(Lemme d'itération, ou lemme de la pompe) Soit $\mathcal{L}$ un langage rationnel sur un alphabet $\mathcal{A}$. Alors il existe un entier naturel n tel que pour tout mot z de $\mathcal{L}$ vérifiant $|z| > n$, il existe $u, v, w \in \mathcal{A}^*$ tels que $z = uvw$, $v \neq \varepsilon$, $|uv| \leq n$ et pour tout $i \in \mathbb{N}, uv^iw \in \mathcal{L}$.*

Ce théorème permet souvent de démontrer par l'absurde qu'un langage $\mathcal{L}$ n'est pas rationnel : il suffit de montrer que pour tout n de $\mathbb{N}$ il existe un mot z de $\mathcal{L}$ qui ne vérifie pas la condition. Si l'on reprend l'exemple de $\{a^m b^m \mid m \in \mathbb{N}\}$, la démonstration prend la forme suivante :

HYPOTHÈSE :

Il existe un entier positif tel que
$\forall z \in \{a^m b^m \mid m \in \mathbb{N}\}$ plus long que cet entier, $\exists u, v, w \in \mathcal{A}^*$ tels que :

H1 : $z = uvw$, **H2 :** $v \neq \varepsilon$, **H3 :** $|uv| \leq n$, **H4 :** $\forall i \in \mathbb{N},\ uv^iw \in \{a^m b^m \mid m \in \mathbb{N}\}$.

Soit n cet entier ($|z| > n$).

LE MOT QUI VA POSER PROBLÈME :

Considérons le mot $z = a^n b^n$. Soient u, v et w tels $z = uvw$ (**H1**) et vérifiant **H2**, **H3** et **H4** (z est plus long que n et par hypothèse, u, v et w existent).

LA CONTRADICTION :

Tout préfixe de z de longueur $l \leq n$ est nécessairement composé uniquement de a. Comme uv est un préfixe de z (**H1 :** $z = uvw$) et $|uv| < n$ (**H3**), ceci est vrai en particulier de $|uv|$.

Donc il existe $k, p \in \mathbb{N}$ ($p \neq 0$) tels que $u = a^k$, $v = a^p$, $p \neq 0$ (**H2 :** $v \neq \varepsilon$) et $k + p \leq n$.
Et on a $z = uvw = a^k a^p w$, avec $w = a^{n-(k+p)} b^n$ (car $z = a^n b^n$).

H4 s'écrit alors $\forall i \in \mathbb{N},\ a^k (a^p)^i a^{n-(k+p)} b^n \in \{a^m b^m \mid m \in \mathbb{N}\}$, or ceci est faux :

$$a^k (a^p)^i a^{n-(k+p)} b^n = a^k a^p (a^p)^{i-1} a^{n-(k+p)} b^n = a^k a^p a^{n-(k+p)} (a^p)^{i-1} b^n = a^n (a^p)^{i-1} b^n$$

et, comme $p \neq 0$, pour n'importe quel $i > 1$, $a^n (a^p)^{i-1} b^n \notin \mathcal{L}$.

CONCLUSION :

Donc l'hypothèse est fausse.
Donc, par le théorème 2.2.3, $\{a^m b^m \mid m \in \mathbb{N}\}$ n'est pas un langage rationnel.

Remarquons que ce lemme ne peut pas servir à démontrer qu'un langage est rationnel : il existe en effet des langages non rationnels qui le vérifient (par exemple $\mathcal{L} = \{a^m b^p c^p | m, p \in \mathbb{N} \text{ } et \text{ } m > p\}$:

il suffit de considérer $n = 2$, car tout mot de $\mathcal{L}$ plus long que 2 commence nécessairement par 2 a et l'on peut itérer à volonté le deuxième a, par exemple, sans sortir du langage $\mathcal{L}$).

2.2.6 Manipulation des expressions rationnelles

Cette section donne un complément sur les expressions rationnelles. Nous présentons tout d'abord une méthode pratique pour déclarer des langages rationnels de façon plus lisibles que sous forme d'expressions compliquées. Nous définissons ensuite des équations entre expressions rationnelles et montrons comment résoudre des systèmes d'équations de ce type.

Définition rationnelle

Pour des raisons de commodité de notation, il est pratique donner des noms aux expressions régulières et de définir ensuite de nouvelles expressions régulières en utilisant ces noms comme s'ils étaient des symboles. La seule précaution à prendre est d'éviter les confusions en choisissant judicieusement les noms que l'on associe aux expressions rationnelles. Dans la définition qui suit, qui introduit la notion de définition rationnelle, nous utilisons pour cela les caractères gras.

Définition 2.2.13 *(Définition rationnelle) Si $\mathcal{A}$ est un alphabet de symboles de base, une définition rationnelle est une suite de n définitions ($n \in \mathbb{N}$) de la forme $\mathbf{d_i} \to r_i$ (pour $i \in \{1, \cdots, n\}$), où chaque $\mathbf{d_i}$ est un nom distinct, n'appartenant pas à $(\mathcal{A} \cup \{|, *, \varepsilon, (,)\})^*$, et chaque r_i une expression régulière sur l'alphabet augmenté $\mathcal{A} \cup \{\mathbf{d_1}, ..., \mathbf{d_{i-1}}\}$.*

Exemple : les identificateurs sont définis par :

- $\mathbf{lettre} \to A \mid B \mid ... \mid Z \mid a \mid b \mid ... \mid z$;
- $\mathbf{chiffre} \to 0 \mid 1 \mid ... \mid 9$;
- $\mathbf{id} \to \mathbf{lettre}(\mathbf{lettre} \mid \mathbf{chiffre})^*$.

On remarque que chaque membre gauche (r_i) d'une règle ($d_i \to r_i$) ne peut faire intervenir que les nouveaux symboles définis antérieurement (les d_j tels que $j < i$) et donc on n'a pas, dans le cas général, de définition récursive, comme le serait par exemple la définition caractérisée par l'unique règle $\mathbf{X} \to a\mathbf{X}b|ab$, qui engendre pour $\mathbf{X}$ le langage $\{a^n b^n \mid n \in \mathbb{N}\}$. En effet, si l'on autorisait l'écriture de définitions récursives, on pourrait caractériser des langages qui ne sont pas rationnels (définition 2.2.11), comme par exemple $\{a^n b^n \mid n \in \mathbb{N}\}$, ci-dessus. Ces définitions récursives seront d'ailleurs utilisées pour construire d'autres classes de langages, à l'aide de "grammaires", comme nous le verrons plus loin. Par contre, il est possible d'autorisé une forme de récursivité : la récursivité droite, c'est-à-dire des définitions de la forme $\mathbf{X} \to x\mathbf{X}|y$ avec x et y des expressions régulières. Par exemple, la définition suivant est autorisée : $\mathbf{X} \to a\mathbf{X}|a$. Elle engendre pour $\mathbf{X}$ le langage $\{a^n \mid n \in \mathbb{N}\}$. Donc $\mathbf{X} \to a\mathbf{X}|a$ est équivalente à $\mathbf{X} \to a^+$.

Systèmes d'équations rationnelles

Nous avons donc vu que les expressions rationnelles décrivaient des langages, c'est-à-dire des ensembles de mots sur un alphabet $\mathcal{A}$ donné (souvent implicitement). De fait, il s'avère que deux expressions différentes peuvent décrire un même langage. Par exemple, a^*a et aa^* désignent toutes deux l'ensemble des chaînes de n a consécutifs pour $n > 0$. On peut par ce biais définir une équivalence entre expressions rationnelles.

Définition 2.2.14 *Deux expressions rationnelles ω_1 et ω_2 sont dites égales, noté "$\omega_1 = \omega_2$", si elles décrivent le même langage rationnel .*

Soient, α, β et γ des expressions rationnelles, on a entre autres les propriétés suivantes :

- $\alpha \mid \beta = \beta \mid \alpha$
- $\alpha^* = \alpha \mid \alpha^*$
- $\alpha(\gamma\beta) = (\alpha\gamma)\beta$
- $\alpha^{*^*} = \alpha^*$
- $\alpha \mid \alpha = \alpha$
- $(\alpha \mid \gamma)\beta = \alpha\beta \mid \gamma\beta$
- $\alpha\varepsilon = \varepsilon\alpha = \alpha$
- $\emptyset^* = \varepsilon$
- $\alpha \mid (\gamma \mid \beta) = (\alpha \mid \gamma) \mid \beta$
- $\alpha\emptyset = \emptyset\alpha = \emptyset$
- $\alpha \mid \emptyset = \alpha$
- $\alpha(\gamma \mid \beta) = \alpha\gamma \mid \alpha\beta$

Les définitions régulières permettent de proposer des expressions régulières plus lisibles. Cependant, il est parfois nécessaire d'obtenir une expression régulière ne comportant que des symboles du vocabulaire. Il faut donc alors essayer de retrouver cette expression. La simple substitution n'est pas toujours suffisante, surtout en cas de récursivité. La solution consiste à passer par la résolution d'un système d'équations régulières. La transformation s'effectue en utilisant directement les définitions régulières. L'opérateur $\rightarrow$ devient $=$, l'opérateur $\mid$ devient $+$ et les identificateurs deviennent des variables. La résolution du système ainsi obtenu donne l'expression régulière du langage. Nous allons donc décrire ces systèmes d'équations ainsi que la méthode de résolution.

Définition 2.2.15 *Les systèmes d'équations régulières sont des ensembles d'équations dont les coefficients sont des expressions rationnelles.*

Par exemple, le système suivant est un système d'équations rationnelles :

$$\left\{\begin{array}{l} X = a_1X + a_2Y + a_3 \\ Y = b_1X + b_2Y + b_3 \end{array}\right. \quad \text{avec } a_i,\ b_i\ (i = 1, 2, 3) \text{ des expressions rationnelles.}$$

Alors une solution est :

$$\left\{\begin{array}{l} X = (a_1 + a_2b_2^*b_1)^*(a_3 + a_2b_2^*b_3) \\ Y = (b_2 + b_1a_1^*a_2)(b_3 + b_1a_1^*a_3) \end{array}\right.$$

Remarque : les solutions ne sont pas uniques, mais, en général, on s'intéresse à la plus petite des solutions (le plus petit point fixe).

Définition 2.2.16 *Un ensemble d'équations rationnelles d'indéterminés $\delta = \{X_1, ..., X_n\}$ est en forme standard si $\forall X_i \in \delta$, il y a une équation de la forme $X_i = \alpha_{i0} + \alpha_{i1}X_1 + ... + \alpha_{in}X_n$ avec α_{ij} des expressions rationnelles sur $\mathcal{A}$ telles que $\mathcal{A} \cap \delta = \emptyset$.*

Remarques : il est possible d'avoir $\alpha_{ij} = \emptyset$ quand il n'y a pas de terme correspondant à X_j dans l'équation de X_i. $\emptyset$ joue le rôle de 0 et ε celui de 1 (l'équation pour X_i comporte alors X_j sans coefficient à droite).

Pour résoudre un tel système, il suffit, par analogie avec les systèmes linéaires, de procéder par élimination de variables (figure 2.1).

étape 1 Pour i de 1 à $n - 1$ faire (en utilisant les propriétés des expressions rationnelles) :

1. écrire l'équation du système sous la forme $X_i = \alpha X_i + \beta$ où α est une expression rationnelle sur $\mathcal{A}$ et β est une expression rationnelle de la forme $\beta_0 + \beta_1 X_{i+1} + ... + \beta_n X_n$ avec β_i des expressions rationnelles sur $\mathcal{A}$.

```
RÉSOURDRESYSTÈME(↘ Q, ↘ δ = {X_1, ..., X_n}, ↗ S = {X_i = α_i})
/Q : Syst. d'équ. rég. en forme standard/
/δ : Ensemble de variables/
/S : Ensemble de solutions/
/ Etape 1 /
Pour i ← 1 à n − 1   Faire
    Ecrire l'équ. du syst. sous la forme X_i = α_1 X_i + β
    Pour j ← i + 1 à n   Faire
        Remplacer X_i par α*β dans l'équ. X_j = ...
    Fin Pour
Fin Pour
/ Etape 2 /
S ← ∅
Pour i ← n à 1   Faire
    S ← S ∪ {X_i = α*β}
    Pour j ← 1 à i − 1   Faire
        Poser X_i = α*β dans X_j = ...
    Fin Pour
Fin Pour
```

FIG. 2.1 – *Algorithme de résolution d'un système d'équations régulières*

2. Sachant que l'on peut montrer que si $X = \alpha X + \beta$ avec α et β des expressions rationnelles alors on peut écrire $X = \alpha^* \beta$ [3], il suffit de remplacer X_i par $\alpha^*\beta$ dans toutes les équations $X_{i+1}, ..., X_n$ [4].

étape 2 Pour i de n à 1 faire (sachant que l'équation pour X_i est de la forme $X_i = \alpha_i X_i + \beta$ avec α et β des expressions rationnelles sur $\mathcal{A}$) poser $X_i = \alpha^*\beta$ et substituer $\alpha^*\beta$ dans toutes les équations de 1 à $(i-1)$.

Par exemple, Soit $\delta = \{X_1, X_2, X_3\}$ avec $\mathcal{A} = \{0, 1\}$ et le système S :

$$S = \begin{cases} X_1 = 0X_2 + 1X_1 + \varepsilon \\ X_2 = 0X_3 + 1X_2 \\ X_3 = 0X_1 + 1X_3 \end{cases}$$

- D'abord, on effectue l'étape 1. On obtient $X_1 = 1X_1 + (\varepsilon + 0X_2) = 1^*(\varepsilon + 0X_2)$. D'où, $X_3 = 1X_3 + (01^* + 01^*0X_2)$. Ensuite, $X_2 = 1X_2 + 0X_3 = 1^*0X_3$. D'où, $X_3 = (1 + 01^*01^*0)X_3 + 01^* = (1 + 01^*01^*0)^*01^*$.
- Ensuite, on passe à l'étape 2. On obtient, $X_2 = 1X_2 + 0(1 + 01^*01^*0)^*01^* = 1^*0(1 + 01^*01^*0)^*01^*$. Et finalement, $X_1 = 1X_1 + 0X_2 + \varepsilon = 1X_1 + 01^*0(1 + 01^*01^*0)^*01^* + \varepsilon = 1^*01^*0(1 + 01^*01^*0)^*01^* + 1^*$.

3. Par récurrence $X_0 = \emptyset, X_1 = c, X_n = ((ab)^n + (ab)^{n-1} + ... + \varepsilon)c$, avec $X_\infty = \lim_{i\to\infty} X_i = (ab)^*c$ et en montrant que $(ab)^*c = ab(ab)^*c + c$.

4. L'équation pour X_i ne comporte que des symboles de $\mathcal{A}$ et $X_{i+1}, ..., X_n$ dans le membre de droite.

2.3 Les grammaires

Un langage est défini par l'ensemble de ses mots, ou, vu autrement, de ses phrases correctes (ce que nous voyions jusque là comme les lettres d'un mot peut alors être vu comme les mots d'une phrase). Il est impossible de donner la liste des phrases (ou mots) correctes d'un langage parce qu'elle est souvent infinie. Pour les langages rationnels que nous venons de voir, nous pouvons les dénoter par des expressions régulières. Toutefois, tous les langages ne sont pas rationnels et cette solution n'est donc pas toujours possible. Pour atteindre d'autres classes de langage, on peut utiliser divers types de *grammaires*, ce que nous présentons dans cette section.

2.3.1 Structure d'une grammaire

Une grammaire à structure de phrase G permet donc de caractériser un langage $L(G)$, qui est un ensemble de phrases (ou mots), appelé le langage engendré par la grammaire. Une telle grammaire ressemble à une définition rationnelle, en plus riche, plus expressif. Elle repose sur :

1. La connaissance d'un vocabulaire terminal noté V_T et dont chaque élément est appelé symbole terminal. Ce vocabulaire correspond à l'alphabet pour les langages réguliers et les symboles terminaux aux lettres. Nous avons donc $L(G) \subseteq V_T^*$.

   Dans ce cours, nous ne considérerons que les vocabulaires terminaux composés de symboles ou de caractères (lettres, chiffres, caractères spéciaux comme '+', '-', ';'...). En compilation, ce vocabulaire est un ensemble d'unités lexicales (identificateurs réservé ou non, paramètres, nom, opérateurs...).

2. La connaissance d'un vocabulaire non terminal noté V_N et dont chaque élément est appelé symbole non terminal. Les symboles non terminaux peuvent être vus comme les nouveaux symboles des définitions régulières (utilisés comme membres gauches des règles). Ils ont toutefois un rôle un peu plus large. Ils n'appartiennent pas au vocabulaire de base (terminaux) et sont destinés à être éliminés, remplacés pour former les mots du langage.

3. Des règles permettant de déterminer quelles séquences de V_T^* sont légales. Ces règles ressemblent à celles des définitions rationnelles mais sont plus souples, plus générales.

Plus rigoureusement, on a la définition suivante :

Définition 2.3.1 *(Grammaire) Une grammaire formelle, appelée aussi grammaire de Chomsky, G est une description de la forme des symboles et des phrases d'un langage noté $L(G)$. Elle est définie par un quadruplet $G = (V_T, V_N, S, R)$ où :*

- *V_T est un ensemble fini non vide : le vocabulaire terminal ;*
- *V_N est un ensemble fini non vide : le vocabulaire non terminal ;*
  On a $V_T \cap V_N = \emptyset$ et on note $V = V_T \cup V_N$ le vocabulaire de la grammaire ;
- *$S \in V_N$ est un symbole non terminal particulier appelé l'*axiome *;*
- *R est un ensemble de règles, appelées règles de production (ou règles de réécriture), de la forme $\alpha \overset{G}{\rightarrow} \beta$ avec $\alpha \in V^+$ et $\beta \in V^*$.*

Les règles sont notées $\alpha \overset{G}{\rightarrow} \beta$ pour indiquer explicitement que la règle fait partie de la grammaire G ou simplement $\alpha \rightarrow \beta$ s'il n'y a pas d'ambigüité. Une règle de production $\alpha \rightarrow \beta$ se lit "α peut être remplacé par β". α est appelé "partie gauche" et β "partie droite" de la règle. On peut également écrire des règles de la forme $\alpha \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$ avec $n \geq 1$ et pour tout $1 \leq i \leq n$, $\beta_i \in V^*$ pour abréger n règles ayant α en partie gauche, de la forme $\alpha \rightarrow \beta_i$.
Par exemple, soit la grammaire $G_1 = (\{a, b\}, \{S, X\}, S, R)$ avec R :

- R1 : $S \rightarrow aXb$;
- R2 : $aX \rightarrow aaXb$;
- R3 : $X \rightarrow \varepsilon$.

Intuitivement, cette grammaire définit le langage $L(G_1) = \{a^n b^n \mid n \geq 1\}$ sur le vocabulaire terminal (ou alphabet) $\mathcal{A} = \{a, b\}$.
En effet, de façon informelle :

1. Appliquer une règle va consister à remplacer dans un mot une occurrence du membre gauche de la règle par le membre droit correspondant. Sur l'exemple, à partir de $\underline{aA}b$, on peut obtenir $\underline{aaXb}b$ en appliquant R2 (le lieu du remplacement étant signalé par les lettres soulignées), ce que l'on peut noter $aXb \overset{R2}{\Rightarrow} aaXbb$.
2. On considérera l'ensemble des mots que l'on peut atteindre en prenant l'axiome S comme mot de départ et en applicant les règles un nombre fini de fois. Sur l'exemple, on peut ainsi atteindre (entre autres) les mots "aXb" (par $S \overset{R1}{\Rightarrow} aXb$), "$aaXbb$" (par $S \overset{R1}{\Rightarrow} aXb \overset{R2}{\Rightarrow} aaXbb$) et "$ab$" (par $S \overset{R1}{\Rightarrow} aXb \overset{R3}{\Rightarrow} ab$); remarquons que l'on peut également atteindre S, par l'application successive de 0 règles. Les mots ainsi accessibles sont appelés "formes sententielles" et les suites d'applications de règles, "dérivations".
3. Les mots du langage caractérisé par la grammaire sont les mots accessibles (au sens du point précédent) qui ne sont composés que de symboles terminaux. Sur l'exemple, parmi les mots de $L(G)$, il y a ab ($S \overset{R1}{\Rightarrow} aXb \overset{R3}{\Rightarrow} ab$), mais aussi $aabb$ ($S \overset{R1}{\Rightarrow} aXb \overset{R2}{\Rightarrow} aaXbb \overset{R3}{\Rightarrow} aabb$), et $aaabbb$ ...

On démontrera rigoureusement plus loin que $L(G_1)$ est exactement $\{a^n b^n \mid n \geq 1\}$. Auparavant, nous donnons les définitions rigoureuses correspondant aux idées que nous venons de présenter intuitivement.

Définition 2.3.2 *(Formes sententielles et langage d'une grammaire)*
Soit $G = (V_T, V_N, S, R)$ une grammaire et $V = V_T \cup V_N$, alors :

- *L'ensemble des formes sententielles (ou proto-phrases) de G est défini récursivement par :*
  1. *L'axiome S est une* forme sententielle *de G.*
  2. *Pour tous $\alpha, \beta, \gamma, \delta \in V^*$, si $\alpha\beta\gamma$ est une forme sententielle de G et si $\beta \rightarrow \delta \in R$, alors $\alpha\delta\gamma$ est une forme sententielle de G.*
- *Le* langage engendré *par G est*
  $L(G) = \{\alpha \in V^* \mid \alpha$ *est une forme sententielle de G et* $\alpha \in V_T^*\}$.

Les formes sententielles ne contenant aucun symbole non terminal sont donc les phrases de $L(G)$. Par ailleurs, la définition par induction des formes sententielles est la base de la notion d'application de règles que nous avons évoquée ci-dessus. Nous précisons ce point dans la section suivante.

2.3.2 Génération de chaînes, fonctionnement et dérivation

Une grammaire G est peut être utilisée soit pour générer les phrases du langage $L(G)$ soit pour déterminer si une phrase donnée appartient ou non au langage $L(G)$:

- La génération consiste à produire des chaînes de symboles par application des règles de la grammaire à partir de son axiome S.
- La vérification (l'analyse) consiste à retrouver la succession d'applications de règles qui a permis d'obtenir un mot en partant de l'axiome S.

On appelle *application de règle*, une dérivation directe. On étend cette notion à des suites finies d'applications de règles par les définitions qui suivent.

Définition 2.3.3 *(Dérivations) Soit $G = (V_T, V_N, S, R)$ une grammaire.*

- *On définit la relation "dérive directement de" sur V^*, notée $\overset{G}{\Rightarrow}$, par :*

$$\forall z, z' \in V^*, \quad z \overset{G}{\Rightarrow} z' \quad ssi \quad \exists \alpha, \beta, \gamma, \delta \in V^*,\ z = \alpha\beta\gamma \ \text{ et } \ z' = \alpha\delta\gamma \ \text{ et } \ \beta \to \delta \in R$$

  $z \overset{G}{\Rightarrow} z'$ est appelé une dérivation directe *de z à partir de z'.*
  De plus, si $\alpha \in V_T^*$ (resp. $\gamma \in V_T^*$) alors $\alpha\beta\gamma \overset{G}{\Rightarrow} \alpha\delta\gamma$ est une dérivation à gauche (resp. à droite)[5].

- *On appelle* dérivation *de longueur n ($n \in \mathbb{N}$) toute suite $s = \alpha_0, \cdots, \alpha_n$ de chaînes de V^* telle que $\alpha_0 \overset{G}{\Rightarrow} \alpha_2$ et $\cdots$ et $\alpha_{n-1} \overset{G}{\Rightarrow} \alpha_n$ (ce que l'on abrège par $\alpha_0 \overset{G}{\Rightarrow} \cdots \overset{G}{\Rightarrow} \alpha_n$)*

Notations et terminologie :
On écrit $\Rightarrow$ pour $\overset{G}{\Rightarrow}$ quand il n'y a pas d'ambigüité sur la grammaire et $\overset{R}{\Rightarrow}$ pour désigner explicitement la règle utilisée. On adopte en outre les conventions suivantes pour les dérivations, où α est la première chaîne de la dérivation et β la dernière :

- *$\alpha \overset{n}{\Rightarrow} \beta$ désigne une dérivation de longueur n.*
- *$\alpha \overset{*}{\Rightarrow} \beta$ désigne une dérivation de longueur quelconque. Autrement dit, $\overset{*}{\Rightarrow}$ est la fermeture réflexive et transitive de $\overset{G}{\Rightarrow}$.*
- *$\alpha \overset{+}{\Rightarrow} \beta$ désigne une dérivation de longueur supérieure à 1. Autrement dit, $\overset{+}{\Rightarrow}$ est la fermeture transitive de $\overset{G}{\Rightarrow}$.*
- *Si $\alpha \overset{*}{\Rightarrow} \beta$, on dit alors que α* engendre *β ou, symétriquement, que β* dérive de *α.*

  Si de plus $\alpha \overset{+}{\Rightarrow} \beta$, on dit "engendre" (resp. "dérive") "de façon non triviale".

En conséquence de ces définitions, les formes sententielles d'une grammaire sont exactement les chaînes que l'on peut dériver à partir de son axiome S. On peut donc proposer une nouvelle définition d'une langage engendré par une grammaire, équivalente à la précédente.

5. Intuitivement, on remplace dans $\alpha\beta\gamma$ le non terminal β (resp. γ) le plus à gauche (resp. à droite) pour obtenir $\alpha\delta\gamma$.

Théorème 2.3.1 *(Langage - version 2 -) Soit $G = (V_T, V_N, S, R)$ une grammaire :*

- *Une chaîne α est une forme sententielle pour la grammaire G si et seulement si $S \stackrel{*}{\Rightarrow} \alpha$.*
- *Le langage* engendré *par G est l'ensemble des chaînes de symboles terminaux que l'on peut dériver à partir de l'axiome S : $L(G) = \{x \in V_T^* | S \stackrel{+}{\Rightarrow} x\}$.*

Cette définition n'est pas très différente de la précédente, mais elle permet de manipuler explicitement les dérivations, ce qui est pratique pour certaines démonstrations.

Si l'on revient sur l'exemple précédent (p. 21), on peut maintenant montrer que le langage de la grammaire G_1 coïncide exactement avec $\{a^n b^n \mid n \geq 1\}$. Il faut pour cela montrer une double inclusion : $\{a^n b^n \mid n \geq 1\} \subseteq L(G_1)$ et $L(G_1) \subseteq \{a^n b^n \mid n \geq 1\}$.

$$\{a^n b^n \mid n \geq 1\} \subseteq L(G_1)$$

Lemme : $\forall n \in \mathbb{N}$, *si* $S \stackrel{*}{\Rightarrow} a^n X b^n$ *alors* $S \stackrel{*}{\Rightarrow} a^n b^n$.
Démo : Ayant une dérivation $S \stackrel{*}{\Rightarrow} a^n X b^n$, on peut construire la dérivation $S \stackrel{*}{\Rightarrow} a^n X b^n \stackrel{3}{\Rightarrow} a^n b^n$

Pour tout n, $a^n b^n \in V_T^*$. Il reste à montrer que $\forall n > 1 \in \mathbb{N}$, $S \stackrel{*}{\Rightarrow} a^n b^n$. Par le lemme, il suffit de montrer que $\forall n > 1 \in \mathbb{N}$, $S \stackrel{*}{\Rightarrow} a^n X b^n$, par récurrence sur n :

- $n = 1$. On a la dérivation $S \stackrel{R1}{\Rightarrow} aXb$
- $n = m + 1$, $m \geq 1$, et on a $S \stackrel{*}{\Rightarrow} a^{m-1} a X b^m$ (HR : hypothèse de récurrence).
  On peut donc construire une dérivation de $a^n b^n$ à partir de S :
  $S \stackrel{*}{\Rightarrow} a^{m-1} \underline{aX} b^m \stackrel{R2}{\Rightarrow} a^{m-1} \underline{aaXb} b^m$.

$$L(G_1) \subseteq \{a^n b^n \mid n \geq 1\}$$

Lemme : $\forall n \in \mathbb{N},\ n \geq 1,\ \forall z \in \{a, b\}^*$, *si* $S \stackrel{n}{\Rightarrow} z$ *alors* $z = a^n X b^n$ *ou* $a^{n-1} b^{n-1}$
Démo : par induction sur la longueur des dérivations,

- $n = 1$. Une seule dérivation possible $S \stackrel{R1}{\Rightarrow} aXb$ donc $z = a^1 X b^1$.
- $n = m + 1$, et on a *si* $S \stackrel{m}{\Rightarrow} w$ *alors* $w = a^m X b^m$ *ou* $w = a^{m-1} b^{m-1}$ (HR).
  Donc si $S \stackrel{m}{\Rightarrow} w \stackrel{R}{\Rightarrow} z$ alors au vu des règles de G_1, $w = a^m X b^m$ et
  - soit $R = R2$, $w = a^{m-1} \underline{aX} b^m$ *et* $z = a^{m-1} \underline{aaXb} b^m = a^n X b^n$
  - soit $R = R3$, $w = a^m \underline{X} b^m$ *et* $z = a^m \underline{\varepsilon} b^m = a^{n-1} b^{n-1}$

Donc $\forall z \in \{a, b\}^*$, *si* $S \stackrel{+}{\Rightarrow} z$ *et* $z \in V_T^*$ *alors* $\exists n \in \mathbb{N},\ n \geq 1,\ z = a^n b^n$.
D'où par définition $L(G_1) \subseteq \{a^n b^n \mid n \geq 1\}$

On a donc établi et illustré la façon dont une grammaire définit un langage. Il peut arriver qu'un même langage L soit atteint par deux grammaires différentes.

Définition 2.3.4 *Deux grammaires G_1 et G_2 sont dites équivalentes si $L(G_1) = L(G_2)$.*

Par exemple, soit la grammaire $G_2 = (\{a,b\},\{S,X\},S,R)$ avec R :

- R1 : $S \to aXb$;
- R2 : $X \to aXb$;
- R3 : $X \to \varepsilon$.

On pourrait montrer que cette grammaire définit le même langage que G_1, c'est-à-dire $L(G_2) = \{a^n b^n \mid n \geq 1\} = L(G_1)$, par une démonstration du même type que la précédente. Autrement dit, G_1 et G_2 sont équivalentes.

2.3.3 Classification des grammaires

De même que les expressions régulières permettent d'atteindre la classe des langages rationnels (ou réguliers), différents types de grammaires permettent d'atteindre différentes classes de langages. La classification de N. Chomsky [Cho56] se fonde sur des contraintes concernant la forme des règles de dérivation. Selon l'ordre croissant des contraintes, nous avons la classification suivante :

Type 0 *grammaire sans contraintes* : règles de la forme $\alpha \to \beta$ avec α et $\beta \in V^*$.

Type 1 *grammaire contextuelle* (*context sensitive*, sensible au contexte) : règles de la forme $\alpha \to \beta$ avec α et $\beta \in V^*$ et $|\alpha| \leq |\beta|$. Une autre définition équivalente existe : une grammaire de type 1 possède des règles de la forme : $\alpha A \beta \to \alpha\gamma\beta$ avec $\alpha, \beta \in V^*$, $A \in V_N$ et $\gamma \in V^+$ tel que $|\gamma| \geq 1$ [6] . Habituellement, on introduit une exception à ces règles : l'axiome, s'il n'est pas en partie droite d'une autre règle que celle qui le définit, peut engendrer ε.

Type 2 *grammaire hors-contexte (context free ou algébrique)* : règles de la forme $A \to \alpha$ avec $A \in V_N$ et $\alpha \in V^*$.

Type 3 *grammaire régulière (linéaire, d'états finis) à droite (resp. à gauche)* : uniquement des règles de la forme : $A \to x \mid xB$ (resp. $A \to x \mid Bx$) avec $A, B \in V_N$ et $x \in V_T^*$.

Remarque : les grammaires dites «à structure de phrase» sont des grammaires de type 1 telles que les règles $\alpha \overset{G}{\to} \beta$, $\alpha \in V_N^+$.

Par exemple, une grammaire avec comme règles $S \to 0S|1S|\varepsilon$ est régulière droite. Les définitions rationnelles (paragraphe 2.2.5, Déf 2.2.13) peuvent être considérées comme des grammaires hors-contexte particulières (si d_1 est l'axiome et si $\forall i, i > 1, d_i \nrightarrow \varepsilon$).

La classification de Chomsky introduit quatre classes de langage engendrées par les quatre types de grammaire. Ces classes de langage satisfont la relation d'inclusion stricte suivante :

$$\boxed{\text{cl. 3 (régulier)} \subset \text{cl. 2 (hors-contexte)} \subset \text{cl. 1 (sous-contexte)} \subset \text{cl. 0 (sans contrainte)}}$$

Inclusions entre les quatres classes de langages associées aux quatres types de grammaires

Cette relation exprime entre autres que tout langage régulier est hors-contexte et qu'il existe des langages hors-contextes qui ne sont pas réguliers. On a le même type de relation entre les autres classes de langages. Les langages hors-contextes et réguliers sont les plus courants en informatique,

6. c'est-à-dire que A peut être remplacé par γ dans le contexte $\alpha A\beta$

en particulier dans le cadre des langages de programmation; les algorithmes de traitement pour ces deux classes sont les plus efficaces et les plus simples. Dans le cadre de ce cours, nous étudierons plus particulièrement le traitement des langages réguliers (en particulier à l'aide des automates finis). Cependant nous présentons avant quelques outils utiles sur les grammaires hors contexte.

2.3.4 Représentations pour les grammaires hors contexte

A propos des grammaires hors contextes, il existe diverses présentations standard pratiques aussi bien pour les règles des grammaires que pour les dérivations qu'elles permettent. Nous donnons dans cette section deux normes de présentation des règles, puis la représentation usuelle des dérivations sous forme d'arbre.

Représentations des règles de production

Il existe donc plusieurs manières de représenter les règles d'une grammaire.

Textuellement, la norme BNF ("Backus-Naur-Form") est la plus utilisée. Elle est particulièrement utile pour analyser un langage défini par une grammaire hors-contexte (et donc aussi pour une grammaire régulière). Les symboles non terminaux sont encadrés par des guillemets ("..." ou «...»), les symboles terminaux ne sont pas distingués. L'axiome est la partie gauche de la première règle énoncée. La disjonction est notée par une barre verticale (|). La flèche des règles est remplacée par le signe "::=". Par exemple, la figure 2.2 illustre la forme BNF pour les règles $\{S \to bA; A \to aA; A \to a\}$.

```
"S" ::= b"A"
"A" ::= a"A" | a
```

FIG. 2.2 – *Écriture BNF pour les règles* $\{S \to bA; A \to aA; A \to a\}$

Il existe aussi une représentation graphique des règles en utilisant les diagrammes de Conway. Elle décrit graphiquement les grammaires hors-contexte. Il permet de regrouper des règles et de mettre en évidence les phénomènes récursifs. La partie gauche des règles est notée à gauche du schéma. Les symboles terminaux sont écrits dans des cercles et les symboles non terminaux sont écrits dans des rectangles. Une chaîne du langage est formée en parcourant les flèches et en recueillant les symboles rencontrés. La figure 2.3 montre les diagrammes de Conway pour la grammaire précédente.

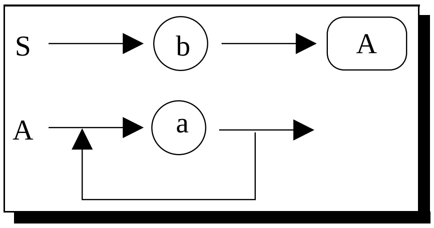
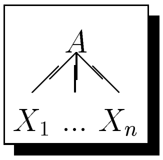


FIG. 2.3 – *Diagrammes de Conway pour les règles* $\{S \to bA; A \to aA; A \to a\}$

Arbres de dérivations

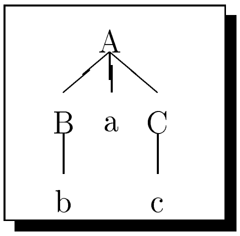
Rappelons que tout mot α du langage engendré par une grammaire G peut être obtenu par une dérivation $S \overset{+}{\Rightarrow} \alpha$ avec α uniquement composé de vocabulaire terminal ($\alpha \in L(G) \subseteq V_T^*$). Une telle dérivation peut être représentée par un arbre de dérivation (appelé aussi “arbre syntaxique” ou “arbre d'analyse”). L'idée consiste à associer à chaque règle R : $A \to X_1 ... X_n$ de la grammaire, où $A \in V_n$ et $\forall 1 \leq i \leq n,\ X_i \in V$, la structure d'arbre suivante :



La correspondance entre arbres et dérivations est alors la suivante :

- Une dérivation directe $A \overset{R}{\Rightarrow} X_1 \cdots X_n$ est représentée par l'arbre ci-dessus.
- Et une dérivation de la forme $A \overset{n}{\Rightarrow} Y_1 \cdots Y_{i-1} Y_i Y_{i+1} \cdots Y_m \overset{R}{\Rightarrow} Y_1 \cdots Y_{i-1} X_1 ... X_m Y_{i+1} \cdots Y_m$ est représentée par l'arbre associé à $A \overset{n}{\Rightarrow} Y_1 \cdots Y_m$, dans lequel on remplace le symbole non terminal correspondant à Y_i par l'arbre associé à R.

Par exemple, en considérant les règles R1 : $A \to BaC$, R2 : $B \to b$ et R3 : $C \to c$, à la dérivation $A \overset{R1}{\Rightarrow} BaC \overset{R2}{\Rightarrow} baC \overset{R3}{\Rightarrow} bac$ correspond l'arbre suivant :



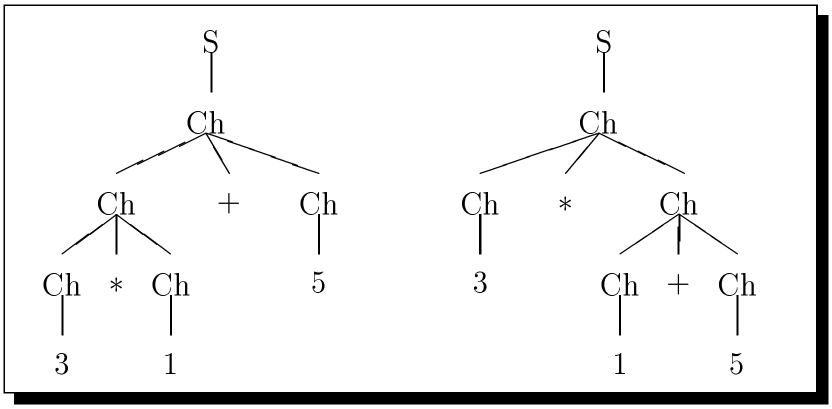
On peut remarquer que la représentation des dérivations sous forme d'arbres ne mémorise pas l'ordre dans lequel les règles sont appliquées, lorsque ces applications n'interfèrent pas. Par exemple, l'arbre ci-dessus correspond également à la dérivation $A \overset{R1}{\Rightarrow} BaC \overset{R3}{\Rightarrow} Bac \overset{R2}{\Rightarrow} bac$

De façon générale, un arbre de dérivation d'un mot du langage d'une grammaire représente une façon de dériver ce mot à partir de l'axiome. La racine de l'arbre est l'axiome, ses feuilles sont les symboles (terminaux) du mots, et ses noeuds internes représentent les applications de règles. Notons que l'arbre de dérivation d'un mot n'est pas toujours unique. Une grammaire est dite ambigüe si au moins un mot dans son langage a plus d'un arbre de dérivation.

Par exemple, soit la grammaire $G = \{\ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *\}, \{S, A\}, R\}$, avec R :

- $S \to Ch$;
- $Ch \to Ch + Ch \mid Ch - Ch$;
- $Ch \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$.

Par cette grammaire, nous pouvons obtenir pour le mot «3∗1+5» les arbres de dérivation suivants :



Notons que ces deux arbres correspondent bien à deux façons différentes d'appliquer les règles qui cette fois, ne sont pas équivalentes. Intuitivement, l'arbre de gauche correspond à l'interprétation $(3 * 1) + 5$ tandis que l'arbre droit correspond à $3 * (1 + 5)$. L'ambigüité dans les grammaires n'est donc pas un problème anodin, mais son étude approfondie sort du cadre de ce cours.

2.4 Cas des langages et expressions régulières

On peut montrer qu'un langage est défini par une grammaire régulière droite si et seulement si c'est un langage rationnel.

Lemme 2.4.1 *(1)$\emptyset$, (2)$\{\varepsilon\}$ et (3)$\{a\}$ avec $a \in \mathcal{A}$ sont des langages linéaires droits.*

Dém. (1) $G = \{\{\}, \{S\}, S, \emptyset\}$ est linéaire droite, $L(G) = \emptyset$;
(2) $G = \{\{\varepsilon\}, \{S\}, S, \{S \to \varepsilon\}\}$ est linéaire droite, $L(G) = \{\varepsilon\}$;
(3) $G = \{\{a\}, \{S\}, S, \{S \to a\}\}$ est linéaire droite, $L(G) = \{a\}$.

Lemme 2.4.2 *Si L_1 et L_2 sont linéaires droits, alors (1) $L_1 \cup L_2$, (2) $L_1 L_2$ et (3) L_1^* sont linéaires droits.*

Dém. On pose $G_1 = \{V_T, V_{N_1}, S_1, R_1\}$ dérivant $L(G_1) = L_1$ et $G_2 = \{V_T, V_{N_2}, S_2, R_2\}$ dérivant $L(G_2) = L_2$. On suppose $V_{N_1} \cap V_{N_2} = \emptyset$ (sinon on les change).

1. Soit $G_3 = \{V_T, V_{N_1} \cup V_{N_2} \cup \{S_3\}, S_3, R_1 \cup R_2 \cup \{S \to S_2 \mid S_1\}\}$ alors $L(G_3) = L_1 \cup L_2$ car $S_3 \overset{+/G_3}{\Rightarrow} \omega$ ssi $S_1 \overset{+/G_1}{\Rightarrow} \omega$ ou $S_2 \overset{+/G_2}{\Rightarrow} \omega$ (car comme $V_{N_1} \cap V_{N_2} = \emptyset$, il n'y a pas de partage entre les différentes grammaires).

2. $G_4 = \{V_T, V_{N_1} \cup V_{N_2} \cup \{S_4\}, S_4, R_4\}$ où :

   - (i) $A \to xB \in R_1$ alors $A \to xB \in R_4$;
   - (ii) $A \to x \in R_1$ alors $A \to xS_2 \in R_4$;
   - (iii) $\forall p \in R_2$, $p \in R_4$.

   On note que toutes les règles de R_4 sont régulières droites. Le langage obtenu est tel que : $L_4 = L_1 L_2$.

3. $G_5 = \{V_T, V_{N_1} \cup \{S_5\}, S_5, R_5\}$ où :
   - (i) $A \to xB \in R_1$ alors $A \to xB \in R_5$;
   - (ii) $A \to x \in R_1$ alors $A \to x$ et $A \to xS_5 \in R_5$;
   - (iii) $S_5 \to S_1 \mid \varepsilon \in R_5$.

   Alors $L(G_5) = L_5 = L_1^*$.

Théorème 2.4.1 *Un langage est rationnel si et seulement s'il est linéaire droit.*

Dém.

1. Si un langage est rationnel alors il est linéaire droit (par récurence sur le nombre d'application des opérations de base).

   (a) Le lemme 2.4.1 nous indique que la relation est vraie pour les langages les plus simples, c'est-à-dire pour $\emptyset$, $\{\varepsilon\}$ et $\{a\}$.

   (b) Supposons la relation vraie après n applications des opérateurs. Soit L le langage obtenu, L est supposé rationnel et linéaire droit. En considérant la définition des langages rationnels et le lemme 2.4.2, on en déduit que $L' = L \cup \emptyset$ ou $L' = L \cup \varepsilon$ ou $L' = L \cup \{x\}$ est rationnel et linéaire droit.

2. Si un langage est linéaire droit alors il est rationnel. Soit $G = \{V_T, V_N, S, R\}$ une grammaire linéaire droite avec $V_N = \{A_1, ..., A_N\}$. On va essayer de construire un système d'équations linéaires associées à G de la façon suivante : les inconnues sont les A_i, l'équation qui correspond à A_i est $A_i = \alpha_{i0} + \alpha_{i1}A_1 + ... + \alpha_{in}A_n$ où :

   (a) $\alpha_{i0} = \omega_1 + \omega_2 + ... + \omega_n$, avec $A_i \to \omega_1 \mid ... \mid \omega_n$ et ω_j, $j = 1, ..., n$ comme membre droit.

   (b) $\alpha_{ij}, j > 0$ est $x_1, x_2, ..., x_n$ où $A_i \to x_1A_j \mid ... \mid x_nA_j$ sont toutes les productions avec A_i à gauche et A_j à droite.

   On obtient donc un système d'équations rationnelles. Si n=0 alors $\alpha_{ij} = \emptyset$. $L(G) = f(S)$ où $f(S)$ est la solution du système s.

Par exemple, Soit G défini par $\{S \to 0A \mid 1S \mid \varepsilon; A \to 0B \mid 1A; B \to 0S \mid 1B\}$ on a alors le système $S = 1S + 0A + \varepsilon; A = 1A + 0B; B = 1B + 0S$. En posant $X_1 = S$, $X_2 = A$ et $X_3 = B$, on a le système d'équations donné en exemple lors de la présentation de la méthode de résolution d'un tel système. G génère des chaînes de caractère dont le nombre de 0 est divisible par 3.

Remarque : De cette démonstration et de ce que l'on a vu à propos du passage d'une suite de définitions régulières à une expression régulière (§ 2.2.6), il est possible de déduire une méthode pour trouver une expression rationnelle (et donc un langage rationnel) à partir d'une grammaire linéaire droite. Ce passage d'une grammaire régulière $G = (\mathcal{A}, Q, S, R)$ à un système d'expressions régulières S s'effectue par une conversion des règles : $A \to rA \mid sB \mid t \in R \iff X = rX + sY + t \in S$.

Par conséquent, les expressions rationnelles sont aussi appelées “expression régulières” (plus souvent utilisé dans la littérature anglo-saxonne [Per95]). De même, les langages rationnels sont aussi appelés les langages réguliers.

Chapitre 3

Les automates

3.1 Introduction

La notion d'automate a été introduite pour formaliser le concept d'algorithme et celui de calculabilité. Les automates sont définis sur la base de la machine de Turing [Tur36]. Ce sont des objets mathématiques. Ils font donc abstraction des capacités des hommes ou des machines. Un résultat sera calculable s'il est atteint en un nombre fini de pas, ce nombre pouvant être aussi grand que l'on veut. Les notions de calculabilité, de décidabilité et de problème démontrable ne seront pas abordées dans ce cours. Nous allons nous intéresser à la structure des automates et à leur utilisation dans le cadre de la théorie des langages.

L'objectif de ce chapitre est d'introduire les automates à travers la présentation des machines de Turing. Nous verrons plus particulièrement deux cas particuliers : les automates linéairements bornés et les automates à pile. Un troisième cas sera étudié plus précisément au chapitre suivant : les automates finis.

3.2 La machine de Turing

3.2.1 Présentation

Une machine de Turing est composée (Fig. 3.1) :

- d'une *unité centrale* trouvant dans un *état* q_i, pris parmi l'ensemble fini de ses états internes, noté $Q = \{q_1, q_2, ..., q_n\}$. Parmi les états, on distingue un *état initial* (état au lancement de la machine) et des *états finaux* (états de la machine en situation de terminaison). Le comportement de l'unité centrale est déterminé par son *programme*.

- d'un *ruban* de taille illimitée divisé en cases, contenant au départ les données à traiter, et à l'arrivée les résultats des calculs. Les données de départ tout comme les résultats sont représentés sur le ruban à l'aide de *symboles* pris dans un *alphabet* $\mathcal{A} = \{a_0, a_1, a_2, ..., a_n\}$. Chaque case ne contient qu'un symbole.

- d'une *tête de lecture-écriture* qui assure la communication entre l'unité centrale et le ruban. Cette tête n'agit que sur une case à la fois. Elle peut *remplacer le symbole lu* par un autre symbole et *déplacer le ruban* d'une case vers la gauche ou vers la droite.

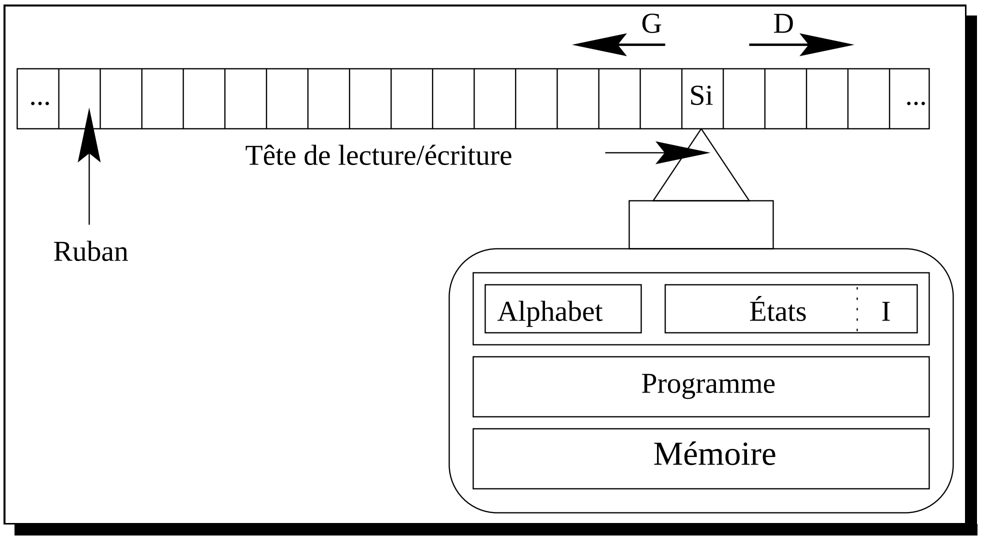


FIG. 3.1 – *Structure d'une machine de Turing*

Le rôle du programme consiste à déterminer, pour un état q_i de la machine et pour un symbole a_j situé devant la tête de lecture-écriture, le nouvel état dans lequel passe la machine ainsi que l'opération effectuée sur le ruban. On représente un programme par un ensemble fini de quintuplets, appelé *fonction de transition*. Celle-ci définit les règles d'évolution, appelées *règles de transition*, de la forme $(q_i, a_j, q_k, a_l, d_m)$ ou $\mu(q_i, a_j) = (q_k, a_l, d_m)$ où :

- $q_i \in Q$ est l'état initial ;
- $a_j \in \mathcal{A}$ le symbole lu ;
- $q_k \in Q$ le nouvel état ;
- $a_l \in \mathcal{A} \cup \{\varepsilon\}$ le caractère à écrire (ε n'écrit rien, c'est-à-dire laisse le caractère a_j sur le ruban) ;
- $d_m \in \{D, G, C\}$ le sens du déplacement de la tête avec respectivement D pour aller d'une case vers la droite, G d'une case vers la gauche et C pour ne pas se déplacer.

Définition 3.2.1 *On appelle* configuration *tout couple symbole / état. Une règle détermine donc pour une configuration courante, l'action à effectuer et donc la prochaine configuration.*

Définition 3.2.2 *La machine de Turing est dite* déterministe *si pour toute configuration, il n'existe qu'une seule fonction de transition applicable, c'est-à-dire si et seulement si pour* $\mu(x, y) = (z_1, t_1)$ *et* $\mu(x, y) = (z_2, t_2)$ *deux règles, alors* $z_1 = z_2$ *et* $t_1 = t_2$ *. Dans le cas contraire, la machine est dite* non déterministe *(pour une configuration donnée, plusieurs règles sont applicables).*

Parmi les états de la machine, il en existe un particulier : l'*état initial* (q_I). C'est le premier état pris par la machine. Il existe aussi un ensemble d'états appelés *états finaux*. Si la machine de Turing est telle qu'il n'existe pas de règle $\mu(q_i, a_j) = (x, y, z)$ pour l'état q_i et le symbole a_j de la configuration courante alors le ruban est dit *terminal*. Si l'état courant est un état final alors l'exécution est réussie et le programme s'est déroulé normalement. Le cas contraire est une situation d'échec.

La suite de symboles D est une chaîne dite terminale (la disposition de départ étant appelée chaîne initiale). Dans cette configuration $D(q)$, la machine de Turing s'arrête. Par ailleurs, le symbole ε permet aussi souvent d'arrêter la machine, c'est-à-dire qu'il n'existe pas de transition de la forme $\mu(q, \varepsilon)$.

Définition 3.2.3 *Une machine de Turing est définie par un quintruplet* $(\mathcal{A}, Q, I, F, \mu)$ *tel que :*

- $\mathcal{A}$ *est l'alphabet (fini, non vide) ;*
- Q *est l'ensemble des états possibles pour la machine (fini et non vide) ;*
- $I \in Q$ *l'état initial ;*
- $F \subseteq Q$ *l'ensemble des états finaux ;*
- μ *la fonction de transition telle que* $\mu : Q \times \mathcal{A} \rightarrow Q \times \mathcal{A} \times \{D, G, C\}$. $\mu(q_i, a_j) = (q_k, a_l, d_m)$ *signifie qu'on effectue les actions de remplacement, de déplacement et de changement d'état en* q_k *pour la configuration courante d'état et de symbole.*

La suite $D_1(q_1), ..., D_n(q_m)$, états (configurations) successifs du ruban et du programme lors de l'exécution d'une machine de Turing T_μ, est appelée calcul. Si D_1 est la chaîne initiale et D_n une chaîne terminale alors D_n est aussi appelée *résultat* du calcul de D_1 par la machine T_μ.

3.2.2 Cas particuliers de machine de Turing

Les différents types d'automates sont des cas particuliers de la machine de Turing. Parmi ceux-ci, on trouve les automates : linéairement bornés, à pile, finis...

3.3 Machine de Turing et la théorie des langages

La machine de Turing manipule des symboles. Ceci n'est pas restrictif car tous les problèmes calculables (pour lesquels il existe un algorithme permettant de les résoudre dans un temps fini) peuvent se ramener à la manipulation de symboles et donc peuvent être programmés sur une machine de Turing.

Dans le cadre de la théorie des langages, la correspondance est évidente. Les machines de Turing sont alors appelées *accepteurs* ou *reconnaisseurs* [ASU91, Gin68].

Une chaîne awb (a et $b \in \mathcal{A}$ et $w \in \mathcal{A}^*$), écrite sur le ruban et entourée de caractères ε, est acceptée si la machine part de la position gauche (i.e. dans la configuration initiale (I, a)) et s'arrête avec la tête de lecture placée après le caractère le plus à droite différent de ε dans un état final (i.e. dans la configuration finale (q_f, b) avec $q_f \in F$). Si elle s'arrête dans un état n'étant pas final alors la chaîne est refusée.

Le langage défini par un accepteur est l'ensemble des chaînes qu'il accepte (fig. 3.2). Il existe donc une correspondance entre les grammaires (et les langages) et les machines de Turing. Une machine de Turing (ou un automate) est donc alors vue comme l'implémentation d'une fonction f telle que : $f : \mathcal{A}^* \rightarrow \{$ Accepté, Refusé $\}$.

Remarque : Cette représentation des accepteurs correspond à des machines de Turing de la classe «linéairement bornées» que nous allons d'ailleurs étudier maintenant.

Chaîne de symboles $\longrightarrow$ $\boxed{\text{Reconnaisseur}}$ $\longrightarrow$ Oui / Non

FIG. 3.2 – *Machine de Turing vue comme un reconnaisseur*

3.4 Les automates linéairement bornés

Les automates linéairement bornés constituent une classe d'automates incluse dans celle des machines de Turing. Leur particularité consiste dans le fait que la zone du ruban sur lequel travaille la machine, est bornée. Le ruban est alors considéré comme fini dans les deux directions.

Définition 3.4.1 *Un automate linéairement borné est composé de la même manière que la machine de Turing générale (Déf. 3.2.3). Il est défini par un quintuplet* $(\mathcal{A}, Q, I, F, \mu)$ *tel que (Fig. 3.3) :*

- $\mathcal{A} \cup \{\#\}$ *est l'alphabet (fini, non vide) ;*
- Q *est l'ensemble des états possibles pour la machine (fini et non vide) ;*
- $I \in Q$ *l'état initial ;*
- $F \subseteq Q$ *les états finaux ;*
- μ *la fonction de transition telle que* $\mu : Q \times \mathcal{A} \to Q \times \mathcal{A} \times \{D, G, C\}$*. Pour la configuration courante d'état et de symbole* $\mu(q_i, a_j) = (q_k, a_l, \delta_m)$ *signifie que :*
  - *on effectue le remplacement par le symbole* a_l*,*
  - *on déplace le curseur (à droite avec* $\delta_m = D$ *ou à gauche avec* $\delta_m = G$*) ou on ne le déplace pas (avec* $\delta_m = C$*),*
  - *enfin on passe dans l'état* q_k*.*

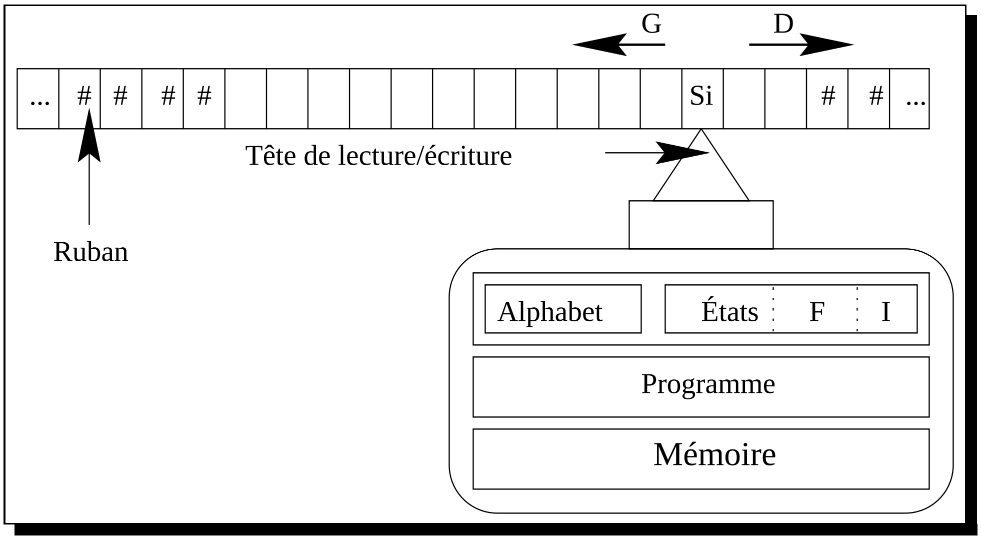


FIG. 3.3 – *Structure d'un automate linéairement borné*

En dehors des bornes, on suppose qu'il y a un caractère spécifique toujours remplacé par lui-même et noté # (ou parfois ε).

Le calcul de l'automate s'effectue ainsi sur une chaîne awb (a et $b \in \mathcal{A}$ et $w \in \mathcal{A}^*$) :

1. au départ, l'unité centrale se trouve dans la configuration (I, a) ;

2. les instructions du programme sont appliquées[1] ;

3. le programme s'arrête lorsqu'aucune règle ne peut s'appliquer ou que la tête se trouve devant un marqueur et dans un état final ;

4. si le programme s'arrête avec la tête de lecture - écriture devant un marqueur et que l'unité centrale se trouve dans un état final (i.e. dans la configuration $(q, \#)$ avec $q \in F$), on dit que le mot est accepté. S'il s'arrête d'une autre façon, il est refusé.

Il est possible de montrer que, pour tout automate linéairement borné A, il existe une grammaire G sous-contexte telle que $L(G) = L_A$ où L_A représente l'ensemble des mots reconnus par l'automate. Pour cela, il suffit de prendre comme vocabulaire terminal celui de l'automate augmenté du symbole # et de construire le vocabulaire non terminal en associant à chaque situation (q_i, a_j) un symbole A_{ij}, un axiome S et un symbole X. Les règles de production de la grammaire sont telles que :

1. $S \to XA_{if}$; $X \to Xa_i$; $X \to a_i$ pour toute situation terminale (a_i, q_f) avec $q_j \in F$;

2. $a_l A_{rk} \to A_{ij} a_r$ pour des transitions de la forme $\mu(q_i, a_j) = (q_k, a_l, D)$;

3. $A_{rk} a_l \to a_r A_{ij}$ pour des transitions de la forme $\mu(q_i, a_j) = (q_k, a_l, G)$;

4. $A_{lk} \to A_{ij}$ pour des transitions de la forme $\mu(q_i, a_j) = (q_k, a_l, C)$;

5. $\#A_{i0} \to \#a_i$ pour tout $a_i \in V_T$.

La grammaire ainsi obtenue vérifie la condition de non-décroissance et est donc sous-contexte. Elle engendre le même langage que l'automate.

Il est possible de prouver la réciproque, c'est-à-dire qu'à toute grammaire sous-contexte G, on peut associer un automate A acceptant le langage engendré par la grammaire. La démonstration est basée sur la transformation des règles de production de la grammaire afin d'obtenir des règles de degré 1 ou 2, c'est-à-dire de la forme $u \to v$ avec u et v des chaînes de longueur inférieure ou égale à 2. C'est à partir de ces règles de production que sont créées les instructions de l'automate correspondant.

Par conséquent, la classe des langages engendrés par des grammaires sous-contexte, coïncide avec la classe des langages reconnus par des automates linéairement bornés (généralement non-déterministes).

1. L'automate peut être déterministe ou non.

3.5 Les automates à pile

Un automate à pile est une machine de Turing dont la mémoire est exploitée d'une façon très spécifique. Elle est considérée comme un ruban limité à gauche mais potentiellement infini à droite. Elle sert au stockage de résultats intermédiaires. Le ruban d'entrée est, quant à lui, toujours infini des deux cotés et contient les données [2].

L'unité centrale possède deux têtes : une première capable de lire les données sur le ruban d'entrée (de gauche à droite et un caractère à la fois) et une seconde capable de lire et écrire des informations sur le ruban de mémoire appelé pile (lecture et écriture se faisant toujours en sommet).

Définition 3.5.1 *Un automate à pile est défini par un 7-uplet* $(\mathcal{A}_E, \mathcal{A}_I, Q, I, @, F, \mu)$ *tel que (Fig. 3.4) :*

- $\mathcal{A}_E$ *est l'alphabet d'entrée (fini, non vide);*
- $\mathcal{A}_I$ *est l'alphabet de la pile (fini, non vide), appelé aussi alphabet intermédiaire, comportant* ε *et qui pouvant même inclure* $\mathcal{A}_E$*;*
- $@ \in \mathcal{A}_I$ *est le symbole de départ de la pile (fond de la pile) ;*
- Q *est l'ensemble des états possibles pour la machine (fini et non vide) ;*
- $I \in Q$ *correspond à l'état initial ;*
- $F \subseteq Q$ *définit l'ensemble des états finaux ;*
- μ *correspond à la fonction de transition telle que* $\mu : \mathcal{A}_E \cup \varepsilon \times Q \times \mathcal{A}_I \to Q \times \mathcal{A}_I^*$*. Pour la configuration courante avec* a_i *sur le ruban d'entrée,* q_j *l'état courant et* A_p *sur le ruban de mémoire,* $\mu(a_i, q_j, A_p) = (q_m, X)$ *signifie qu'on effectue un passage à l'état* q_m *et une action dépendant des caractères présents sur les deux rubans.*

Une configuration de l'automate à pile est définie par le triplet $(a, q, P) \in \mathcal{A}_E \times Q \times \mathcal{A}_I^*$ (avec respectivement : le symbole du ruban de lecture, l'état courant et l'état de la pile). Au départ, la donnée est écrite sur le ruban d'entrée sous forme d'une suite de symboles encadrée par des ε. L'unité centrale se trouve dans la configuration $(a_i, q_0, @)$.

S'il n'existe aucune instruction applicable à la situation courante, l'automate s'arrête. De même, si la pile est vide alors aucune transition n'est possible.

S'il existe une ou plusieurs transitions, l'automate en choisit une : par exemple $\mu(a_i, q_j, \alpha) = (q_m, \omega)$. On obtient alors le changement de configuration (la dérivation) suivant : $(a_i, q_j, \alpha\gamma) \to (a_{i+1}, q_m, \omega\gamma)$. Autrement dit, le résultat de l'application de la transition $\mu(a, q_j, A_p) = (q_m, X)$ est :

1. l'unité centrale passe dans l'état q_m ;
2. suivant a :
   - $= \varepsilon$, le ruban d'entrée ne change pas de position,
   - $= a_i \in \mathcal{A}_E$, le ruban d'entrée se déplace vers la gauche d'une case ;

2. En pratique, ne servant qu'en lecture, il est borné autour des données inscrites

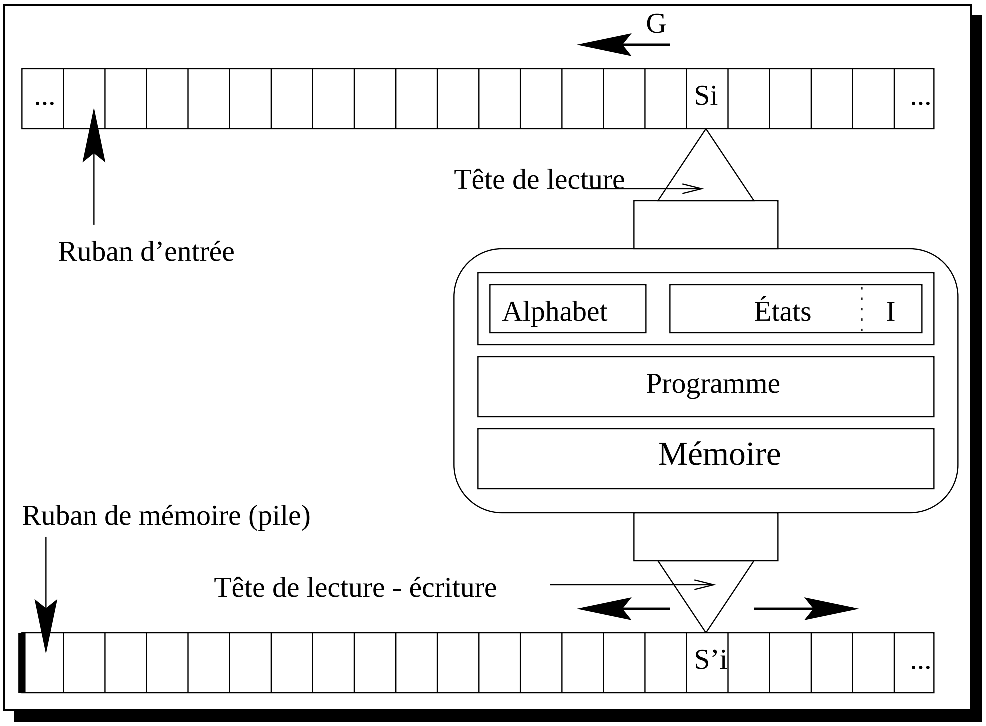


FIG. 3.4 – *Structure d'un automate à pile*

3. suivant X :

   - $= \varepsilon$, le symbole courant du ruban mémoire A_p est effacé et le ruban est déplacé d'une case vers la droite[3],
   - $= \alpha \in \mathcal{A}_I^*$, la suite de symboles qui constituent A_p est effacée et la suite de symboles constituant X est inscrite à droite du symbole courant sur le ruban de mémoire. Ce ruban est alors déplacé du nombre de symboles de X de telle sorte que la tête se positionne en face du dernier symbole de X[4].

Une chaîne $x \in \mathcal{A}_E^*$ est acceptée par un automate à pile si le calcul se termine dans la situation $(\varepsilon, q, \varepsilon), q \in F$.

Le langage accepté par un automate à pile est défini comme l'ensemble des chaînes acceptables par l'automate, c'est-à-dire s'il existe un calcul pour lequel la chaîne est acceptée. Il est possible de montrer que la classe des langages acceptés par les automates à pile coïncide avec la classe des langages hors-contextes.

3.6 Les automates finis

Les automates finis sont des machines de Turing très spécifiques. Leurs caractéristiques, leur manipulation et leurs applications sont présentés en détail dans les chapitres qui suivent.

3. Le fonctionnement de l'automate à la rencontre du symbole ε matérialise l'action de dépilage.
4. Cette partie du fonctionnement de l'automate matérialise l'action d'empilage.

Chapitre 4

Les automates finis

Nous allons étudier en détail un cas particulier de machine de Turing : les automates finis. Nous en donnerons d'abord une définition précise. Nous verrons ensuite les différents modes de représentation, leurs propriétés ainsi que les algorithmes permettant de les utiliser.

4.1 Présentation des automates finis

La théorie des automates finis commence avec l'article de S.C. Kleene [Kle56] paru en 1956. Dans cet article tant de fois cité, Kleene démontre le premier résultat de cette théorie. C'est à la fois le premier dans l'ordre chronologique mais aussi dans l'ordre d'importance. Nous verrons un peu plus loin ce théorème.

Un automate fini, très souvent appelé «automate d'état fini» [Per95], («finite-state machine», «finite automaton», «Rabin-Scott Automaton» [Gin68]) est une machine de Turing dont le ruban se déplace toujours vers la gauche et case par case. Ce ruban est infini à droite et fini à gauche. Par ailleurs, la tête de lecture de la machine se contente de lire des symboles et n'effectue aucune écriture.

Définition 4.1.1 *Un automate fini (AFN) est composé de la même manière que la machine de Turing (Déf. 3.2.3). Généralement non déterministe, son comportement est plus restreint. Il est défini par un quintuplet* $(\mathcal{A}, Q, I, F, \mu)$[1] *tel que (Fig. 4.1) :*

- $\mathcal{A}$ *est l'alphabet (fini, non vide) ;*
- Q *est l'ensemble des états possibles pour la machine (fini et non vide) ;*
- $I \in Q$ *est l'état initial ou état de départ ;*
- $F \subseteq Q$ *correspond aux états finaux ou états d'acceptation ;*
- μ *la fonction de transition*[2] *telle que* $\mu : \mathcal{A} \cup \{\varepsilon\} \times Q \to Q$*. Pour la configuration courante*

1. Certains auteurs, dont [Wat93a], définissent les automates par un 6-uplet $(\mathcal{A}, Q, E, I, F, \mu)$ en ajoutant un ensemble E des ε-transitions (Déf. 4.8.1). Dans notre définition, ces transitions sont définies par μ. De plus, [Wat93a] ne définit pas un seul état initial mais un ensemble d'états initiaux.

2. μ est une fonction partielle, car elle n'est pas définie pour toutes les valeurs du domaine. Le plus souvent, pour les couples (symbole, état) qui ne sont pas explicités, elle est définie de la façon suivante : $\mu(a, q) = 0$ où 0 est un état qui n'existe pas. Ces transitions provoquent un arrêt de l'automate ainsi qu'une erreur. L'état 0 peut être considéré comme l'état d'erreur de l'automate.

d'état et de symbole, $\mu(a_j, q_i) = q_k$ signifie qu'on passe dans l'état q_k et que le ruban est déplacé d'une case vers la gauche.

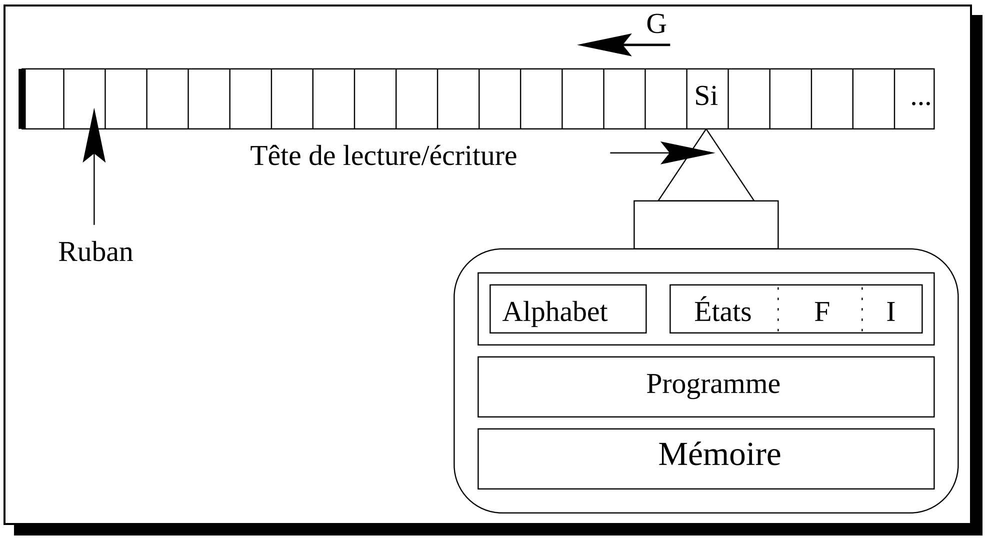


FIG. 4.1 – *Structure d'un automate fini*

Au départ, une phrase construite sur le vocabulaire de l'automate est inscrite sur le ruban d'entrée et est entourée de symboles ε. L'unité centrale se trouve dans l'état initial I et la tête de lecture est placée devant le symbole le plus à gauche de la phrase. A partir de cette position, l'automate va exécuter son programme. Pour une configuration courante (a_i, q_j), l'automate applique une transition $\mu(a_i, q_j) \rightarrow q_k$, passe alors dans l'état q_k et déplace le ruban d'une case vers la gauche. L'automate tel qu'il est défini n'est pas forcément déterministe. Plusieurs transitions concurrentes sont donc parfois applicables[3]. L'automate s'exécute jusqu'à ce qu'il arrive à une situation (a_t, q_n) pour laquelle il n'existe pas de transition permettant au programme de poursuivre son calcul. Il suspend alors son exécution. Si la configuration finale est telle que $a_t = \varepsilon$ et $q_n \in F$ alors la phrase inscrite sur le ruban est *acceptée* par l'automate. Dans tous les autres cas, elle est *refusée*. Le schéma général d'un algorithme basé sur un AFN est décrit à la figure 4.2.

4.2 Formalisme de description

Les automates peuvent être représentés de plusieurs manières : par des graphes orientés ou des représentations matricielles.

Un AFN peut être représenté par un graphe orienté, appelé «diagramme de transition» ou «graphe de transition». Les noeuds correspondent aux états et les arcs représentent les transitions possibles (calculées par la fonction de transition) entre les états. Ces arcs sont étiquetés par le symbole permettant cette transition. L'état initial est représenté par un noeud possédant une flèche entrante sans origine. Un état fial est représenté par un noeud doublement cerclé (fig. 4.3).

3. Une façon utile de regarder le non déterminisme est de considérer qu'il permet à un automate de «formuler des hypothèses». Si, dans un état donné, on ne sait pas quoi faire sur un certain caractère d'entrée, il existe plusieurs possibilités pour le choix de l'état suivant. Comme chaque chemin étiqueté par une chaîne de caractères et conduisant à un état d'acceptation est interprété comme acceptable, on fait confiance à l'automate non déterministe dès qu'il fait une hypothèse juste, quel que soit le nombre d'hypothèses fausses qu'il ait faites auparavant.

```
AutomateFini(↘ (𝒜, Q, I, F, μ))
Initialisations
/ Comporte en particulier le remplissage du diagramme de transition /
/ et l'inscription de la chaîne sur le ruban... /
Se mettre au début du ruban
EtatCourant ← I   / initialisation de l'automate/
Fin ← Faux
Répéter
      Obtenir l'élément e suivant
      / passer à l'état suivant et se déplacer /
      Si ∃μ(EtatCourant, e) ≠ 0
          Alors  Sélectionner une règle μ(EtatCourant, e) / non-déterminisme /
                 EtatCourant ← μ(EtatCourant, e)
                 Avancer d'une case vers la droite
          Sinon  / Retour en arrière /
                 Si  Il est possible de revenir sur un choix de transition
                     Alors  Retourner en arrière dans les choix
                            Reculer la tête vers la gauche d'autant
                     Sinon Fin ← Vrai
                 Fin Si
      Fin Si
Jusqu'à Fin
Retourner EtatCourant ∈ F  et  e = ε
```

Fig. 4.2 – *Algorithme de simulation d'un AFN*

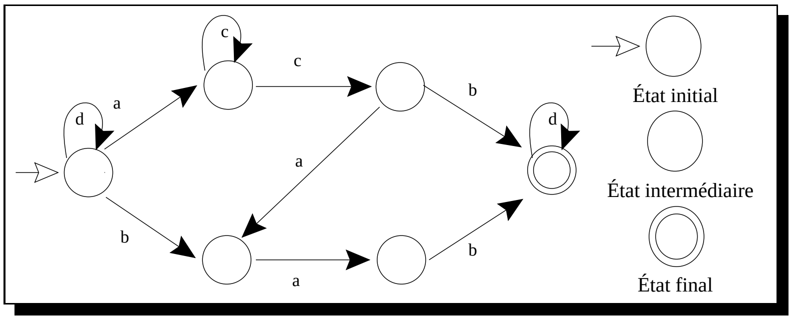


Fig. 4.3 – *Graphe de transition d'un AFN permettant de reconnaître des mots du langage* $d^*(ac^+(\varepsilon|aa)|ba)bd^*$

Par exemple, le graphe de transition représenté par la figure 4.3 correspond à l'AFN $(\mathcal{A}, Q, I, F, \mu)$ avec $\mathcal{A} = \{a, b, c, d\}$, $Q = \{1, 2, 3, 4, 5, 6\}$, $I = 1$ et $F = \{6\}$.

Les fonctions de transition sont, la plupart du temps, définies de manière extensive.

μ peut alors être définie avec :
$\mu(a, 1) = 2; \mu(b, 1) = 3; \mu(d, 1) = 1;$
$\mu(c, 2) = 2; \mu(c, 2) = 4; \mu(a, 3) = 5;$
$\mu(a, 4) = 3; \mu(b, 4) = 6; \mu(b, 5) = 6;$

$\mu(d, 6) = 6$.

Hormis la notation graphique, la différence essentielle entre les représentations des AFN porte sur la fonction de transition.

Souvent la fonction de transition (et donc l'automate) est représentée par des règles. Par exemple, l'AFN précédent est représenté de la manière suivante :
$(a, 1) \rightarrow 2; (b, 1) \rightarrow 3; (d, 1) \rightarrow 1;$
$(c, 2) \rightarrow 2; (c, 2) \rightarrow 4; (a, 3) \rightarrow 5;$
$(a, 4) \rightarrow 3; (b, 4) \rightarrow 6; (b, 5) \rightarrow 6;$
$(d, 6) \rightarrow 6$.

Cette fonction est très souvent représentée par une «table de transition» (parfois aussi appelée «matrice de transition», «flow table» en anglais). Cette représentation est très commode du point de vue de l'implémentation informatique des AFNs. Les tables de transition sont des tableaux (ou des matrices) dont les colonnes correspondent aux symboles et les lignes aux états. La cellule (l_i, c_j) indique les transitions $(a_i, q_j) \rightarrow q_k$. L'AFN étant susceptible d'être non déterministe, plusieurs états peuvent être présents dans une cellule (l_i, c_j). Les caractères ne donnant aucune transition depuis un noeud dans un certain état sont alors à «0» (transition vers un état inexistant). Cet état correspond à un état d'erreur. Si on reprend l'AFN servant d'exemple, nous avons alors la table 4.1. C'est cette forme qui est le plus souvent utilisée en programmation.

μ	a	b	c	d
1	2	3	0	1
2	0	0	2,4	0
3	5	0	0	0
4	3	6	0	0
5	0	6	0	0
6	0	0	0	6

TAB. 4.1 – *Graphe de transition d'un AFN*

Il existe aussi d'autres représentations possibles comme des matrices binaires associées à chacun des symboles...

Remarque. Dans certains cas, il est plus facile de représenter un groupe de transitions par une notation ensembliste. Par exemple, supposons que pour un état donné q_i, nous avons soit un passage à l'état q_j pour les symboles a_p et a_q soit une conservation de l'état courant pour les autres symboles de l'alphabet $\mathcal{A}$. Ce type de transition s'écrit :
$(a_p, q_i) \rightarrow q_j; (a_q, q_i) \rightarrow q_j;$
$(a_1, q_i) \rightarrow q_i; (a_2, q_i) \rightarrow q_i;$ ...Soit :$(a_j, q_i) \rightarrow q_i, \forall i \notin \{p, q\}$

Pour simplifier l'écriture, nous aurons :
$(\{a_p, a_q\}, q_i) \rightarrow q_j; (\mathcal{A} - \{a_p, a_q\}, q_i) \rightarrow q_i$.

Plus généralement, les symboles peuvent être regroupés sous différentes classes et l'automate possède des transitions fonction de ces classes. Nous retrouvons ici la même simplification que nous avions utilisée pour les expressions régulières (nous verrons que cette similitude n'est pas un hasard).

4.3 Configurations

Pour pouvoir exprimer le déroulement d'un programme sur un automate fini, il est nécessaire d'introduire les notions de configuration et d'action.

Une configuration correspond à un état possible de l'automate, c'est-à-dire à un couple (symboles restant à voir sur le ruban d'entrée, état courant).

Définition 4.3.1 *Si $T = (\mathcal{A}, Q, I, F, \mu)$ est un AFN, alors tout couple (a, q) avec $a \in \mathcal{A}^*$ et $q \in Q$ est une configuration de T (voir aussi Déf. 3.2.1).*

Il existe deux ensembles de configurations particulières :

- (a, I) appelées *configurations initiales* ou *configurations de départ* ;
- (ε, q_F) avec $q_F \in F$ appelées *configurations finales* ou *configurations terminales.*

Une action représente le passage d'une configuration donnée à une autre par l'application d'une transition de l'automate. Bien évidemment, pour les AFNs, il peut exister plusieurs actions différentes pour une même configuration (non déterminisme).

Définition 4.3.2 *Une action (ou un calcul, un déplacement) de l'automate $T = (\mathcal{A}, Q, I, F, \mu)$ est représentée par une relation binaire sur les configurations notée $\xmapsto{T}$ ou $\xmapsto{1/T}$. Alors la transition $\mu(a, q_i) = q_j$ peut s'écrire $(aw, q_i) \xmapsto{1/T} (w, q_j)$.*

On dit que $C \xmapsto{0/T} C'$ si et seulement si $C = C'$ et $C \xmapsto{k/T} C_k$, $k \geq 1$ si et seulement si $\exists C_1, ..., C_{k-1}$ tels que $C_i \xmapsto{T} C_{i+1} \forall i, o \leq i < k$ et $C_0 = C$

Notations : $C \xmapsto{+/T} C'$ signifie $C \xmapsto{k/T} C'$ avec $k \geq 1$ et $C \xmapsto{*/T} C'$ signifie $C \xmapsto{k/T} C'$ avec $k \geq 0$.

Remarques :

- $\xmapsto{+/T}$ est la fermeture transitive de $\xmapsto{T}$ et $\xmapsto{*/T}$ est la fermeture réflexive et transitive de $\xmapsto{T}$.
- Parfois, il est possible de trouver l'utilisation d'une fonction partielle récursive μ^* définie à partir de la fonction de transition μ sur $\mathcal{A}^* \times Q \to Q$ telle que :
  $\mu^*(\varepsilon, q) = q$ et $\mu^*(aw, q) = \mu^*(w, \mu(a, q))$.

  Cette fonction est équivalente à la notion d'action : avec $a \in \mathcal{A}$ et $w \in \mathcal{A}^*$,
  $\mu^*(aw, q) = q' \equiv (aw, q) \xmapsto{1/T} (w, q')$ et $\mu^*(w, I) = q_f \equiv (w, I) \xmapsto{*/T} (\varepsilon, q_f)$.

4.4 AFN et langages

Les machines de Turing (et donc les AFNs), nous l'avons vu, sont aussi appelées «accepteurs» («Language Acceptor») lorsqu'elles sont utilisées pour reconnaître un langage donné en Théorie de Langages. Un automate fini sert à reconnaître un langage bien particulier. Il est donc possible de définir *le langage reconnu par un automate donné*. Pour cela, nous utilisons les notions de *langage entre deux états*, de *langage gauche* et de *langage droit.*

Un langage entre deux états correspond aux différentes successions de symboles permettant de passer d'un état donné à un autre.

Définition 4.4.1 *Un langage entre deux états quelconques q_0 et q_1 de $T = (\mathcal{A}, Q, I, F, \mu)$ est défini par :*
$l_T(q_0, q_1) = \{w \mid w \in \mathcal{A}^*, x \in \mathcal{A}^*, (wx, q_0) \overset{*/T}{\longmapsto} (x, q_1)\}$.

Le langage gauche d'un état est l'ensemble des suites de symboles permettant d'atteindre cet état à partir de l'état initial. Le langage droit d'un état est celui permettant, à partir de cet état, d'atteindre un état final.

Définition 4.4.2 *Le langage gauche d'un état $q \in Q$ d'un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est donné par la fonction $\overleftarrow{L}_T \in Q \to \mathcal{A}^*$ où $\overleftarrow{L}_T(q) = l_T(I, q) = \{w \mid w \in \mathcal{A}^*, x \in \mathcal{A}^*, (wx, I) \overset{*/T}{\longmapsto} (x, q)\}$*
Le langage droit d'un état $q \in Q$ d'un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est donné par la fonction $\overrightarrow{L}_T \in Q \to \mathcal{A}^*$ où $\overrightarrow{L}_T(q) = \bigcup_{q_f \in F} l_T(q, q_f) = \{w \mid w \in \mathcal{A}^*, \exists q_f \in F, (w, q) \overset{*/T}{\longmapsto} (\varepsilon, q_f)\}$.

Un langage reconnu par un AFN est donc l'ensemble des suites de symboles permettant d'atteindre un état final à partir de l'état initial.

Définition 4.4.3 *Un langage reconnu (accepté, défini) par un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est donné par la fonction $L \in AFN \to \mathcal{A}^*$ où $L(T) = \bigcup_{q_f \in F} l_T(I, q_f) = \overrightarrow{L}_T(I) = \bigcup_{q \in F} \overleftarrow{L}_T(q)$. $L(T)$ est donc l'ensemble des chaînes acceptées par T, c'est-à-dire :*
$L(T) = \{\omega \mid \omega \in \mathcal{A}^*, \exists q \in F : (\omega, I) \overset{*/T}{\longmapsto} (\varepsilon, q)\}$.

Il est possible alors de comparer deux automates à partir des langages qu'ils reconnaissent.

Définition 4.4.4 *Deux automates sont équivalents s'ils acceptent le même langage, c'est-à-dire :* $T_1 \equiv T_2 \iff L(T_1) = L(T_2)$.

4.5 Automates finis généralisés

Les AFNs sont très utilisés pour les traitements séquentiels sur des suites de symboles (mots, phrases, suites d'actions...), à cause de la facilité de mise en oeuvre du procédé et des avantages qu'ils offrent dans le cas de modifications ou d'extensions d'un traitement.

Cependant, on utilise souvent des AFNs généralisés qui ne se contentent pas de reconnaître des chaînes de symboles, mais éventuellement qui les traitent au passage. Les généralisations classiques sont les suivantes :

- la tête de lecture reconnaît une classe de symboles (au minimum, une par symbole et toutes disjointes), et pas seulement un symbole (comme nous l'avons vu au paragraphe précédent) ;
- un traitement particulier peut être associé à un état de la machine, visant soit à produire un résultat, soit à modifier l'état de l'automate ;
- un traitement d'erreur est souvent prévu en cas de mauvaise analyse d'un symbole.

```
AutomateFiniGénéralisé(↘ (𝒜, Q, I, F, μ))
 / schéma général d'utilisation d'un automate fini /
 Initialisations
 / comporte en particulier le remplissage du diagramme de transition /
 Se mettre au début du ruban Ch
EtatCourant ← I   / initialisation de l'automate/
Erreur ← Faux
Répéter
       Obtenir l'élément e suivant dans Ch
       / trouver la classe de l'élément /
      Cl ← Classe(e)
       / passer à l'état suivant /
       Choisir une transition
      EtatCourant ← μ(EtatCourant, Cl)
       / traiter l'élément /
      Suivant
         EtatCourant = e1 :  Liste action 1
                            ...
         EtatCourant = en :  Liste action n
         Autrement  :  Mise à jour de Erreur ou retour arrière
      Fin Suivant
Jusqu'à (EtatCourant ∈ F  et  Ch = ε)  ou  Erreur
 Traitement lors de l'arrêt de l'automate
```

Fig. 4.4 – *Algorithme de simulation d'un AFN généralisé*

Le schéma général d'un algorithme utilisant un AFN généralisé est décrit à la figure 4.4. Les actions liées à la gestion du non-déterminisme n'ont pas été présentées dans un souci de clarté de l'algorithme.

Une application illustrant cette généralisation concerne les « transducteurs» («Machine séquentielle»)[4]. Un transducteur permet de réaliser une transduction de l'alphabet d'entrée vers un alphabet de sortie. La figure 4.5(a) illustre un automate à deux états[5] dont l'alphabet est binaire et dont les états concervent le dernier caractère lu. La figure 4.5(b) illustre une sortie sur l'alphabet $\{x, y, z, t\}$. La transduction réalisée effectue le codage par blocs chevauchants de longueur 2 avec la correspondance: $x = 00$, $y = 01$, $z = 10$ et $t = 11$. Le mot 0110 fait passer successivement l'automate par les états p, p, q, q, p et le mot de sortie est $xyzt$ (codage du mot d'entrée précédé de 0).

4. Un transducteur est aussi appelé «Machine de Moore» lorsque la sortie est produite pour chaque état et «Machine de Mearly» lorsqu'elle est produite pour chaque transition.

5. $(\{0, 1\}, \{1, 2\}, 1, \{1, 2\}, \mu)$

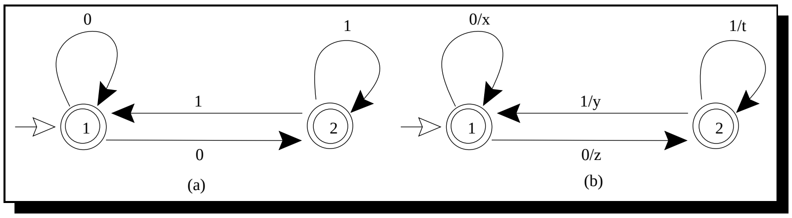


FIG. 4.5 – *AFN et transducteur*

4.6 Propriétés des états

Outre les états finaux et l'état initial, il existe d'autres catégories d'états spécifiques : les états stériles, accessibles, isolés...

Définition 4.6.1 *Un état stérile (état mort ou «dead state», «trap state») e est un état pour lequel il n'est pas possible d'atteindre un état final*[6]*, c'est-à-dire si :* $\overrightarrow{L}(e) = \emptyset$.

Bien sûr, ces états sont à éviter lors de la construction d'un automate sauf s'ils sont finaux et qu'il existe un chemin depuis l'état initial. De même, les noeuds non initiaux pour lesquels aucune transition n'aboutit[7] ne sont pas utiles, et sont donc à éviter. L'état ne faisant l'objet d'aucune transition est dit *isolé* et n'est, lui aussi, d'aucune utilité.

Plus généralement, tout état qui ne peut être atteint depuis l'état initial en suivant les transitions (on dit qu'ils ne sont pas *accessibles* depuis l'état initial) ou qui est stérile est à supprimer.

Définition 4.6.2 *Un état e est dit «accessible depuis un état» f pour un AFN* $T = (\mathcal{A}, Q, I, F, \mu)$ *si* $\exists wx \in \mathcal{A}^*$ *telle que* $(wx, f) \overset{*/T}{\longmapsto} (x, e)$.

Un état e est dit «accessible» pour un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ *si* $\exists wx \in \mathcal{A}^*$ *telle que* $(wx, I) \overset{*/T}{\longmapsto} (x, e)$ *(il existe au moins un chemin de I à e, c'est-à-dire* $\overleftarrow{L}_T(e) \neq \emptyset$*). Dans le cas contraire, l'état e est dit «inaccessible» (il n'existe pas de chemin de I à e, c'est-à-dire* $\overleftarrow{L}_T(e) = \emptyset$*).*

Par conséquent, un état (et les transitions qui le concernent) peut être supprimé de l'automate s'il est *inutile*, c'est-à-dire inaccessible ou stérile.

Définition 4.6.3 *Soit un AFN* $T = (\mathcal{A}, Q, I, F, \mu)$ *et un état* $e \in Q$ *:*

- *e est* $utile_I$ *s'il est accessible, c'est-à-dire si :* $\overleftarrow{L}(e) \neq \emptyset$ *;*
- *e est* $utile_F$ *s'il n'est pas stérile, c'est-à-dire si :* $\overrightarrow{L}(e) \neq \emptyset$ *:*
- *e est utile si :* $utile_I(e) \wedge utile_F(e) \equiv Accessible(e) \wedge \neg Sterile(e)$*. Dans le cas contraire, l'état est dit inutile.*

6. En théorie des graphes, ils sont aussi appelés «noeuds puits».
7. Ils sont appelés «noeuds sources» en théorie des graphes.

4.7 Propriétés des AFNs

Outre le fait qu'un automate soit ou non déterministe, il est possible de mettre en évidence d'autres caractéristiques.

La *dimension* d'un automate fini correspond simplement au nombre d'états qu'il possède. Cette dimension peut être *minimale* lorsqu'il n'est plus possible de trouver un automate équivalent de dimension strictement inférieure. Nous verrons au paragraphe 5.3 un algorithme pour minimiser un automate fini.

Définition 4.7.1 *La dimension d'un AFN $T = (\mathcal{A}, Q, I, F, \mu)$, notée $\mid T \mid$ est définie par :* $\mid T \mid = \mid Q \mid$.

Un automate fini *complet* est un automate pour lequel, pour toute configuration, il existe une transition vers un état de cet automate. La fonction d'appartenance est totalement définie (pour tout couple (symbole courant, état courant)).

Définition 4.7.2 *Un AFN complet $T = (\mathcal{A}, Q, I, F, \mu)$ est un AFN vérifiant :* $Complet(T) \equiv (\mu(a, q) \neq \emptyset, \forall q \in Q, \forall a \in \mathcal{A})$.

Remarque : $Complet(T) \Rightarrow \{\bigcup \overleftarrow{L}(q), \forall q \in Q\} = \mathcal{A}^*$.

Un automate fini est *utile* si tout état de cet automate possède un langage gauche et un langage droit, c'est-à-dire si tout état de l'automate peut être utilisé pour reconnaître au moins un mot du langage.

Définition 4.7.3 *Soit un AFN $T = (\mathcal{A}, Q, I, F, \mu)$:*

- *$Utile_I$ est défini par :* $Utile_I(T) \equiv utile_I(q), \forall q \in Q$;
- *$Utile_F$ est défini par :* $Utile_F(T) \equiv utile_F(q), \forall q \in Q$;
- *$Utile$ est défini par :* $Utile(T) \equiv Utile_I(T) \wedge Utile_F(T) \equiv utile(q), \forall q \in Q$.

4.8 Cas des ε-transitions

Parmi les transitions possibles dans un AFN, il en existe une particulièrement spécifique appelée l'ε-transition. De telles transitions sont caractéristiques des automates *non déterministes*. La figure 4.6 propose un exemple d'automate comportant des ε-transitions.

Définition 4.8.1 *Une ε-transition est une transition qui n'utilise aucune entrée. C'est une transition «spontanée», c'est-à-dire que l'automate «décide» simplement de changer d'état sans lire de symbole.*

Ces transitions très particulières sont principalement introduites lors de la construction automatique d'un automate à partir d'une expression régulière (§ 5.4). Cependant, de part leur nature, elles sont à éviter. Nous verrons comment les supprimer en présentant la transformation d'un AFN en AFD (§ 5.2.2). En effet, à tout AFN comportant des ε-transitions il est possible

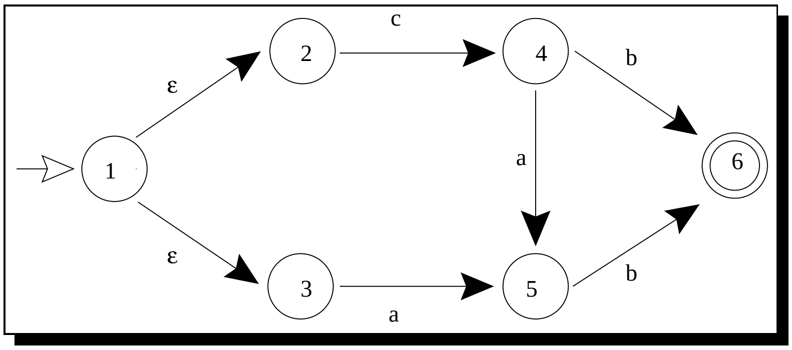


FIG. 4.6 – *AFN avec ε-transitions*

de faire correspondre un AFN (et même un AFD) équivalent, c'est-à-dire reconnaissant le même langage, sans ε-transition.

Théorème 4.8.1 *A tout automate $T = (\mathcal{A}, Q, I, F, \mu)$ comportant des ε-transitions existe un automate $T' = (\mathcal{A}, Q, I, F', \varepsilon - fermeture_\mu)$ équivalent, sans ε-transition, avec $F' = F \cup \{I\}$ s'il existe une chaîne de ε-transitions entre I et un état $q \in F$ et $F' = F$ sinon.*

Dém. Voir [Sha97] par induction sur la longueur de la chaîne.

La fonction $\varepsilon - fermeture_\mu(a, q)$ détermine tous les états dans lesquels peut passer l'automate dans l'état q après (1) zéro ou plus ε-transitions, (2) une transition sur a puis (3) zéro ou plus ε-transitions[8].

L'AFN sans ε-transition obtenu est alors appelé ε-libre.

Définition 4.8.2 *Un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est dit ε-libre s'il ne possède pas de ε-transition, c'est-à-dire si : $\mu(\varepsilon, a) = 0, \forall q \in Q$.*

Remarque : Même si un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est ε-libre, il est possible que $\varepsilon \in L(T)$ quand $I \in F$.

4.9 AFD et AFN

4.9.1 Un AFD

Lorsqu'il n'existe au plus qu'une seule transition pour une configuration donnée, l'automate fini est alors dit *déterministe*. La figure 4.7 propose un exemple d'automate déterministe reconnaissant des chaînes du langage a^+b^*c.

Définition 4.9.1 *Un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est dit déterministe (AFD) s'il est ε-libre et si $\forall q \in Q, \forall a \in \mathcal{A} : \mid \mu(a, q) \mid \leq 1$. Autrement dit, si $\mu(a, q) = q_1$ et $\mu(a, q) = q_2 \Rightarrow q_1 = q_2$.*

8. L'algorithme permettant de déterminer l'ε-fermeture pour un ensemble d'états sera décrit au paragraphe 5.2.2 et donné par la figure 5.2.

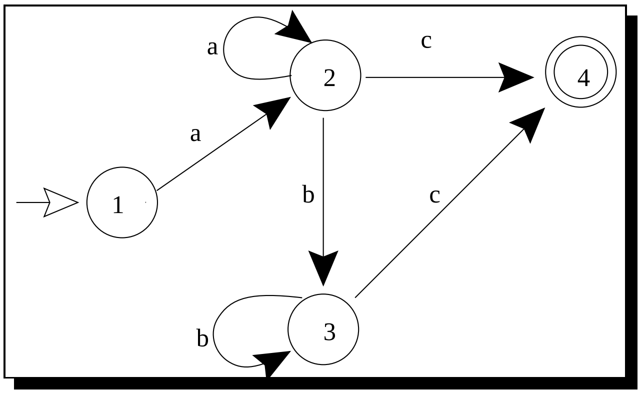


FIG. 4.7 – *Un AFD reconnaissant les chaînes du langage* a^+b^*c

Notation : L'expression $\mid \mu(a, q) \mid$ avec $q \in Q, a \in \mathcal{A}$ indique le nombre de transitions existant dans l'automate sur la configuration (a, q).

Remarque : Un AFD ne doit posséder qu'un seul état initial (dans le cas de la définition des AFNs à états initiaux multiples, voir déf. 4.1.1).

Les AFDs sont intéressants car, lors d'une analyse, ils ne posent pas de problème de choix : pour un état et un symbole, il n'existe qu'une seule transition possible.

Le schéma général d'un algorithme de reconnaissance de chaînes en utilisant un AFD est décrit à la figure 4.8. Notons que dans ce cas, le ruban est remplacé directement par la chaîne de caractères donnée. L'analyse se termine si tous les caractères ont été vus ou si l'état courant est l'état d'erreur.

RECONNAÎTRELANGAGE($\searrow AFD(\mathcal{A}, Q, I, F, \mu), \searrow Chaine$)
$Initialisationdel'automate$
/ Configuration initiale /
$EtatCourant \leftarrow I$
$Ch \leftarrow PremierCaractere(Chaine)$
Tant Que $Ch \neq \varepsilon$ et $EtatCourant \neq 0$ **Faire**
    / Effectuer la transition /
    $EtatCourant \leftarrow \mu(Ch, EtatCourant)$
    / Lire le prochain symbole /
    $Ch \leftarrow CaractereSuivant(Chaine)$
Fin Tant Que
/ Configuration finale /
Si $EtatCourant \in F$ et $Ch = \varepsilon$
  Alors Retourner $Vrai$
  Sinon Retourner $Faux$ / Echec de la reconnaissance /
Fin Si

FIG. 4.8 – *Algorithme de reconnaissance d'une chaîne par un AFD*

4.9.2 AFD minimal

Un automate fini déterministe minimal est un cas particulier d'AFD pour lequel il n'existe pas d'AFD équivalent possèdant moins d'états. Par contre, il est possible qu'il y en ait avec autant d'états (l'AFD minimal n'est pas unique). Pour définir un automate minimal, il est possible de s'appuyer sur les notions d'états indistinguables et d'automates réduits.

Définition 4.9.2 *On dit que la chaîne d'entrée $w \in \mathcal{A}^*$ distingue l'état e de l'état f $(e \neq f)$ dans l'AFD $T = (\mathcal{A}, Q, I, F, \mu)$ si :*

- *partant de l'état e et le faisant fonctionner avec w, on termine dans un état d'acceptation ;*
- *partant de l'état f et le faisant fonctionner avec w, on termine dans un état de non-acceptation (ou vice versa).*

Les états e et f sont alors dits distinguables. Autrement dit, w distingue e et f si $\exists q_1, q_2 \in Q, (x, e) \overset{*/T}{\longmapsto} (\varepsilon, q_1)$ et $(x, f) \overset{*/T}{\longmapsto} (\varepsilon, q_2)$ avec exactement 1 des états q_1 ou $q_2 \in F$.

Définition 4.9.3 *Deux états e et f d'un AFD T sont k-indistinguables, noté $e \overset{k}{\equiv} f$ si : $\nexists w \in \mathcal{A}^*, \mid w \mid \leq k$ qui distingue e et f.*

Deux états e et f d'un AFD T sont indistinguables, si $\forall k \geq 0, e \overset{k}{\equiv} f$.

Un automate *réduit* est un automate qui ne possède pas de couples d'états indistinguables.

Définition 4.9.4 *Un automate $T = (\mathcal{A}, Q, I, F, \mu)$ est dit réduit si tous les états de Q sont accessibles et si tous les d'états de Q pris deux à deux sont distinguables.*

Un automate minimal est alors un automate réduit dont on ne peut trouver un automate équivalent avec un nombre d'états strictement inférieur au sien (de dimension minimale).

Définition 4.9.5 *Un AFD $T = (\mathcal{A}, Q, I, F, \mu)$ est dit minimal s'il est réduit et si :*
$Min(T) \equiv \forall T' \in AFD, L(T) = L(T') :\mid T \mid \leq \mid T' \mid.$

Un AFD $T = (\mathcal{A}, Q, I, F, \mu)$ est dit minimal complet si :
$Min_C(T) \equiv \forall T' \in AFD, Complet(T'), L(T) = L(T') :\mid T \mid \leq \mid T' \mid.$

Remarque : Parfois, on trouve la notion d'automate *faiblement* déterministe. Un AFN $T = (\mathcal{A}, Q, I, F, \mu)$ est dit faiblement déterministe (AFd) si :
$\forall q_0 \in Q, \forall q_1 \in Q, q_0 \neq q_1, \overleftarrow{L}(q_0) \cap \overrightarrow{L}(q_1) = \emptyset.$

On peut montrer que $AFD \Rightarrow AFd$.

Chapitre 5

Manipulation des automates finis

5.1 Introduction

Après avoir introduit les automates finis non déterministes (AFN) et déterministes (AFD), nous allons tout d'abord présenter des algorithmes permettant de passer d'un AFN à un AFD puis permettant de simplifier un AFD. Puis, nous étudierons le processus pour de passer d'une expression régulière à un automate fini déterministe, c'est-à-dire à un AFD capable de reconnaître des mots du langage défini par cette expression.

5.2 Passage d'un AFN vers un AFD

Il n'est pas difficile de comprendre que la complexité d'un algorithme de reconnaissance basé directement sur un AFN est beaucoup plus importante que celle basée sur un AFD. Hors, il est possible de montrer que, pour tout AFN, il existe un AFD équivalent, c'est-à-dire reconnaissant le même langage. Il est donc intéressant de chercher à transformer un AFN en AFD. Pour cela, nous allons présenter une méthode simple appelée la méthode par construction des sous-ensembles.

5.2.1 Théorème

L'algorithme de transformation d'un AFN en AFD n'est possible que parce qu'on peut montrer qu'à un AFN donné il existe un AFD équivalent.

Théorème 5.2.1 *Soit $T = (\mathcal{A}, Q, I, F, \mu)$ un AFN qui reconnaît le langage $L = L(T)$ alors il existe un AFD $T' = (\mathcal{A}, Q', I', F', \mu')$ tel que $L' = L(T') = L$.*

Dém. (constructive) On construit T' de la façon suivante :

1. $Q' \subseteq P(Q)$, c'est-à-dire que les états de T' sont des ensembles d'états de T ;
2. $I' = \{I\}$;
3. $F' = \{Q' \subseteq Q : Q' \cap F \neq \emptyset\}$;
4. $\forall S \subset Q$ et $\forall a \in \mathcal{A}, \mu'(a, S) = S'$ où $S' = \{p \mid \exists q \in S : p \in \mu(a, q)\}$.

On peut montrer que $(w,S) \overset{i/T'}{\longmapsto} (\varepsilon,S')$ si et seulement si $S' = \{p \mid \exists q \in S : (w,q) \overset{i/T}{\longmapsto} (\varepsilon,p)\}$. Cas particulier : $(w,\{q_0\}) \overset{i/T'}{\longmapsto} (\varepsilon,S')$ pour $S' \in F'$ si et seulement si $(w,q_0) \overset{i/T}{\longmapsto} (\varepsilon,p)$ pour $p \in F$. En effet :

1. Vrai pour $i=1$: $(a,S) \overset{1/T'}{\longmapsto} (\varepsilon,S')$ donc $\forall p \in S', \exists q \in S, p \in \mu(a,q)$ d'où $\forall p \in S(a,q) \overset{1/T}{\longmapsto} (\varepsilon,p)$ donc $S' = \{p \mid \exists q \in S : (w,q) \overset{1/T}{\longmapsto} (\varepsilon,p)\}$;

2. Supposons vrai pour $i=n$, montrons pour $i=n+1$: $(aw,S) \overset{n+1/T'}{\longmapsto} (\varepsilon,S') \iff (aw,S) \overset{1/T'}{\longmapsto}_1 (w,S'') \overset{n/T'}{\longmapsto}_2 (\varepsilon,S')$. $\overset{1/T'}{\longmapsto}_1 \iff \mu(a,S) = S''$ et $S'' = \{q'' \mid \exists q \in S : q'' \in \mu(a,q)\}$. $\overset{n/T'}{\longmapsto}_2 \iff S' = \{p \mid \exists q'' \in S'' : (w,q'') \overset{n/T}{\longmapsto} (\varepsilon,p)\}$. En fusionnant S' et S'' $S' = \{p \mid \exists q \in S : q'' \in \mu(a,q), (w,q'') \overset{n/T}{\longmapsto} (\varepsilon,p)\}$. Or $\{p \mid \exists q \in S : q'' \in \mu(a,q)\} \Rightarrow (aw,q) \overset{1/T}{\longmapsto} (\varepsilon,p)$ donc $S' = \{p \mid \exists q \in S : (aw,q) \overset{n+1/T}{\longmapsto} (\varepsilon,p)\}$.

5.2.2 Construction d'un AFD à partir d'un AFN avec ε-transitions

Nous présentons maintenant un algorithme qui construit à partir d'un AFN, un AFD qui reconnaît le même langage. Cet algorithme, appelé souvent *par construction des sous-ensembles*, est utile pour faire simuler un AFN par un programme.

L'idée générale sur laquelle se base la transformation d'un AFN en AFD est que chaque état de l'AFD correspond à un ensemble d'états de l'AFN. Un état courant de l'AFD correspond à l'ensemble des états possibles de l'AFN qui sont atteints avec la même chaîne d'entrée.

La construction d'un AFD à partir d'un AFN se base sur les définitions de l'ε-fermeture, permettant de supprimer les ε-transitions et d'une fonction ξ permettant de déterminer les états accessibles depuis un ensemble d'états par une transition selon un symbole donné de l'alphabet.

Définition 5.2.1 *L'ε-fermeture d'un état $e \in Q$ d'un AFN $T = (\mathcal{A},Q,I,F,\mu)$, notée $\varepsilon{-}fermeture(e)$, est l'ensemble des états de T accessibles depuis l'état e par des ε-transitions seulement.*

Définition 5.2.2 *L'ε-fermeture[1] d'un ensemble d'états $E \subset Q$ d'un AFN $T = (\mathcal{A},Q,I,F,\mu)$, notée $\varepsilon - fermeture(E)$, est l'ensemble des états de T accessibles depuis un état $e \in E$ par des ε-transitions seulement.*

Définition 5.2.3 *La fonction $\xi(a,E)$ est l'ensemble des états E' de T vers lesquels il existe une transition sur le symbole d'entrée a à partir d'un état $e \in E$.*

L'algorithme construit la fonction de transition μ' de l'AFD $T' = (\mathcal{A},Q',I',F',\mu')$. Chaque état de l'AFD est un ensemble d'états de l'AFN $T = (\mathcal{A},Q,I,F,\mu)$ et on construit μ' de telle manière que l'AFD simulera «en parallèle» tous les déplacements possibles que l'AFN peut effectuer sur une chaîne d'entrée donnée. On utilise les opérations définies ci-dessus pour garder trace des ensembles d'états de l'AFN (Fig. 5.1).

On construit Q', I' et μ' de la manière suivante. Chaque état de Q' correspond à un ensemble d'états de T dans lesquels l'automate pourrait se trouver après avoir lu une suite de symboles en entrée en incluant les ε-transitions possibles avant ou après la lecture des symboles. L'état de départ I' est l'ε-fermeture de I. Un état de T' est un état d'acceptation si c'est un ensemble d'états de T qui contient au moins un état d'acceptation.

1. L'ε-fermeture permet de définir la fonction μ^* ([Lan98]).

```
AFN2AFD(↘ T = (A, Q, I, F, μ), ↗ T' = (A, Q', I', F', μ'))
I' ← ε − fermeture(T, I)
Q' ← I'
F' ← ∅
Marque(I') ← Faux
Tant Que ∃E ∈ Q', ¬Marque(E)   Faire
      Marque(E) ← Vrai
      Pour chaque symbole d'entrée a   Faire
            U ← ε − fermeture(T, ξ(a, E))
            Si U ∉ Q'
                  Alors Marque(U) ← Faux
                        Q' ← Q' ∪ U
                        Si ∃q ∈ U, q ∈ F
                              Alors F' ← F' ∪ U
                        Fin Si
            Fin Si
            CreerTransition(μ'(E, q) = U)
      Fin Pour
Fin Tant Que
```

FIG. 5.1 – *Algorithme de passage d'un AFN à un AFD équivalent*

Le calcul de l'ε-fermeture d'un ensemble E est un processus typique de recherche dans un graphe d'un ensemble donné de noeuds. Dans ce cas, les états de E forment un ensemble donné de noeuds et le graphe est constitué uniquement des ε-transitions de l'AFN. Pour calculer l'ε-fermeture, on utilise une pile pour conserver les états dont les transitions n'ont pas encore été examinées (figure 5.2).

```
ε-FERMETURE(↘ T = (A, Q, I, F, μ), ↘ E)
Empiler(Pile, Etats(E))
Fermeture ← E
Tant Que ¬Vide(Pile)   Faire
      t ← Depiler(Pile)
      Pour chaque u ∈ {e | e ∈ Q, μ(ε, t) = e}   Faire
            Si u ∉ Fermeture
                  Alors Fermeture ← Fermeture ∪ u
                        Empiler(Pile, u)
            Fin Si
      Fin Pour
Fin Tant Que
Retourner Fermeture
```

FIG. 5.2 – *Algorithme de l'ε-fermeture*

Comme exemple, prenons l'automate non déterministe de la figure 5.3 reconnaissant $a(a \mid 0)^*$.

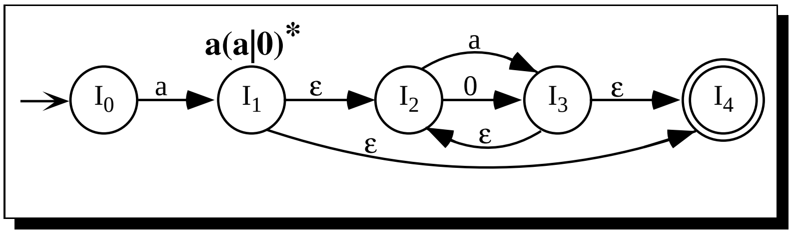


FIG. 5.3 – *Automate non déterministe reconnaissant le langage* $a(a \mid 0)^*$

L'application de l'algorithme par construction des sous-ensembles permet d'obtenir l'automate déterministe de la figure 5.4 reconnaissant le même langage $a(a \mid 0)^*$. Cependant, nous constatons facilement que cet AFD n'est pas minimal (entre autre, C est inutile et les états B et C peuvent être fusionnés).

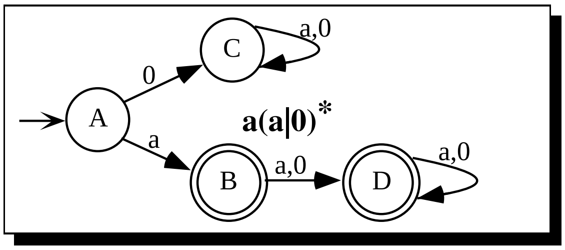


FIG. 5.4 – *Automate déterministe reconnaissant le langage* $a(a \mid 0)^*$

5.3 Minimisation d'un AFN

Certains AFDs possèdent un nombre d'états et de transitions important. États et transitions ne sont pas toujours indispensables. Typiquement, cette situation a lieu lorsque l'automate est construit à l'aide d'algorithmes automatiques comme dans le cas de la construction d'un AFN et de la transformation d'un AFN en AFD. Afin d'améliorer la reconnaissance des chaînes et d'optimiser la place mémoire utilisée, il est intéressant d'essayer de minimiser un AFD. Une première approche de cette minimisation peut consister à étudier les caractéristiques des états et de supprimer ceux qui sont inutiles (définition 4.6.3). Cependant, il est possible d'aller plus loin en garantissant que l'AFD obtenu est un AFD minimal (définition 4.9.5). Ceci est l'objectif de la méthode que l'on présente ici.

Il existe de nombreuses stratégies pour minimiser un AFD [Wat93b][2]. L'algorithme présenté ici est un algorithme, issu de [ASU91], basé sur les classes d'équivalence d'états (figure 5.5).

L'algorithme de minimisation du nombre d'états d'un AFD fonctionne en déterminant tous les groupes d'états qui peuvent être distingués par une chaîne d'entrée. Chaque groupe d'états indistinguables est alors fusionné en un état unique. L'algorithme travaille en mémorisant et en raffinant une partition de l'ensemble des états. Chaque groupe d'états à l'intérieur de la partition

2. Dans cette référence, on trouvera aussi la formalisation et la caractérisation des classes d'équivalence.

Minimiser($\searrow T = (\mathcal{A}, Q, I, F, \mu), \nearrow T' = (\mathcal{A}, Q', I', F', \mu')$)
$Partitionner(\{F; Q - F\}, T, \Pi_f)$ / Fig. 5.6 /
$ConstruireAFDMinimal(\Pi_f, T, T')$ / Fig. 5.7 /
$SimplifierAFDMinimal(T')$ / Fig. 5.8 /

Fig. 5.5 – *Algorithme de réduction d'un AFD*

correspond aux états qui n'ont pas encore été distingués les uns des autres. Toute paire d'états extraits de différents groupes a été prouvée «distinguable» par une chaîne (Fig. 5.6).

Partitionner($\searrow \Pi, \searrow T = (\mathcal{A}, Q, I, F, \mu), \nearrow \Pi_f$)
Répéter
  $\Pi \leftarrow \Pi_n$ /avec $\Pi = \{G_1; G_2; ...; G_n\}$ /
  $\Pi_n \leftarrow \emptyset$
  Pour chaque G_i **dans** Π **Faire**
    Partitionner $G_i \rightarrow Gn_{ki}, k \in [1, m_i]$
    / $\forall (e, f) \in G_i^2, e \in Gn_{ki}, f \in Gn_{ki}$ /
    / $\iff$ /
    / $\forall a \in \mathcal{A}, e' = \mu(a, e), f' = \mu(a, f), (e', f') \in G_l^2$ /
    $\Pi_n \leftarrow \Pi_n \cup \{Gn_{k1}; ...; Gn_{km_i}\}$
  Fin Pour
Jusqu'à $\Pi = \Pi_n$
$\Pi_f \leftarrow \Pi$ /avec $\Pi_f = \{Gf_1; Gf_2; ...; Gf_n\}$ /

Fig. 5.6 – *Algorithme de partitionnement pour la minimisation*

Initialement, la partition consiste à former deux groupes : les états d'acceptation et les autres. L'étape fondamentale prend un groupe d'états et un symbole puis étudie les transitions de ces états sur ce symbole. Si ces transitions conduisent à des états qui tombent dans au moins deux groupes différents de la partition courante, alors on doit diviser ce groupe. La division est effectuée avec pour objectif que les transitions depuis chaque sous-groupe soient confinées à un seul groupe de la partition courante. Ce processus de division est répété jusqu'à ce qu'aucun groupe n'ait plus besoin d'être divisé.

Les états du nouvel automate sont donnés par un représentant de chaque groupe. Les états finaux sont les représentants des groupes possédant un état final de l'AFD de départ. L'état de départ est le représentant du groupe possédant l'état de départ de l'AFD initial. Les transitions sont construites avec les transitions de chaque état représentant de groupe vers un état représentant d'un autre groupe si cet état engendre une transition vers un élément du groupe (Fig. 5.7).

Sur les états créés, il faut supprimer les états stériles non finaux et les états non-accessibles depuis l'état initial (Fig. 5.8).

Théorème 5.3.1 *L'automate T' obtenu par cet algorithme est minimal (Déf. 4.9.5).*

ConstruireAFDMinimal($\searrow \Pi_f, \searrow T = (\mathcal{A}, Q, I, F, \mu), \nearrow T' = (\mathcal{A}, Q', I', F', \mu')$)
$Q' \leftarrow \emptyset$
$F' \leftarrow \emptyset$
Pour chaque Gf_i **dans** Π_f **Faire**
  Prendre un $e_i \in Gf_i$
  $Q' \leftarrow Q' \cup e_i$
  Si $\exists q \in Gf_i : q \in F$
    Alors $F' \leftarrow F' \cup e_i$
  Fin Si
  Si $I \in Gf_i$
    Alors $I' \leftarrow e_i$
  Fin Si
Fin Pour
/avec $Q' = \{e_1, e_2, ..., e_n\}$ /
Pour chaque e_i **dans** Q' **Faire**
  Si $\exists a \in \mathcal{A}, \exists t \in Gf_j, j \in [1, n] : \mu(a, e_i) = t$
    Alors $CreerTransition(\mu'(a, e_i) = e_j)$
  Fin Si
Fin Pour

Fig. 5.7 – *Algorithme de construction de l'AFD minimal*

SimplifierAFDMinimal($\searrow\nearrow T' = (\mathcal{A}, Q', I', F', \mu')$)
Répéter
  $Modifications \leftarrow Faux$
  Pour chaque e_i **dans** Q' **Faire**
    Si $e_i \notin F'$ et $\forall w \in \mathcal{A}, \mu'(w, e_i) = e_i$
      Alors $Q' \leftarrow Q' - e_i$
        $SupprimerTransition(\mu'(a, q) = e_i), \forall a \in \mathcal{A}, \forall q \in Q'$
      Sinon Si $\nexists w \in \mathcal{A}^*, \mu^*(w, I') = e_i$
        Alors $Q' \leftarrow Q' - e_i$
          $SupprimerTransition(\mu'(a, q) = e_i),$
          $\forall a \in \mathcal{A}, \forall q \in Q'$
      Fin Si
    Fin Si
  Fin Pour
Jusqu'à $\neg Modifications$

Fig. 5.8 – *Algorithme de simplification de l'AFD minimal*

Dém. Supposons que $\exists T'' = (\mathcal{A}, Q'', I'', F'', \mu''), \mid T'' \mid < \mid T' \mid, L(T'') = L(T')$.
$\forall q \in Q', Accessible(q)$.
$\exists w, x \in \mathcal{A}^*, (w, I'') \overset{*/T''}{\longmapsto} (\varepsilon, q)$ et $(x, I'') \overset{*/T''}{\longmapsto} (\varepsilon, q)$.
Mais w et x conduisent T' à des états différents. Il s'en suit que ces deux chaînes conduisent

T à des états différents p et q qui sont distingables, c'est-à-dire $\exists y \in \mathcal{A}^*$, exactement une des chaînes xy et $wy \in L(T)$. Mais, ces chaînes doivent conduire T'' au même état, c'est-à-dire l'état $s : (y,q) \overset{*/T''}{\longmapsto} (\varepsilon, s)$. Donc $(wy, I'') \overset{*/T''}{\longmapsto} (y,q) \overset{*/T''}{\longmapsto} (\varepsilon, s)$ et $(xy, I'') \overset{*/T''}{\longmapsto} (y,q) \overset{*/T''}{\longmapsto} (\varepsilon, s)$. Il n'est donc pas possible qu'une seule chaîne wy et $xy \in L(T'')$ donc à $L(T)$. Donc T'' n'existe pas.

Si on applique cet algorithme à l'automate déterministe de l'exemple de la section précédente (fig. 5.4), on obtient l'automate déterministe de la figure 5.9.

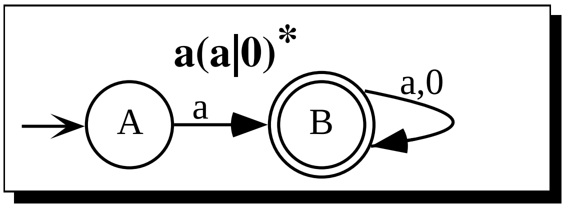


FIG. 5.9 – *Automate déterministe minimal reconnaissant le langage* $a(a \mid 0)^*$

5.4 Expressions régulières et automates finis

5.4.1 Présentation

Dans [Kle56], Kleene démontre le premier résultat de cette théorie. C'est à la fois le premier dans l'ordre chronologique mais aussi dans l'ordre d'importance. Il s'agit d'une sorte de théorème de représentation qui établit l'équivalence entre deux modes de définition du même objet. Ces objets sont ce que Kleene appelle les *événements réguliers*, aujourd'hui souvent appelés *langages* plutôt qu'événements et qualifiés de *rationnels* plutôt que de réguliers. Le terme d'événement renvoie bien sûr à la terminologie du calcul des probabilités mais aussi au point de vue de départ de cette théorie qui s'adresse à la *description* de phénomènes plutôt qu'à la *spécification* de propriétés formelles, comme de nos jours.

Les événements réguliers sont ceux que l'on peut décrire à partir d'événements de base en utilisant les opérateurs de :

1. l'union ensembliste (équivalent au 'ou' logique) ;

2. la concaténation ;

3. l'itération (notée par une étoile $*$).

Théorème 5.4.1 *(Théorème de Kleene) Les événements, que l'on peut décrire à partir d'événements de base en utilisant les trois opérateurs d'union, de concaténation et d'itération, sont exactement ceux que l'on peut spécifier à l'aide d'un automate fini.*

Grâce à ce théorème, il est possible de déduire qu'à tout automate fini il est possible d'associer une grammaire régulière droite.

Théorème 5.4.2 *Soit T un AFN tel que $L(T) = L$ alors il existe une grammaire linéaire droite LD telle que $L = L(LD)$.*

Dém. (par construction) Une dérivation de la grammaire correspond à un mouvement de l'automate. Soit $T = (\mathcal{A}, Q, I, F, \mu)$, on pose $G = (\mathcal{A}, Q, I, S)$ où S est défini par :

- si $\mu(a, q) = r$ alors $q \to ar \in S$;
- si $p \in F$ alors $p \to \varepsilon \in P$.

De même, il est possible de construire un AFN reconnaissant un langage régulier par induction à partir des symboles et des trois opérateurs de base.

Théorème 5.4.3 *Soit L_1 un langage décrit par l'AFN T_1 et L_2 celui décrit par T_2. Alors $L_1 \cup L_2$, $L_1 \oplus L_2$ et L_1^* sont des langages reconnus par des AFNs.*

Dém. (par construction) Soient $T_1 = (\mathcal{A}, Q_1, I_1, F_1, \mu_1)$ et $T_2 = (\mathcal{A}, Q_2, I_2, F_2, \mu_2)$ tels que $Q_1 \cap Q_2 = \emptyset$ (éventuellement en renommant) et $a \in \mathcal{A}$:

1. Pour $L_1 \cup L2$: Soit $T = (\mathcal{A}, Q_1 \cup Q_2 \cup \{q_I\}, q_I, F, \mu)$ où $F = F_1 \cup F_2$ si $\varepsilon \notin L_1 \cup L2$ ou $F = F_1 \cup F_2 \cup \{q_I\}$ sinon et $\mu = \mu_1 \cup \mu_2$ et $\mu(a, q_I) = \mu_1(a, I_1) \cup \mu_2(a, I_2), \forall a \in \mathcal{A}$.
2. Pour $L_1 L_2$: Soit $T = (\mathcal{A}, Q_1 \cup Q_2, I_1, F, \mu)$ où $\mu(a, q) = \mu_1(a, q), \forall q \in Q_1$
   F_1, $\mu(a, q) = \mu_1(a, q) \cup \mu_2(a, q), \forall q \in F_1$ et $\mu(a, q) = \mu_2(a, q), \forall q \in Q_2$, et, si $I_2 \notin F_2, F = F_2$ et $F = F_1 \cup F_2$ sinon.
3. Pour L_1^* : Soit $T = (\mathcal{A}, Q_1 \cup \{I\}, I, F_1 \cup \{I\}, \mu)$ où $\mu(a, q) = \mu_1(a, q)$ si $q \in Q_1$
   F_1, $\mu(a, q) = \mu_1(a, q) \cup \mu_1(a, I_1)$ si $q \in F_1$ et $\mu(a, I) = \mu_1(a, I_1)$.

Théorème 5.4.4 *(Conséquence du théorème de Kleene) Chaque ensemble régulier est reconnu par un AFD à nombre d'états minimum qui est unique à un renommage des états près et réciproquement. Autrement dit, un langage est accepté par un AFD si et seulement si c'est un langage rationnel.*

Pour effectuer le passage d'une expression régulière à un automate fini déterministe minimal, une méthode possible consiste à construire d'abord un AFN reconnaissant le même langage, de le transformer en un AFD équivalent puis, finalement, de le réduire (Fig. 5.10). Pour les deux dernières phases, nous avons déjà présenté une méthode adaptée. Intéressons-nous désormais à la phase de construction de l'AFN à partir d'une expression régulière.

Expression régulière $\overset{Production}{\models\circlearrowright\Longrightarrow}$ AFN $\overset{Transformation}{\models\circlearrowright\Longrightarrow}$ AFD $\overset{Minimisation}{\models\circlearrowright\Longrightarrow}$ AFD minimal

FIG. 5.10 – *Passage d'une expression régulière à un AFD minimal*

5.4.2 Construction d'un AFN à partir d'une expression régulière

Il existe de nombreuses stratégies pour construire un reconnaisseur à partir d'une expression régulière [Wat93a][3]. L'algorithme présenté ici est l'algorithme de construction de Thompson [Tho68]. Plusieurs variantes sont possibles. Celle présentée ici est simple et facile à implanter.

L'algorithme est dirigé par la syntaxe, c'est-à-dire qu'il utilise la structure syntaxique de l'expression régulière pour guider le processus de construction.

L'algorithme est récursif sur l'arbre syntaxique de l'expression régulière (figure 5.11). A partir d'une expression régulière r sur $\mathcal{A}$ on cherche à construire un AFN $N(r)$ qui reconnaît $L(r)$.

Thompson($\searrow r, \searrow \mathcal{A}_r, \nearrow T = (\mathcal{A}, Q, I, F, \mu)$)
$\mathcal{A} \leftarrow \mathcal{A}_r$
Suivant
  $r = \varepsilon$ **:** $(Cf.fig.\ 5.12(1))$
    $Q \leftarrow \{i, f\}; I \leftarrow i; F \leftarrow \{f\}$
    $CreerTransition(\mu(\varepsilon, i) = f)$
  $r = a, a \in \mathcal{A}_r$ **:** $(Cf.fig.\ 5.12(2))$
    $Q \leftarrow \{i, f\}; I \leftarrow i; F \leftarrow \{f\}$
    $CreerTransition(\mu(a, i) = f)$
  $r = s \mid t$ **:** $(Cf.fig.\ 5.12(3))$
    $Thompson(s, \mathcal{A}_r, T_s = (\mathcal{A}, Q_s, I_s, F_s = \{f_s\}, \mu_s))$
    $Thompson(t, \mathcal{A}_r, T_t = (\mathcal{A}, Q_t, I_t, F_t = \{f_t\}, \mu_t))$
    $Q \leftarrow \{i, f\} \cup Q_s \cup Q_t; I \leftarrow i; F \leftarrow \{f\}; \mu \leftarrow \mu_s \cup \mu_t$
    $CreerTransition(\mu(\varepsilon, i) = I_s); CreerTransition(\mu(\varepsilon, i) = I_t)$
    $CreerTransition(\mu(\varepsilon, f_s) = f); CreerTransition(\mu(\varepsilon, f_t) = f)$
  $r = st$ **:** $(Cf.fig.\ 5.12(4))$
    $Thompson(s, \mathcal{A}_r, T_s = (\mathcal{A}, Q_s, I_s, F_s = \{f_s\}, \mu_s))$
    $Thompson(t, \mathcal{A}_r, T_t = (\mathcal{A}, Q_t, I_t, F_t = \{f_t\}, \mu_t))$
    $I \leftarrow I_s; F \leftarrow \{f_t\}; Q \leftarrow Q_s \cup Q_t; I_t \leftarrow f_s; \mu \leftarrow \mu_s \cup \mu_t$
    $ModifierTransition(\mu(x, I_t) = q, \mu(x, f_s) = q)$
  $r = s^*$ **:** $(Cf.fig.\ 5.12(5))$
    $Thompson(s, \mathcal{A}_r, T_s = (\mathcal{A}, Q_s, I_s, F_s = \{f_s\}, \mu_s))$
    $Q \leftarrow \{i, f\} \cup Q_s; I \leftarrow i; F \leftarrow \{f\}; \mu \leftarrow \mu_s$
    $CreerTransition(\mu(\varepsilon, i) = f); CreerTransition(\mu(\varepsilon, i) = I_s)$
    $CreerTransition(\mu(\varepsilon, f_s) = f); CreerTransition(\mu(\varepsilon, f_s) = I_s)$
  $r = (s)$ **:** $Thompson(s, \mathcal{A}_r, T_s = T = (\mathcal{A}, Q, I, F, \mu))$
Fin Suivant

Fig. 5.11 – *Algorithme de Thompson*

Si l'expression est réduite à un symbole s alors on construit l'automate fig. 5.12(1) pour $s = \varepsilon$ ou fig. 5.12(2) pour tout symbole $s = a \in \mathcal{A}$. Cet automate reconnaît alors le langage $\{s\}$.

Si l'expression n'est pas un symbole alors la décomposer selon l'opérateur utilisé puis construire les automates $N(s)$ et $N(t)$ associés aux opérandes s et t.

3. Cette référence présente de manière formelle différents algorithmes dont celui de Thompson.

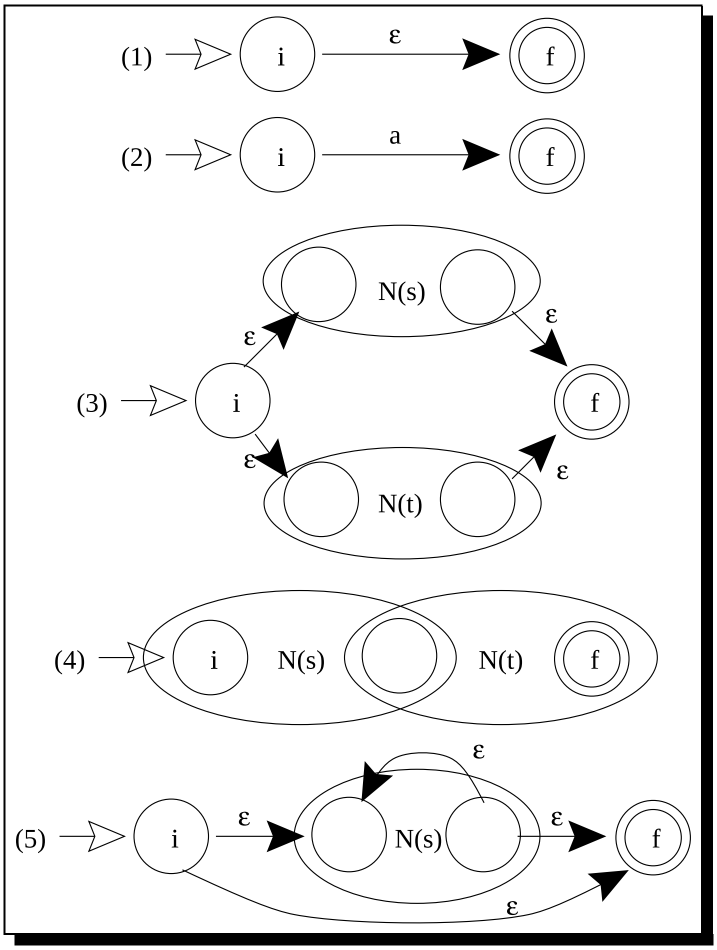


FIG. 5.12 – *Construction d'un AFN par la méthode de Thompson*

Selon le type d'expression :

- $s \mid t$: on construit l'automate $N(s \mid t)$ fig. 5.12(3) où i est un nouvel état de départ et f un nouvel état d'acceptation. Il y a une ε-transition depuis i vers les états de départ de $N(T)$ et $N(s)$ et depuis $N(s)$ et $N(t)$ vers f. Les états de départ et d'acceptation de $N(s)$ et $N(t)$ sont des états de transition pour $N(s \mid t)$. Tout chemin depuis i vers f doit traverser soit $N(s)$ soit $N(t)$ exclusivement donc l'automate composé reconnaît $L(s) \cup L(t)$.

- st : on construit l'automate $N(st)$ fig. 5.12(4) où l'état de départ de $N(s)$ devient celui de $N(st)$ et l'état d'arrivée de $N(t)$ devient celui de $N(st)$. L'état d'acceptation de $N(s)$ et celui de départ de $N(t)$ sont fusionnés. Cet état n'est qu'un état de transition. Un chemin depuis i vers f doit d'abord traverser $N(s)$ puis $N(t)$ sans retour en arrière possible. Donc, $N(st)$ reconnaît le langage $L(s)L(t)$.

- s^* : on construit l'automate $N(s^*)$ fig. 5.12(5) où i est un nouvel état de départ et f un nouvel état d'acceptation. On peut aller soit directement de i à f en suivant un arc étiqueté ε qui représente le fait que $\varepsilon \in L(s^*)$ soit aller de i à f en traversant $N(s)$ une ou plusieurs fois. Cet automate reconnaît le langage $L(s^*)$.

- (s) : on construit l'automate $N((s)) = N(s)$.

Chaque fois que l'on construit un nouvel état, on lui donne un nom distinct pour qu'il n'y ait pas deux états de même nom. Même si un symbole apparaît plusieurs fois dans r, on crée pour chaque symbole un AFN séparé avec ses propres états. Chaque étapes de la construction produit un AFN qui reconnaît le langage correct.

L'AFN ainsi produit possède les propriétés suivantes :

- $N(r)$ possède au plus deux fois plus d'états qu'il n'y a de symboles et d'opérateurs dans r.
- $N(r)$ a exactement un état de départ et un état d'acceptation.
- Chaque état de $N(r)$ a soit une transition sortante sur un symbole de $\mathcal{A}$ soit au plus deux ε-transitions sortantes.

L'application de cet algorithme sur l'expression régulière $(a \mid b)^*abb$ donne l'AFN de la figure 5.16.

Remarque : il est possible de construire directement un AFD à partir d'une expression régulière dite «étendue», c'est-à-dire à laquelle on ajoute en fin le caractère #. Nous n'aborderons pas cette méthode ici (voir [ASU91], pp.159–165).

5.4.3 Simulation directe d'un AFN issu d'une expression régulière

Il est possible d'effectuer une reconnaissance de chaîne donnée directement à partir de l'AFN construit à partir d'une expression régulière par la méthode de Thompson (Fig. 5.13).

ReconnaîtreLangage2($\searrow T(\mathcal{A}, Q, I, F, \mu), \searrow Chaine$)
$EtatsCourants \leftarrow \varepsilon - fermeture(T, I)$
$Ch \leftarrow PremierCaractere(Chaine)$
Tant Que $Ch \neq \varepsilon$ et $EtatsCourants \neq \emptyset$ **Faire**
  $EtatsCourants \leftarrow \varepsilon - fermeture(T, \xi(Ch, EtatsCourants)$
  $Ch \leftarrow CaractereSuivant(Chaine)$
Fin Tant Que
Si $EtatsCourants \cap F \neq \emptyset$ et $Ch = \varepsilon$
  Alors Retourner $Vrai$
  Sinon Retourner $Faux$
Fin Si

Fig. 5.13 – *Algorithme de reconnaissance d'une chaîne par un AFN(r)*

Cet algorithme construit dynamiquement les sous-ensembles. Il calcule une transition depuis l'ensemble courant d'états vers le prochain ensemble d'états en deux étapes. D'abord, il détermine l'ensemble de tous les états accessibles depuis l'ensemble courant par une transition sur le caractère courant. Ensuite, il calcule l'ε-fermeture, c'est-à-dire tous les états qui peuvent être atteints par zéro ou plusieurs ε-transitions.

5.4.4 Suite d'expressions régulières

Nous venons de présenter une méthode permettant de passer d'une expression régulière décrivant un langage à un automate déterministe minimal le reconnaissant. Cependant, il est souvent plus facile de définir un langage à l'aide d'un ensemble d'expressions plutôt qu'avec une seule et unique. Pour cela, il est possible de donner un ensemble de définitions régulières. Le langage décrit est alors l'union des langages définis par chacune des définitions.

Par exemple, supposons un langage $\mathcal{L}$ composé d'identificateurs et d'entiers, il y aura alors deux définitions régulières (elles-mêmes basées sur deux classes de symboles). Avec les deux classes «$Ch \rightarrow 0 \mid 1 \mid ... \mid 9$» et «$Ltr \rightarrow A \mid ... \mid Z \mid a \mid ... \mid z$», les définitions pourraient être :

- $Id \rightarrow Ltr(Ltr \mid Ch)^*$;
- $Ent \rightarrow ChCh^*$.

Nous allons maintenant présenter une méthode pour regrouper les différents automates associés à chacune des définitions.

Soit une suite de définitions régulières de la forme :

- $A_1 \rightarrow R_1$;
- $A_2 \rightarrow R_2$;
- ...
- $A_n \rightarrow R_n$.

La génération pour cet ensemble d'expressions régulières passera par les étapes suivantes :

1. La production des AFDs $M_i = (\mathcal{A}_i, Q_i, I_i, F_i, \mu_i)$ pour les expressions régulières R_i (algorithme de Thompson) avec des ensembles Q_i disjoints.

2. La construction de l'AFD $M = (\mathcal{A}, Q, I, F, \mu)$, union des M_i. Cet AFD est donc obtenu par la mise en commun des différents AFDs, mis en relation par un nouvel état initial I et un ensemble d'ε-transitions depuis I vers les états initiaux des automates (la figure 5.14 présente cet AFD pour l'exemple du langage $\mathcal{L}$). M est tel que :
   - $\mathcal{A} = \bigcup_{i=1}^{n} \mathcal{A}_i$,
   - $Q = \bigcup_{i=1}^{n} Q_i \cup \{I\}, I \notin \bigcup_{i=1}^{n} Q_i$,
   - $F = \bigcup_{i=1}^{n} F_i$,
   - $\mu = \bigcup_{i=1}^{n} \mu_i \cup \{\mu(\varepsilon, I) = I_i, i \in [1..n]\}$.

3. L'application de l'algorithme par construction des sous-ensembles sur M permettant de le rendre déterministe. On obtient $M' = AFN2AFD(M)$.

4. L'application de l'algorithme de minimisation sur M' avec comme partition initiale : $\Pi = \{F'_1, F'_2, ..., F'_n, Q' - \bigcup_{i=1}^{n} F'_i\}$ où les F'_i sont les états finaux dans M' correspondant aux classes A_i tels que : $F'_i = \{s \mid s \cap F'_i \neq \emptyset \text{ et } s \cap F'_j = \emptyset, j \leq i-1\}$. Autrement dit, si un symbole amène l'automate dans un état final pour M_i, on lui associe la classe apparue plus tôt dans la liste des définitions.

La figure 5.15 présente l'automate finalement obtenu en appliquant cette méthode sur l'exemple du langage $\mathcal{L}$.

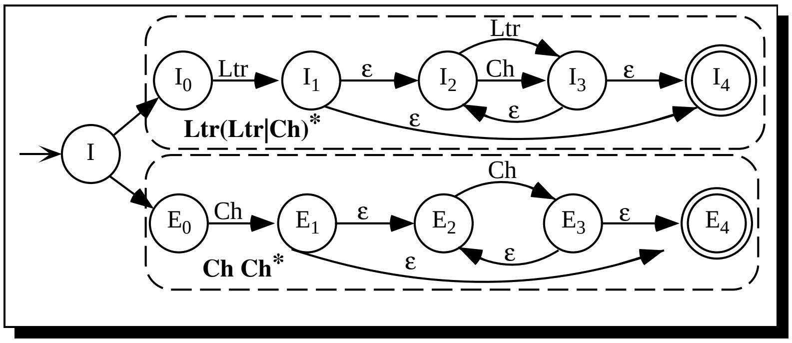


FIG. 5.14 – *Définition régulière concernant un identificateur et un entier*

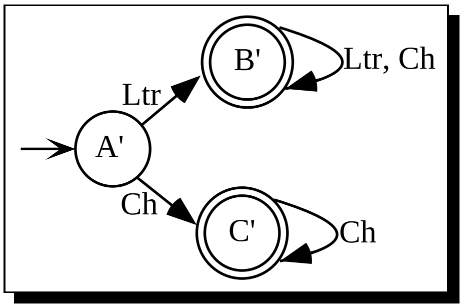


FIG. 5.15 – *Automate minimal reconnaissant les définitions régulières de la figure 5.14*

5.5 Illustration

Nous venons donc de proposer un ensemble d'algorithmes permettant de passer d'une expression régulière à un AFN puis d'un AFN à un AFD minimal.

Tout d'abord, l'application de l'algorithme de Thompson sur l'expression $\{(a \mid b)^*abb\}$ est illustrée par la figure 5.16[4].

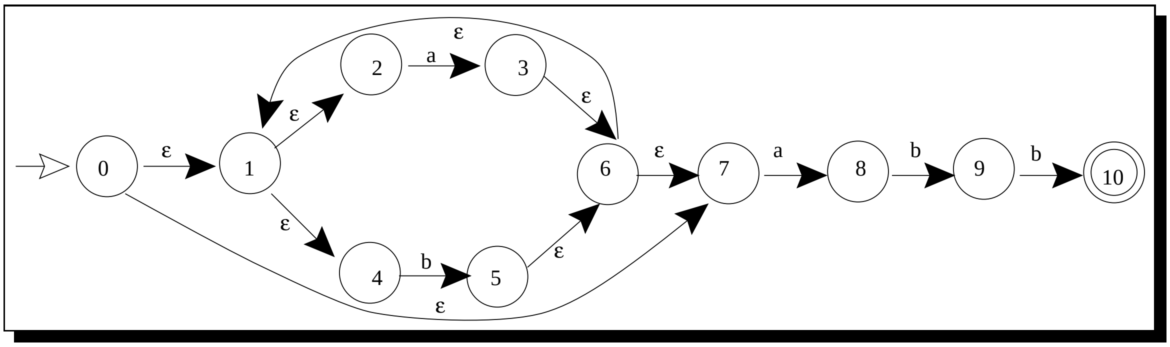


FIG. 5.16 – *Un AFN reconnaissant les chaînes du langage* $\{(a \mid b)^*abb\}$

Ensuite, pour cet exemple, le processus de conversion d'un AFN en AFD minimal est illustré par le passage de l'AFN de la figure 5.16 à l'AFD équivalent de la figure 5.17 puis, par minimisation, à celui de la figure 5.18.

4. Cet exemple est tiré de [ASU91].

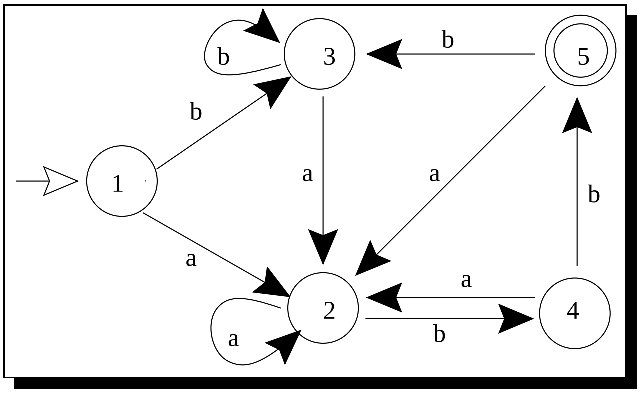


FIG. 5.17 – *AFD équivalent à l'AFN de la figure 5.16*

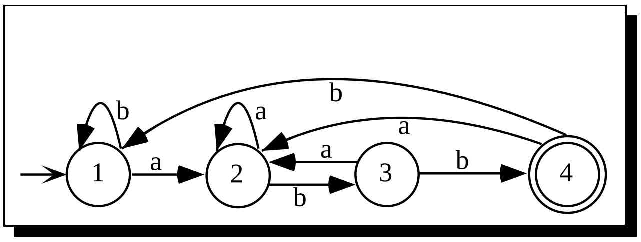


FIG. 5.18 – *AFD minimal équivalent à l'AFD de la figure 5.17*

De même, la figure 5.3 présente l'AFN déduit de l'expression régulière $a(a \mid 0)^*$. Puis la figure 5.4 présente l'automate déterministe équivalent et, finalement, la figure 5.9 l'automate déterministe minimal.

5.6 Conclusion

Finalement, au cours de ces chapitres, nous avons présenté le passage entre différentes représentations d'une même entité : les langages réguliers. La figure 5.19 présente ces relations avec le passage :

1. d'une expression régulière r à un automate fini (AFN et même AFD) $T = (\mathcal{A}, Q, I, F, \mu)$ (Fig 5.19(1)) par l'algorithme de Thompson (§ 5.4.2);

2. d'un système d'équations régulières S à une expression régulière r (Fig 5.19(2)) par la résolution du système (§ 2.2.6) ;

3. d'un automate $T = (\mathcal{A}, Q, I, F, \mu)$ à une grammaire régulière $G = (\mathcal{A}, Q, S, R)$ (Fig 5.19(3)) par une conversion sur la fonction de transition :
   $\{\mu(a_i, q_j) = q_i, \forall i \in [1, n]\} \iff q_j \to a_1 q_1 \mid ... \mid a_n q_n \in R$, $q = I \iff q = S$ et $q \in F \iff q \to \varepsilon$;

4. d'une grammaire régulière $G = (\mathcal{A}, Q, S, R)$ à un système d'expressions régulières S (Fig 5.19(4)) par une conversion des règles (§ 2.2.6 et 2.4) :
   $A \to rA \mid sB \mid t \in R \iff X = rX + sY + t \in S$;

5. d'un automate $T = (\mathcal{A}, Q, I, F, \mu)$ à un système d'expressions régulières S (Fig 5.19(5)) par la fonction de transition :
$\{\mu(a_i, q_j) = q_i, \forall i \in [1, n]\} \iff X_j = a_1X_1 + a_2X_2 + ... + a_nX_n \in S$;

6. d'un automate $T = (\mathcal{A}, Q, I, F, \mu)$ à une expression régulière r (Fig 5.19(6)) par une méthode de réduction d'automate en assignant aux arcs non plus un symbole mais une expression rationnelle (voir [Lan98] «Lecture 10»). Ce passage d'un automate à une expression régulière peut aussi être effectué par induction sur les transitions (voir [Sha97]).

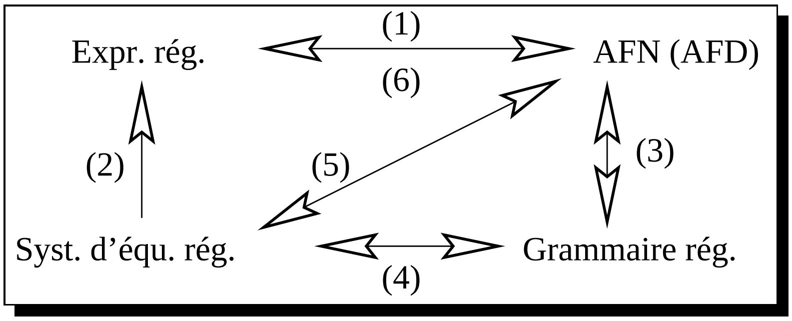


Fig. 5.19 – *Bilan des conversions*

Chapitre 6

Applications des automates

6.1 Introduction

Dans les chapitres précédents, nous avons présenté une introduction à la théorie des langages et des automates. Ces notions ne sont pas seulement des éléments théoriques abstraits mais sont aussi des outils performants trouvant leur utilité dans de nombreux de domaines. Nous allons présenter ici une utilisation dans le domaine du traitement de langages à syntaxe contrainte (langages de programmation) ou souple (langage naturel). Dans un premier temps, nous étudierons la gestion des langages, et surtout l'utilisation des automates, dans le cadre du processus de transformation d'un programme formulé dans un langage de programmation donné vers un langage compréhensible par une machine : c'est le processus de compilation. Puis, nous donnerons quelques éléments liés à l'analyse du langage naturel. Enfin, nous terminerons par la présentation d'autres domaines d'application.

6.2 La compilation

La compilation est l'étude de la traduction de programmes écrits dans un langage de programmation (C, PASCAL, FORTRAN...), en des programmes équivalents écrits dans un langage de plus bas niveau (machine ou cible).

6.2.1 Éléments principaux

Le langage

Un langage est formé d'un ensemble de phrases qui permet de désigner des objets d'un certain domaine sémantique ou univers et des actions portant sur ces objets.

Une phrase est un assemblage de mots appelés symboles dont l'ensemble constitue le vocabulaire du langage. Chaque mot ou symbole résulte de la concaténation d'une suite finie de signes ou caractères. Un symbole est la plus petite sous-séquence de signes chargée de sens que l'on peut classer suivant trois catégories :

1. ceux qui désignent des objets du domaine sémantique ;
2. ceux qui désignent des relations ou des applications sur le domaine sémantique ;
3. ceux qui servent pour la ponctuation ou la coordination de parties de phrases.

Dans le cadre de l'informatique, de nombreux logiciels admettent en entrée des langages soumis à des règles d'écritures précises appelées règles syntaxiques.

Par exemple pour un SGBD, le langage d'interrogation d'une base de données permet de gérer les éléments de la base. Le logiciel doit contrôler la validité des commandes d'interrogation de la base (analyse syntaxique) puis est amené à exécuter des traitements spécifiques, ou actions, qui entraînent la recherche dans la base d'informations pour construire la réponse.

Dans ce cas, l'analyseur syntaxique dirige ou "pilote" le logiciel et on parle de programmation dirigée par la syntaxe. C'est une méthode de programmation en tant que telle.

Le compilateur

De façon générale, un compilateur est un programme qui lit en entrée un programme écrit dans un langage appelé «langage source». Puis, il le traduit dans un programme équivalent dans un langage différent appelé langage cible. La compilation d'un programme se déroule en deux parties successives : l'analyse et la synthèse (fig. 6.1).

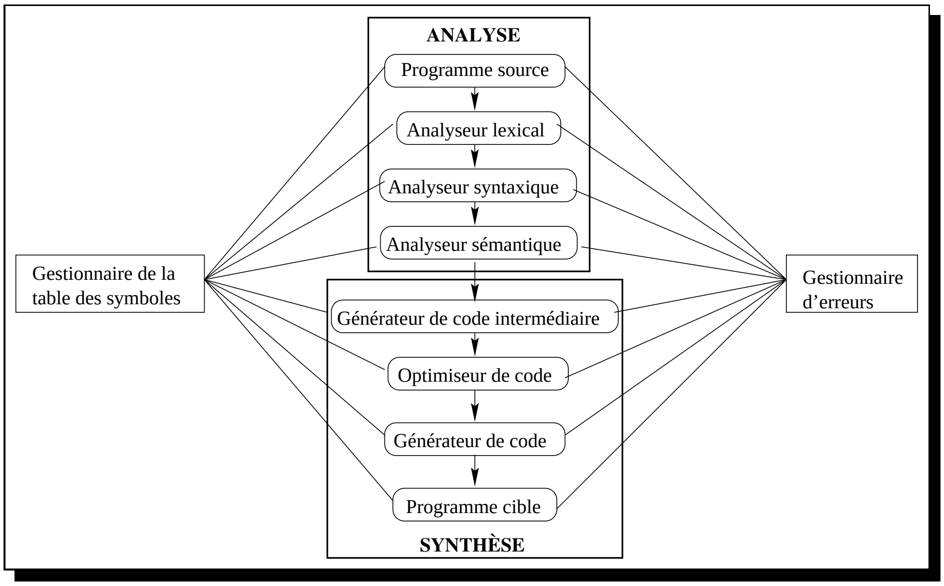


FIG. 6.1 – *Différentes phases d'un compilateur*

L'analyse. Elle permet de trouver la structure du programme à partir du texte du programme, c'est-à-dire d'une suite de caractères. L'analyse est uniquement dépendante du langage à traduire (indépendante du langage cible). Nous distinguons les trois phases d'analyse suivantes (fig. 6.1) :

- l'analyse lexicale qui lit la chaîne de caractères d'entrée, caractère par caractère, et la transforme en une suite de mots ou unités lexicales ;
- l'analyse syntaxique qui vérifie si la suite d'unités lexicales reconnues par l'analyseur lexical est légale. L'analyse syntaxique dérive une structure hiérarchique sur le flot d'unités lexicales qui peut être représentée par des arbres abstraits ;
- l'analyse sémantique qui contrôle si le programme source a un sens. L'un des rôles importants de l'analyse sémantique est, en particulier, le contrôle de type : on vérifie si les opérandes de

chaque opérateur sont conformes aux spécifications du langage source (opérations arithmétiques entre nombres entiers, ou réels...).

Au cours de ce processus d'analyse, un rôle important du compilateur est de signaler à son utilisateur, la présence d'erreurs dans le programme source via des messages d'erreurs. Par exemple, les erreurs peuvent être :

- lexicales, comme l'écriture erronée d'un identificateur, d'un mot clé ou d'un opérateur ;
- syntaxiques, comme une expression mathématique mal parenthèsée ;
- sémantiques, comme un opérateur appliqué à un opérande incompatible ;
- logiques, comme un appel récursif infini.

La synthèse. Après l'analyse du programme source vient la partie synthèse. Elle permet de traduire la structure du programme en prenant en compte les possibilités du langage cible (optimisation). Elle dépend du langage cible (choix des constructions machines, ressources disponibles...) et de la définition du langage compilé (c'est-à-dire du sens associé aux structures du programme : la sémantique du langage).

Elle peut se décomposer selon les phases suivantes (fig. 6.1) :

- production de code intermédiaire : certains compilateurs construisent explicitement une représentation intermédiaire qui peut être considérée comme un programme pour une machine abstraite (machine virtuelle) ;
- optimisation de code : on tente ici d'améliorer le code intermédiaire de telle sorte que le code machine s'exécute le plus rapidement possible ;
- production du code cible qui est un code machine translatable ou un code d'assemblage. On cherche en particulier ici à gérer au mieux l'assignation des variables aux registres de la machine.

Par exemple : Pascal (vers du P-CODE ou du langage machine), C, Ada...

L'interpréteur

La production d'un code machine destiné à être exécuté par le système procure une grande rapidité d'exécution parce qu'il peut être optimisé en exploitant au mieux les caractéristiques de l'architecture de la machine sur lequel il s'exécute. Mais cette approche révèle certains inconvénients :

- faible degré de portabilité : le code machine peut varier d'un type d'ordinateur à un autre ;
- contrôle assuré directement par le système à l'exécution (peu d'interactivité) ;
- difficulté de mise au point.

Au lieu de produire un programme cible comme dans une technique de traduction normale, l'interprète effectue lui-même les opérations spécifiées par le programme source. Il exécute (interpréte) un code intermédiaire qui est indépendant d'un code machine donné. L'interprète peut être écrit dans un langage évolué suffisamment universel pour que le changement de type d'ordinateur ne pose pas de trop gros problèmes. Si l'inconvénient majeur de l'interprète est sa lenteur, ses avantages sont sa portabilité et sa souplesse à l'exécution. En effet, l'utilisateur peut plus facilement intervenir durant l'exécution pour modifier son cours.

Par exemple : Basic, Lisp, Caml, Clips, Perl, Prolog, P-CODE...

Le compilateur Java génère du *byte-code* (indépendant de toute architecture) et non du code machine. Pour vraiment exécuter un programme Java, il faut utiliser un interpréteur qui exécute le byte-code généré. Un programme Java peut être exécuté sur tout système implémentant l'interpréteur Java et le runtime Java. Cet interpréteur et ce runtime implémentent à eux deux la *machine virtuelle Java* (ou JVM pour "Java Virtual Machine"). De ce fait, l'exécution d'un programme en Java est 20 fois plus lente qu'un programme équivalent en C. Cependant, la compilation en byte-code permet d'atteindre des temps d'exécution bien supérieurs aux programmes écrits en langages de scripts comme Tcl, Perl ou les shells UNIX.

Néanmoins, il existe des moyens de donner aux programmes en Java des performances très proches de celles d'un programme en C. Outre l'apparition de processeurs Java, il existe des compilateurs «just in time» (ie. "qui compilent au dernier moment") capables de compiler le byte-code Java en code machine dédié à un processeur particulier et ce en cours d'exécution. Cette génération de code machine est relativement simple à réaliser (prévue dès la conception du byte-code) et produit un code raisonnablement efficace. Ces compilateurs permettent de générer du code machine soit durant la génération du programme soit au moment de l'exécution. Dans ce dernier cas, la compilation s'effectue au premier passage dans le code en parallèle de la première exécution. Les passages suivants se font directement sur le code machine.

6.2.2 L'analyse lexicale

Principe

L'analyse lexicale ou lexicographique permet de déterminer les unités lexicales qui ont un sens. Cette analyse se base sur une grammaire. A l'aide de celle-ci, et donc du langage engendré, il est possible de mettre en place une procédure d'analyse d'une chaîne donnée. Sachant que la plupart des grammaires utilisées en analyse lexicale sont régulières, il est possible d'utiliser un automate fini pour cette analyse.

L'analyse lexicale est donc la reconnaissance d'un langage régulier (le plus souvent) et le codage des mots. Elle permet de découper le texte d'entrée en une suite de symboles.

Cette analyse utilise des expressions régulières, définissant les symboles (lexèmes) du langage, converties en un AFD minimal unique et un ensemble de couples : type de symbole / état final. L'analyse lexicale fait tourner l'AFD jusqu'à un état final et que le prochain caractère mène à une erreur.

Remarque : L'analyse lexicale fonctionne selon le principe de la plus grande correspondance : l'analyseur cherche à reconnaître la plus longue chaîne.

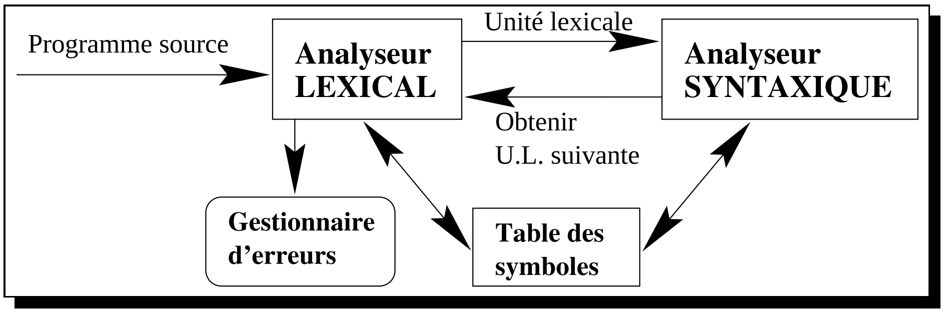


FIG. 6.2 – *Rôle de l'analyse lexicale*

```
AnalyseurLexical(↘ AFD, ↘ Act = {a}, ↘ TypeSymb = {s}, ↘ Chaine)
 / AFD = (A, Q, I, F, μ) /
 /initialisation de l'automate/
EtatCourant ← I
mot ← ∅
Ch ← PremierCaractere(Chaine)
Tant Que Ch ≠ ε et EtatCourant ≠ 0   Faire
     mot ← mot + Ch
     EtatCourant ← μ(Ch, EtatCourant)
     Ch ← CaractereSuivant(Chaine)
     Si EtatCourant ∈ F et μ(Ch, EtatCourant) = 0
        Alors  / avec q_i = EtatCourant /
                 s_i, lexeme ← A_i(mot), a_i ∈ Act, s_i ∈ TypeSymb
                 Si s_i ≠  délimiteur ou commentaire
                    Alors no ← EnvoyerTableSymboles(lexeme, s_i)
                              EnvoyerAnalyseurSyntaxique(lexeme, (s_i; no))
                 Fin Si
                 mot ← ∅
                 EtatCourant ← I
     Fin Si
Fin Tant Que
Si EtatCourant ∈ F et Ch = ε
   Alors Retourner Vrai
   Sinon Retourner Faux
Fin Si
```

FIG. 6.3 – *Algorithme d'analyse lexical à partir d'un AFD*

Gestion de la table des symboles

Les unités lexicales reconnues sont stockées dans la table des symboles. Cette table indique la classe lexicale (rôle de l'unité lexicale dans la phrase) et la catégorie du symbole (Par exemple, table 6.1).

Si le symbole est présent dans la table, celle-ci fournit son code lexical. Sinon, la classe de ce symbole est calculée. Son numéro est alors l'indice de la place libre suivante dans la table et

Catégorie du symbole	Classe lexicale
mot clé	classe de ce mot clé
nombre	C^{ste}
chaîne	C^{ste}
autre mot	identificateur

TAB. 6.1 – *Table des symboles*

l'information sur le symbole est rangée à cette place.

La gestion de cette table est un élément important de la phase d'analyse du programme. A l'accès associatif, on préfère souvent une gestion plus optimisée avec un accès direct. En pratique, on utilise souvent le «hashcoding».

Remarque : les mots clés du langage sont pré-enregistrés dans la table avec leur classe lexicale.

Traitement des erreurs

Les principales erreurs soulevées sont la rencontre d'un caractère non-autorisé ou le débordement de la table des symboles. Dans le premier cas, il suffit de «passer la main» à l'analyseur syntaxique avec «autre» comme classe de symboles. Dans le second cas, l'unique solution consiste à arrêter le programme (erreur fatale).

6.2.3 LEX

LEX est un outil pour construire des analyseurs lexicaux à partir de notations basées sur les expressions régulières. Il est généralement utilisé de la manière décrite par la figure 6.4. Le programme obtenu permet alors de décomposer une chaîne en lexèmes (Fig. 6.5).

Programme source LEX : expressions régulières et actions (lex.l)
↓
Compilateur LEX
↓
Table de transition, fonctions de reconnaissance des lexèmes, actions en C... (lex.yy.c)
↓
Compilateur C
↓
Analyseur Lexical (a.out)

FIG. 6.4 – *Principe de fonctionnement de LEX*

Un programme LEX se décompose en quatre parties (fig. 6.6) :

1. les déclarations (pour définir les constantes, les variables et autres structures de données) ;
2. les définitions régulières ;

flot d'entrée ⟶ a.out ⟶ Suite d'unités lexicales

Fig. 6.5 – *Principe de fonctionnement du programme engendré par LEX*

3. les règles de traduction (avec des actions permettant par exemple la gestion de la table des symboles ou permettant d'effectuer des calculs) ;
4. les procédures auxiliaires utilisées par les actions ou permettant de faire un programme indépendant (fonction «Main»).

```
%{
/* Déclaration des variables */
/* et des symboles appelés constantes littérales */
/* #include... */
%}
/* Déclarations des classes et des expressions régulières */
%%
/* Règles de traduction */
m1  {actions1}
m2  {actions2}
...
mn  {actionn}
/* mi est une expression régulière */
/* actioni est une action lorsque le lexème correspond à mi*/
%%
/* Procédures auxiliaires en C  utiles dans les actions
et/ou permettant de faire un programme indépendant */
```

Fig. 6.6 – *Constitution d'un programme LEX*

Exemple : Analyse lexicale d'un flottant saisi au clavier (programme en langage LEX de la figure 6.7). Un flottant est ici composé de la manière suivante :

- d'un signe + ou - facultatif ;
- d'une partie entière d'au moins un chiffre s'il n'y pas de partie décimale ;
- éventuellement d'une partie décimale : un point suivi d'au moins un chiffre ;
- le tout peut être suivi d'un exposant qui est constitué de la lettre «e» ou «E» suivie éventuellement d'un signe et d'un ou deux chiffres.

La collaboration avec l'analyseur syntaxique s'effectue selon l'algorithme déjà présenté.

```
%{
%}
Sig   [+\-]
Ent   [1-9][0-9]*
Dec   [0-9]*\.[0-9]+
Exp   [eE]{Sig}?([0-9]{1,2})
Delim [\n\t ]
%%
Delim                       {}
{Sig}?({Ent}|{Dec})({Exp})? {printf("flottant:%s\n", yytext);}
.                            ECHO ;

%%
int main(void)
{
yylex();
return 0;
}
```

FIG. 6.7 – *Exemple de programme LEX*

valeur	resultat
fg	fg
12	flottant : 12
+12	flottant : +12
-12.	flottant : -12 .
-12.32	flottant : -12.32
12.2e1	flottant : 12.2e1
12e123	flottant : 12e12 flottant : 3
e23	eflottant : 23
1e-2365	flottant : 1e-23 flottant : 65
12E-12.1e2	flottant : 12E-12 flottant : .1e2

TAB. 6.2 – *Jeux d'essai proposés pour le programme de la figure 6.7*

6.2.4 L'analyse syntaxique et limites des AEF

Il est possible de mettre en place une analyse syntaxique basée sur des grammaires régulières (donc utilisant un AFD). Les caractères du vocabulaire sont alors les symboles du langage. Sur les transitions, on associe alors des actions permettant de réagir à la présence d'un symbole (stockage de valeurs, affichage de résultats, calculs...).

Cependant, les AFDs (et les grammaires régulières) sont très souvent inadaptés pour représenter les langages de haut niveau non réguliers. Ces langages sont souvent des langages hors-contexte.

Le moyen le plus adapté pour de tels langages est l'automate à pile permettant d'implanter une analyse descendante (mais il existe aussi d'autres méthodes d'analyse utilisant d'autres algorithmes) et permettant de provoquer des actions après seulement une certaine série de symboles (problème du parenthésage par exemple ainsi que des opérateurs binaires et des expressions arithmétiques).

Ces automates sont déterministes seulement pour des grammaires hors-contexte de type LL(1) (voir [Hég95]). Nous n'aborderons pas cela dans le cadre de ce cours mais ce sera étudié en cours de compilation.

6.3 Le langage naturel [Per95]

Dans l'article fondateur de Shannon [1], on trouve, comme nous l'avons vu, un modèle très voisin des automates finis, qui est directement issu des chaînes de Markov [2]. Ce modèle est appliqué en particulier à ce que Shannon appelle des *approximations* du langage naturel. Il s'agit de chaînes de Markov d'ordre croissant réalisant un automate local pour la structure de la langue au niveau lexical (lettre par lettre ou mot par mot). Ainsi, en prenant successivement au hasard des digrammes chevauchant dans un texte en anglais, on obtient l'exemple saisissant :

ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE

dans lequel certains mots existent réellement ('*on, are, be, ...* ') et d'autres le mériteraient ('*deamy, ilonasive, ...*'). Le résultat obtenu avec des trigrammes au lieu de digrammes est encore plus étonnant :

IN NO IST LAT WHEY CRACTIC FROURE BIRS GROCID PONDENOME OF DEMONSTRURES OF THE REPTAGIN IS REGOACTIONA OF CRE

Shannon présente d'autres exemples utilisant des chaînes de Markov au niveau des mots au lieu des lettres. En fait, dès le départ, A. A. Markov a introduit son modèle dans cette perspective et son article, daté de 1913, s'intitule "Essai d'une recherche statistique sur le texte du roman Eugène Onegine" [3].

L'idée d'utiliser des automates pour décrire la langue naturelle est reprise un peu plus tard par Chomsky [Cho56]. En fait, Chomsky n'introduit les automates que pour les écarter assez rapidement au profit du niveau supérieur de ce que l'on nomme aujourd'hui la *hiérarchie de Chomsky* : automates finis au rez-de-chaussée, automates à pile et grammaires context-free au premier étage, grammaires context-sensitives au second, machines de Turing au dernier. Les arguments employés par Chomsky pour écarter les automates finis comme modèle adéquat des langues naturelles sont fondés sur la présence de *structures parenthésées* provenant de constructions grammaticales comme les propositions conditionnelles :

$$\text{si } S_1 \text{ alors } S_2$$

analogues à celles des langages de programmation et qui permettent, en principe, une imbrication non bornée. De nombreux modèles autres que les automates ont été proposés pour rendre compte des structures syntaxiques y compris les grammaires de Lambek ou les grammaires transformationelles de Z. Harris.

1. C. E. Shannon, "A Mathematical Theory of Communication", *Bell Systems Tech. Jour.*, XXVII:379–423, 623–656, 1948.

2. Une chaîne de Markov à temps discret est un système évoluant au cours du temps en passant par différents états. Les transitions se font à des instants $t_1, t_2, ...t_n$. Le passage d'un état e_1 vers un état e_2 se fait selon une probabilité p ne dépendant que des états e_1 et e_2 et indépendante de l'instant n où le système se trouve dans l'état e_1.

3. *Bull. Acad. Imper. Sci. St Petersbourg*, 7, 1913.

Le fait que l'anglais ne soit pas un langage rationnel semble aujourd'hui être, à la fois, non prouvable et en balance avec le fait que, de toutes façons, une description complète de la syntaxe de l'anglais (ou du français) est aussi difficile à réaliser sous forme d'automate fini que de grammaire. Les automates finis ayant par contre l'avantage sur les grammaires de se prêter à un grand nombre de manipulations automatisables. Certains, comme Maurice Gross, pensent que ce sont finalement les automates finis qui fournissent l'essentiel de la description[4]. Du point de vue des automates, la différence essentielle avec les grammaires est le fait que la théorie logique associée aux automates (la logique monadique de Büchi) est décidable alors que la plupart des problèmes associés aux grammaires ne le sont pas. Si par exemple on peut assez facilement déterminer si deux automates finis sont équivalents, il n'en est pas de même des grammaires : on peut démontrer que le problème de savoir si deux grammaires sont équivalentes est indécidable.

6.4 Autres applications

Nous venons de voir que langage et automates sont utilisés pour générer des programmes et pour comprendre le langage naturel. Cependant, ce ne sont certainement pas les seules utilisations possibles.

Les automates sont aussi souvent utilisés pour modéliser des phénomènes naturels liés au comportement. En particuliers, ils permettent de modéliser des connaissances complexes ainsi que des règles de comportement dépendant d'événement «extérieurs». Pour cela, il est possible d'utiliser des *réseaux de Pétri*. En particulier, les *modèles de comportement* en animation se basent sur des *graphes d'événements*.

Dans certains cas, le comportement est aussi lié à des paramètres stockastiques. Aussi, lorsque l'automate n'est pas déterministe, il est parfois associé aux transitions sur un même symbole (ou événement) un paramètre probabiliste : ce sont les *automates stockastiques* ou *chaînes de Markov*. Ces automates permettent, en particulier, de modéliser la gestion de files d'attentes...

Les automates sont aussi utilisés en modélisation et conception de systèmes. Une modélisation OMT consiste non seulement en une description des objets mis en jeu mais aussi leurs relations vis-à-vis des divers évènements.

Toujours lié au comportement de systèmes, on trouve des applications des automates pour la gestion du dialogue Homme-Machine et pour la conception de jeux.

4. voir M. Gross et D. Perrin, "Electronic Dictionnaries and Automata in Computational Linguistics", Lecture Notes in Computer Science, 377. Springer, 1989.

Bibliographie

[AEL98] C. Attiogbe, C. Enguehard, and P. Lamarre. *Module Informatique III, Deug MIAS.* Université de Nantes - Faculté des sciences et des techniques, 1998.

[All97] F. Alliaume. *Documents pour la compilation.* Université de Nantes, Faculté des sciences, Juin 1997.

[ASU91] A. Aho, R. Sethi, and J. Ullman. *Compilateurs : Principes, techniques et outils.* Inter-Editions, 1991.

[Aut94] J.-M. Autebert. *Théorie des langages et des automates.* Masson, 1994.

[Cho56] N. Chomsky. Three models for the description of language. In sept 56 Proc. of the symposium on Information Theory, editor, *IRE Trans. Inf. Th.* IT-2, 1956.

[Gin68] A. Ginzburg. *Algebraic Theory of Automata.* ACM, Academic Press, 1968.

[Hég95] G. Hégron. *Compilation.* Eccole des Mines de Nantes, FI1A, Mars 1995.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In C.E.Shannon and J.McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[Lan98] L. Lander. *CS 573 : Automata Theory and Formal Languages.* Thomas J. Watson School of Engineering and Applied Science, Binghamton University, University of New York, 1998. (http://bingweb.binghamton.edu/ lander/cs573.html).

[LPS83] M. Lucas, J.-P. Peyrin, and P.-C. Scholl. *Algorithmique et représentation des données : 1 - Files, automates d'états finis.* Masson, 1983.

[Mar93] D. Martin. *Probabilités pour l'informatique. Support de cours.* Maîtrise d'informatique de l'Université de Nantes, 1993.

[Nie] T. Niemann. *A Guide to LEX & YACC.* (http://members.xoom.com/thomasn/y_man.html).

[Per95] D. Perrin. Les débuts de la théorie des automates. *Technique et Science Informatique (TSI)*, 14:409–443, 1995. (http://www-igm.univ-mlv.fr/ perrin/Recherche/Publications/Loi/copie3.ps).

[Sha97] H. Shapiro. *CS 500 : Introduction to the Theory of Computation.* The University of New Mexico, 1997. (ftp://ftp.cs.unm.edu/pub/shapiro/CS500/chapter3.ps).

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[Tur36] A. Turing. On computable numbers with an application to the entscheidungs problem. *Proceedings of the London Math. Soc.*, 42:230–265, 1936.

[Wat93a] B. W. Watson. A taxonomy of finite automata construction. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993. (ftp://ftp.win.tue.nl/pub/techreports/pi/automata/taxonomy/2nd.edition/constax.ps.Z).

[Wat93b] B. W. Watson. A taxonomy of finite automata minimization algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993. (ftp://ftp.win.tue.nl/pub/techreports/pi/automata/taxonomy/2nd.edition/mintax.ps.Z).

[WM94] R. Wilhelm and D. Maurer. *Les compilateurs : théorie, construction, génération.* Masson, 1994.