Licence d'informatique

Université de Marne-la-Vallée

AUTOMATES ET GRAMMAIRES

Édition 2004–2005

Jean Berstel

Table des matières

Chapitre 1

Mots, langages, équations, codes

Ce chapitre introductif contient les définitions des notions liées aux mots et aux langages. Nous avons ajouté une digression sur les équations, et la définition des codes. Une dernière section contient un exposé de l'algorithme de Sardinas et Patterson.

1.1 Mots

Un *alphabet* est un ensemble dont les éléments sont des *lettres*. Les alphabets sont toujours supposés finis. Un *mot* est une suite finie de lettres que l'on note par simple juxtaposition :

$$w = a_1 a_2 \cdots a_n, \quad a_i \in A\,.$$

Le *mot vide* est le seul mot composé d'aucune lettre. Il est noté ε ou 1. La *longueur* d'un mot w est le nombre de lettres qui le composent, et est notée $|w|$. Le mot vide est le seul mot de longueur 0.

Le produit de concaténation de deux mots $x = a_1 a_2 \cdots a_n$ et $y = b_1 b_2 \cdots b_m$ est le mot xy obtenu par juxtaposition :

$$xy = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m\,.$$

Bien entendu, on a $|xy| = |x| + |y|$. On note A^* l'ensemble des mots sur A.

Exemple 1.1. Les gènes sont des mots sur l'alphabet `ACGT`, les protéines sont des mots sur un alphabet à 20 lettres. Les entiers naturels, écrits en base 10, sont des mots sur l'alphabet des dix chiffres décimaux. En écriture hexadécimale, s'y ajoutent les “chiffres” $A - F$. Les codes d'entrée dans les immeubles appelés “digicodes” sont des mots écrits sur un alphabet à 12 symboles. ◁

Soit A un alphabet. Soit B une partie de A. Pour tout mot $w \in A^*$, la *longueur* en B de w est le nombre d'occurrences de lettres de B dans le mot w. Ce nombre est noté $|w|_B$.

En particulier, $|w| = |w|_A$. Pour toute lettre $a \in A$, $|w|_a$ est le nombre d'occurrences de a dans w. On a
$$|w|_B = \sum_{b \in B} |w|_b \ .$$
Soit $w = a_1 \cdots a_n$, avec $a_1, \ldots, a_n \in A$. Le mot *miroir* de w est le mot noté $\tilde{w}$ ou $w^\sim$ défini par
$$w^\sim = a_n \cdots a_1$$
Evidemment, $(uv)^\sim = \tilde{v}\tilde{u}$ et $(w^\sim)^\sim = w$.

Un mot u est *préfixe* ou *facteur gauche* d'un mot v s'il existe un mot x tel que $ux = v$. Le mot u est *préfixe strict* ou *propre* si, de plus, $u \neq v$. De manière symétrique, u est *suffixe* ou *facteur droit* de v si $xu = v$ pour un mot x. Si $u \neq v$, alors u est suffixe propre ou strict. Le nombre de préfixes d'un mot v non vide est $1 + |v|$.

Un mot u est *facteur* d'un mot v s'il existe x, y tels que $v = xuy$. Le mot $aabab$ sur $A = \{a, b\}$ a 12 facteurs.

Lemme 1.2. (Lemme de Levy) *Soient x, y, z, t des mots tels que*
$$xy = zt \ .$$
Alors il existe un mot w tel que
– ou bien $xw = z$ et $y = wt$;
– ou bien $x = zw$ et $wy = t$.

Il en résulte en particulier que si $|x| = |z|$, le mot w est vide, et donc $x = z$ et $y = t$. En d'autres termes, un monoïde libre est *simplifiable* à gauche et à droite.

Preuve. Posons $x = a_1 \cdots a_n$, $y = a_{n+1} \cdots a_m$ avec $a_i \in A$ et de même $z = b_1 \cdots b_p$, $b_{p+1} \cdots b_q$ avec $b_i \in A$. Comme $xy = zt$, on a $m = q$ et $a_i = b_i$ pour $i = 1, \ldots, m$, de sorte que $z = a_1 \cdots a_p$ et $t = a_{p+1} \cdots a_m$. Si $|z| = p \leq n = |x|$, posons $w = x_{p+1} \cdots x_n$. Alors
$$x = zw \ \text{ et } \ wy = t \,.$$
Si $|z| > |x|$, posons $w = u_{n+1} \cdots x_p$. Alors
$$xw = z \ \text{ et } \ y = wt \,.$$
∎

1.2 Langages

Les sous-ensembles de A^* sont appelés des *langages* formels. Par exemple, pour $A = \{a, b\}$, l'ensemble $\{a^n b^n \mid n \geq 0\}$ est un langage.

On définit sur les langages plusieurs opérations. Les *opérations booléennes* sont l'union, l'intersection, la complémentation et la différence qui s'en déduit. Si X et Y sont deux parties de A^*, alors
$$\begin{aligned} X \cup Y &= \{z \in A^* \mid z \in X \text{ ou } z \in Y\} \\ X \cap Y &= \{z \in A^* \mid z \in X \text{ et } z \in Y\} \\ X^c &= A^* \setminus X = \{z \in A^* \mid z \notin X\} \\ X \setminus Y &= X \cap Y^c = \{z \in A^* \mid z \in X \text{ et } z \notin Y\} \end{aligned}$$

Le *produit* (de concaténation) de deux langages X et Y est le langage

$$XY = \{xy \mid x \in X, \text{ et } y \in Y\}$$

On a en particulier $X\{\varepsilon\} = \{\varepsilon\}X = X$. On vérifie que

$$X(Y \cup Z) = XY \cup XZ, \quad X(Y \cap Z) \subset XY \cap XZ$$

La deuxième inclusion est en général stricte.

Les puissances de X sont définies par $X^0 = \{\varepsilon\}$, $X^1 = X$, et $X^{n+1} = X^n X$ pour $n \geq 1$. En particulier, si A est un alphabet, A^n est l'ensemble des mots de longueur n.

L'*étoile* de X est l'ensemble

$$X^* = \bigcup_{n \geq 0} X^n = \{x_1 \cdots x_n \mid n \geq 0, x_1, \ldots x_n \in X\}$$

L'opération *plus* est définie de manière similaire.

$$X^+ = \bigcup_{n > 0} X^n = \{x_1 \cdots x_n \mid n > 0, x_1, \ldots x_n \in X\}$$

Exemple 2.1. Soit $X = \{a, ba\}$. Les mots de X^*, classés par longeuur, sont

0	ε
1	a
2	aa, ba
3	aaa, aba, baa
4	$aaaa, aaba, abaa, baaa, baba$
5	$aaaaa, aaaba, aabaa, abaaa, ababa, baaaa, baaba, babaa$

1.3 Equations

Proposition 3.1. *Soient u et v deux mots non vides. Les conditions suivantes sont équivalentes :*

(1) $uv = vu$;
(2) *il existe deux entiers $n, m \geq 1$ tels que $u^n = v^m$;*
(3) *il existe un mot w non vide et deux entiers $k, \ell \geq 1$ tels que $u = w^k$, $v = w^\ell$.*

Preuve. (1) $\Rightarrow$ (3). Si $|u| = |v|$, alors $u = v$ et l'implication est évidente. En raisonnant par récurrence sur $|uv|$, supposons $|u| > |v|$. Soit alors w tel que $u = vw$. En reportant dans l'équation $uv = vu$, on obtient $vwv = vvw$, d'où en simplifiant $wv = vw$. Par récurrence, il existe un mot x et des entiers $k, \ell \geq 1$ tels que $v = w^k$, $w = x^\ell$, d'où $u = x^{k+\ell}$.

(3) $\Rightarrow$ (2). Si $u = w^k$ et $v = w^\ell$, alors $u^\ell = v^k$.

(2) $\Rightarrow$ (1). La conclusion est évidente si $u = v$. Supposons donc $|u| > |v|$, et soit w tel que $u = vw$. Alors

$$u^n v = (vw)^n v = v(wv)^n = v^{m+1}$$

et en simplifiant la dernière égalité, $(wv)^n = v^m$. Comme $v^m = u^n = (vw)^n$, on a $(wv)^n = (vw)^n$, donc $wv = vw$, ou encore $uv = vu$.

(2) ⇒ (1) (variante). On a également $u^{2n} = v^{2m}$. Or

$$u^{2n} = u^{n+1}u^{n-1} = uv^m u^{n-1} \ = \ v^{2m} = v^{m+1}v^{m-1} = vu^n v^{m-1}$$

et ce mot commence par uv et par vu, donc $uv = vu$. ■

1.3.1 Compléments sur les équations

Soient V et A deux alphabets disjoints. Une *équation* sur A est un couple $e = (\alpha, \beta)$ de mots de $(V \cup A)^*$. Une équation $e = (\alpha, \beta)$ est *non triviale* si $\alpha \neq \beta$.

Une *solution* de l'équation e est un morphisme $h : (V \cup A)^* \to A^*$ *invariant* sur A tel que

$$h(\alpha) = h(\beta)\ .$$

Une solution h est *cyclique* s'il existe un mot w tel que $h(x) \in w^*$ pour toute variable x. Une équation est *sans constante* si $\alpha, \beta \in V^*$.

La proposition précédente affirme que les solutions de l'équation sans constante $xy = yx$ en deux variables x, y sont cycliques. La proposition suivante en donne une extension.

Proposition 3.2. *Soit $e = (\alpha, \beta)$ une équation non triviale, sans constante, en deux variables x et y. Alors toute solution de e est cyclique.*

En d'autres termes, si deux mots u et v vérifient n'importe quelle relation non triviale, ils sont puissances d'un même mot.

Preuve. Clairement, si α ou β est le mot vide, la seule solution est celle qui envoit x ou y (ou les deux) sur le mot vide. On peut donc supposer les deux mots non vides, et on peut supposer qu'ils commencent et finissent pas des lettres différentes. On est donc ramené aux équations

$$x\gamma x = y\gamma' y \quad \text{ou} \quad x\gamma y = y\gamma' x$$

La preuve est par récurrence sur la longueur $|h(x| + |h(y)|$ d'une solution. Considérons une solution $h(x) = u$, $h(y) = v$ de l'une des équations. Si $|u| = |v|$, alors $u = v$. Sinon, supposons $|u| > |v|$ et posons $u = vw$. On obtient alors les égalités

$$wh(\gamma)vw = vh(\gamma')v \quad \text{ou} \quad wh(\gamma)v = vh(\gamma')vw$$

Ceci montre que (v, w) est solution d'une équation $\alpha' = \beta'$, où toute occurrence de x est remplacée par xy. Comme $|v| + |w| < |h(x| + |h(y)|$, cette solution de la dernière équation est cyclique, et donc h également. ■

Voici d'autres exemples d'équations :

$$\begin{aligned} x^2y^2 &= z^2 \\ x^3y^3 &= z^3 \\ xy &= yz \end{aligned}$$

Voici maintenant deux systèmes d'équations :

$$\begin{array}{ll} (x_1x_2x_3)^i = x_1^i x_2^i x_3^i & i = 2,3 \\ (x_1x_2x_3x_4)^i = x_1^i x_2^i x_3^i x_4^i & i = 2,3,4 \end{array}$$

Ces deux systèmes n'ont que des solutions cycliques. Considérons en effet le premier système, et notons $1, 2, 3$ les composantes d'une solution. On a pour $i = 2$ après simplification

$$2312 = 1223 \tag{1}$$

et, pour $i = 3$

$$223312 = 122233 \tag{2}$$

$$231122 = 112223 \tag{3}$$

Si 1 est préfixe de 23, alors $1x = 23$ pour un mot x, et (2), qui débute par 223, débute par $21x$. Comme (2) débute aussi par 12, on a

$$21 = 12$$

Ainsi $1, 2, 12$ et 23 (par l'équation (1) sont puissances d'un même mot, donc $1, 2$ et 3 également. On raisonne de même si 3 est suffixe de 12 : on a $12 = y3$, et alors (3) se termine par $122 = y32$ et par 23, donc $23 = 32$.

Reste le cas où 23 est préfixe propre de 1 et 12 est suffixe propre de 3. Ceci est impossible car alors

$$|3| < |1| < |3|$$

La preuve pour le deuxième est similaire. On connaît des résultats plus généraux dans cette direction, à savoir :

L'équation

$$x_1^n x_2^n \cdots x_k^n = y^n$$

pour $n \geq k$ n'a que des solutions cycliques. (Appel, Djorup, Trans. Amer. Math. Soc. **134** (1968), 461–470).

1.4 Codes

1.4.1 Définition des codes

On appelle *code* toute partie C d'un monoïde libre A^* qui vérifie la condition suivante : pour tout $x_1, \ldots, x_n, y_1, \ldots y_m \in C$,

$$x_1 \cdots x_n = y_1 \cdots y_m \Longrightarrow n = m, x_i = y_i, i = 1, \ldots, n$$

En d'autres termes, C est un code si tout mot de C^* se factorise, de manière unique, en un produit de mots de C. Lorsqu'un ensemble n'est pas un code, on s'en aperçoit en général assez facilement, en exhibant une double factorisation. Il est plus difficile d'établir qu'un ensemble est effectivement un code.

Exemple 4.1. L'ensemble $\{a, ab, ba\}$ n'est pas un code puisque le mot aba s'écrit à la fois comme produit $a \cdot ba$ et comme produit $ab \cdot a$.

L'ensemble $C = \{b, ab, baa, abaa, aaaa\}$ est un code. En effet, un mot de C^* qui aurait deux factorisations commencerait par baa ou par $abaa$. Regardons le premier cas (le deuxième est en fait similaire). L'une des factorisations commencerait par b et l'autre par baa. Pour compléter ces factorisations, il faut compenser l'excès de la deuxième factorisation, soit le mot aa. Pour cela, on doit ajouter à la première factorisation le seul mot de C commençant par aa, à savoir a^4. La première factorisation commence donc par $(b, aaaa)$. Mais alors, la deuxième factorisation ne peut être complétée que par a^4, et devient $(baa, aaaa)$. On est alors revenu au point de départ, et il n'y a donc pas de double factorisation possible. ◁

Les codes les plus simples sont les *codes uniformes*. Ce sont des ensembles dont tous les mots ont même longueur. Par exemple, l'ensemble A^n des mots de longueur n est un code, si $n \geq 1$. Le code ASCII qui associe à certains caractères des mots binaires de longueur 7 est un code.

Une autre famille importante de codes est formée des *codes préfixes*. Une partie X de A^* est dit *préfixe* si deux éléments distincts de X sont incomparables pour l'ordre préfixiel, donc si $x, xu \in X$ implique $u = 1$. Une partie préfixe contenant le mot vide est réduite au mot vide. Tout autre partie préfixe est un code, appelé code préfixe. En effet, soit $X \subset A^+$ une partie préfixe. Si elle n'était pas un code, il existerait des mots $x, y \in X$, $z, t \in X^*$ tels que $xz = yt$, avec $x \neq y$. Mais alors, le lemme de Levy implique que x et y sont comparables dans l'ordre préfixiel, contradiction.

Par passage à l'image miroir, on obtient la classe des *codes suffixes*. Un code qui est à la fois préfixe et suffixe est *bipréfixe* ou *bifixe*. Par exemple, l'ensemble $a \cup ba^*b$ est un code bifixe.

1.4.2 Messages courts

Le service des messages courts (SMS, pour "short message service") est disponible sur le téléphone. Un message est tapé avec les touches des chiffres du clavier. A chacune des touches correspondent en général trois lettres. Ainsi, à la touche 2 correspondent les lettres a, b et c, et à la touche 8 correspondent le t, u et v. Pour obtenir la lettre a, on tape 2, et pour obtenir b ou c, on tape deux ou trois fois le 2. Ainsi, pour écrire *bonjour*, on tape 22 pour b, 666 pour o puis une petite pause, puis 66 pour n, pour 5 (j) 666 (o) 88 (u) et 777 (r), soit en tout

22666 66566688777

Ceci est bien laborieux, et c'est pourquoi il existe une version "intuitive" qui propose, au fur et à mesure de la frappe, le mot le plus "plausible" possible (ou d'autres si on appuie sur la touche adéquate).

La version normale est bien un code, au sens de la théorie des codes. Le code n'est pas préfixe, puisque 66 et 666 par exemple sont des mots du code. Il y a un mot spécial dans le code, l'espace ou blanc, qui sert de séparateur. Le code SMS, s'il n'est pas préfixe, est à *délai de déchiffrage* 1. Par définition, ceci signifie qu'il suffit de connaître la lettre suivante pour savoir si ce que l'on vient de lire est un mot du code ou non. Par exemple, si on a vu un 6, cela peut être un m ou le début d'un n ou d'un o. Le symbole suivant permet de décider si c'est le code d'un m : c'est le cas si le symbole suivant est tout sauf un 6.

1.4.3 Unicode

Les caractères d'un texte sont représentés, dans les supports électroniques, sous forme de séquences de bits, en général groupés en octets.

Historiquement, les premiers codes de caractères sont US-ASCII et ISO-8859-1. Le premier code 128 symboles sur 7 bits, le deuxième code sur 8 bits 256 caractères. Ce que l'on appelle Unicode ou UCS (Universal Character Set) est une convention de codage de caractères sur 16 bits.

En fait, l'ensemble de caractères Unicode est défini, maintenu et développé par deux entités qui – heureusement – coopèrent et qui dénomment de deux façons différentes (presque) la même chose :
- "Unicode standard" est géré par le Unicode Consortium (Californie), une organisation sans but lucratif financée par les constructeurs informatique américains
- "ISO/IEC 10646 Universal Multiple-Octet Coded Character Set (UCS)" est géré par le "Joint technical committee" numéro 1 de l'ISO (International Organization for Standardization) et IEC (International Electronical Commission).

L'unicode d'un caractère est donc de 16 bits. Les codes des caractères sont choisis avec soin. Ainsi, pour les 128 caractères qui figurent dans le code ASCII, le premier octet de l'unicode est nul et le deuxième est le code ASCII usuel. De même pour ISO-8859-1.

En fait, chaque caractère est défini par un nom (par exemple *Latin capital letter A*, et un numéro (par exemple `U+0041`) qui est son indice dans la table. Il faut distinguer le caractère (aussi appelé *script*) de son image telle qu'elle apparaît par exemple sur une feuille de papier (aussi appelé *glyph*). La forme (italique, sans jambage, taille, etc) ne figure pas dans l'unicode. En revanche, il y a une différence entre le a latin et le α grec, entre majuscules et minuscules, etc.

Unicode ne concerne que le texte écrit, à l'exception notable de la notation musicale.

Les 65536 entrées ne sont pas toutes remplies (la version 3.0 de Unicode contient 49194 caractères), mais dès à présent, ne suffisent pas pour couvrir l'ensemble des besoins prévus. Un mécanisme d'extension est déjà en vue. En fait, le comité ISO avait envisagé, au départ, le codage de chaque caractère sur 31 bits. Le codage sur 16 bits est considéré comme le premier BMP (Basic Multilingual Plane).

Dans la pratique, des sous-ensembles ne contenant que des caractères utilisés fréquemment ont été définis. Ainsi
- Windows Glyph List (version 4 : WGL4) contient 650 caractères.
- Des sous-ensembles "européens" MES-1, MES-2, MES-3 ont été définis. Dans MES-1 il y a 335 caractères latins, correspondant aux caractères contenus dans ISO 8859, parties 1, 2, 3, 4, 9, 10, 15. MES-2 contient 1024 caractères, MES-3 en contient 2819.
- JIS est un ensemble de caractères pour le japonais.

1.4.4 Unicode et UTF

Représenter un texte en unicode est une perte de place, dans la plupart des cas. En effet, en général on écrit – en français ou en anglais – avec un jeu de caractères qui tient sur 8 bits, voire sur 7 bits.

Le codage UTF (Unicode Transformation Format, ou Uniform Transfert Format) remédie à cet inconvénient. Le codage présenté ici est UTF-8, d'autres variantes existent.

Le codage est à *longueur variable*. A chaque caractère unicode (sur 2 octets) est associé un codage UTF sur 1, 2, ou 3 octets, selon la répartition suivante
- les caractères de 0 à 127 (de `U+0000` à `U+007F`) sont codés sur un octet ;
- les caractères de 128 à 2047 (de `U+0080` à `U+07FF`) sont codés sur deux octets ;
- les caractères de 2048 à 65535 (de `U+0800` à `U+FFFF`) sont codés sur trois octets.

Le codage est comme suit.
- Les caractères de 0 à 127 sont codés comme les octets de `0x00` à `0x7F`. Ceci signifie que les textes qui contiennent de l'ASCII sur 7 bits ont le même codage en ASCII et en UTF-8. Aussi, le bit le plus fort du code d'un tel caractère est nul. Il a donc la forme `0xxxxxxx`.
- Pour les caractères de 128 à 2047, le premier octet commence par `110` et le deuxième par `10`. Il reste 5 bits dans le premier octet et 6 bits dans le deuxième. Ces 11 bits suffisent pour représenter les nombres jusqu'à 2047. Un code a donc la forme `110xxxxx 10xxxxxx`.
- Pour les caractères de 2048 à 65535, le premier octet commence par `1110` et les deux suivants par `10`. Un code a donc la forme `1110xxxx 10xxxxxx 10xxxxxx`.Il reste $4 + 6 + 6 = 16$ bits, suffisants pour représenter les caractères.

EXEMPLE 4.2. Le caractère `U+00A9 = 1010 1001` qui est le signe de copyright est codé en *110*00010 *10*10 1001.

Le caractère `U+2260 = 0010 0010 0110 0000` (signe "différent") est codé par les trois octets *1110*0010 *10*001001 *10*100000 = `0xE2 0x89 0xA0`. ◁

Notons que le décodage se fait en examinant les premiers bits de chaque octet. Si le premier bit est nul, c'est le code d'un caractère ascii. Pour les octets qui commencent par 1, il faut regarder le bit suivant. Si c'est 0, il s'agit d'un octet qui complète un caractère. Si c'est 10, c'est le début du code d'un caractère entre 128 et 2047, et si c'est 110 c'est un autre caractère. Noter qu'il y a des octets qui ne correspondent à aucun codage.

En fait, le code UTF-8 est conçu même pour les caractères du UCS complet, c'est-à-dire jusqu'à 31 bits. Les octets de tête de tels caractères commencent par 1^n0 pour $n = 3, 4, 5, 6$, les octets de suite sont les mêmes.

Il existe des variantes du codage UTF-8, nommés UTF-1, UTF-7,5, UTF-7, UTF-16.

1.4.5 Complément : algorithme de Sardinas et Patterson

Il est en général facile de tester si un code est préfixe ou suffixe. Lorsqu'un ensemble n'est pas un code, on s'en aperçoit assez facilement, en exhibant une double factorisation. Il est plus difficile d'établir qu'un ensemble est effectivement un code. Par exemple, des deux ensembles

$$X = \{b, ab, baa, abaa, aaaa\}, \qquad Y = \{b, ab, aab, abba\}$$

le premier est un code, le deuxième ne l'est pas : le mot $u = abbab$ a les deux factorisations $ab \cdot b \cdot ab = abba \cdot b$.

Nous allons donner un algorithme, connu sous le nom d'*algorithme de Sardinas et Patterson* pour tester si un ensemble est un code. Soit donc X un ensemble de mots, non vides, sur un alphabet A. On construit un graphe $G(X) = (P, U)$, où P est l'ensemble des préfixes non vides de mots de X, et U l'ensemble des couples (u, v) tels que
– $uv \in X$ ou
– $v \notin X$ et il existe $x \in X$ tel que $ux = v$.
Un arc *croisé* est un arc de la première espèce, un arc *avant* un arc du deuxième type.

Exemple 4.3. Pour $X = \{a, bb, abbba, babab\}$, l'ensemble P contient, en plus de X, les mots $\{b, ab, abb, abbb, ba, bab, baba\}$. Les arcs avant sont (a, abb), $(ab, abbb)$, (b, ba) et $(bab, baba)$. Les arcs croisés sont (b, b), (abb, ba), $(abbb, a)$, (ba, bab), (bab, ab) et $(baba, b)$. Le graphe $G(X)$ est donné dans la figure 4.1.

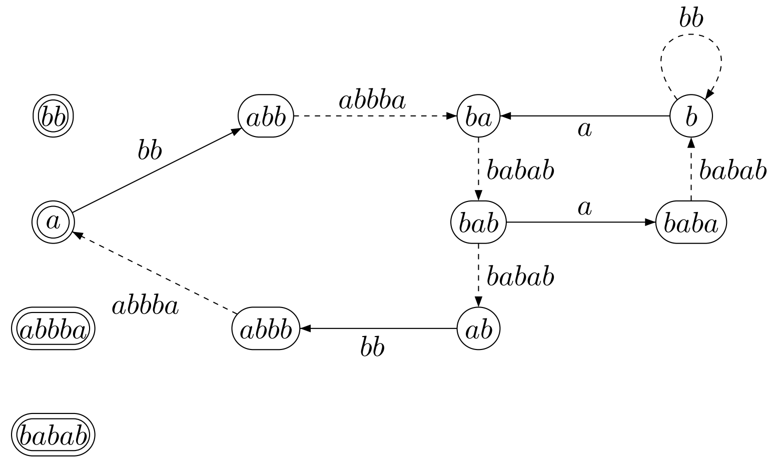

Fig. 4.1 – Le graphe de Sardinas et Patterson pour $\{a, bb, abba, babab\}$.

Les sommets correspondant aux mots de X sont doublement cerclés. Les arcs croisés sont tracés en pointillé, et les arcs avant sont tracés en trait plein. L'étiquette de chaque arc est un mot de l'ensemble X. Si l'arc (u, v) est croisé, alors l'étiquette est uv, sinon c'est le mot x tel que $ux = v$. Dans notre exemple, il y a un chemin qui part de a et arrive en a. En vertu du théorème suivant, l'ensemble X n'est pas un code (il aurait suffi d'un chemin entre deux sommets quelconques de X). ◁

Nous allons montrer la propriété suivante :

Théorème 4.4. *L'ensemble X est un code si et seulement s'il n'y a pas de chemin non trivial, dans $G(X)$, d'un sommet de X à un sommet de X.*

Nous commençons par un lemme.

LEMME 4.5. *Il y a un chemin d'un sommet $x \in X$ à un sommet $u \in P - X$ ssi il existe des mots $y, z \in X^*$ tels que $yu = z$.*

Preuve. Nous prouvons le lemme par récurrence sur la longueur du chemin. Soit (x, u) un arc. S'il est croisé, alors $xu \in X$; s'il est un arc avant, on a $xx' = u$ pour un $x' \in X$. Dans les deux cas, la conclusion est vérifiée.

Supposons l'existence d'un chemin de x à u, et que $yu = z$. Si (u, v) est un arc croisé, alors $y(uv) = zv \in X^*$, et si (u, v) est un arc avant, on a $ut = v$ pour un $t \in X$, donc $yv = zt \in X$. Dans les deux cas, l'existence de la factorisation est montrée.

Réciproquement, supposons que $yu = z$, pour $y, z \in X^*$ et $u \in P - X$. Il existe $t \in X^*$ et $w \in X$ tels que $z = tw$, soit encore

$$yu = tw$$

Si u est suffixe propre de w, on a $vu = w$ pour un mot dans P. Donc (v, u) est un arc croisé, et $y = tv$. Si au contraire w est un suffixe propre de u, on a $u = vw$, et (v, u) est un arc avant. Dans ce cas $yv = t$. Si $v \notin X$, on procède par récurrence. Si $v \in X$, c'est l'arc (v, u) qui constitue le chemin cherché. ∎

Preuve du théorème. Considérons un chemin non trivial de $x \in X$ à $y \in X$. Soit (u, y) son dernier arc qui est nécessairement croisé. On a donc $t = uy \in X$ et $t \neq y$. Si $u \in X$, alors $t \in X \cap X^2$ et X n'est pas un code. Si $u \in P - X$, il existe $x', y' \in X^*$ tels que $x'u = y'$. Mais alors $x't = y'y$, et comme $t, y \in X$ et $t \neq y$, X n'est pas un code.

Réciproquement, si X n'est pas un code, il existe $y', z' \in X^*$ et $y, z \in X$ tels que

$$y'y = z'z$$

et $y \neq z$. On peut supposer z plus long que y ; soit u tel que

$$uy = z$$

Alors (u, y) est un arc croisé. Si $u \in X$, l'arc (u, y) est un chemin de la forme cherchée ; si $u \in P - X$, l'égalité $z'u = y'$ prouve l'existence d'un chemin d'un $x \in X$ à u, d'où un chemin de x à y. ∎

Chapitre 2

Automates

Dans ce chapitre, nous présentons les bases de la théorie des automates finis. Nous montrons l'équivalence entre les automates finis et les automates finis déterministes, puis nous étudions des propriétés de fermeture. Nous prouvons le théorème de Kleene qui montre que les langages reconnaissables et les langages rationnels sont une seule et même famille de langages. Nous prouvons l'existence et l'unicité d'un automate déterministe minimal reconnaissant un langage donné, et nous présentons un algorithme de minimisation.

2.1 Introduction

Les automates finis constituent l'un des modèles de calcul les plus anciens en informatique. A l'origine, ils ont été conçus et employés comme une tentative de modélisation des neurones ; les premiers résultats théoriques, comme le théorème de Kleene, datent de cette époque. Parallèlement, ils ont été utilisés en tant qu'outils de développement des circuits logiques. Les applications les plus courantes sont à présent la modélisation de certains mécanismes de contrôle et le traitement de texte, dans son sens le plus général. L'analyse lexicale, la première phase d'un compilateur, est réalisée par des algorithmes qui reproduisent le fonctionnement d'un automate fini. La spécification d'un analyseur lexical se fait d'ailleurs en général en donnant les expressions rationnelles des mots à reconnaître. Dans le traitement de langues naturelles, on retrouve les automates finis sous le terme de réseau de transitions. Enfin, le traitement de texte proprement dit fait largement appel aux automates, que ce soit pour la reconnaissance de motifs ou pour la description de chaînes de caractères, sous le vocable d'expressions régulières. De nombreuses primitives courantes dans les systèmes d'exploitation modernes font appel, implicitement ou explicitement, à ces concepts.

Le concept d"'état" renvoit, en pratique, à une configuration donnée par l'ensemble des valeurs, à un moment donné, d'un groupe de paramètres. C'est l'abstraction de ce concept vers un élément dans un ensemble fini qui est à la base de la formalisation des automates en tant que machines ayant un nombre fini d'états. Tous les modèles qui gèrent des ensembles d'états et les contrôlent relèvent donc, à un degré plus ou moins important, des automates finis. On peut citer la modélisation UML, les circuits, les tables logiques, et aussi la vérification des programmes, où les automates sont en général gigantesques et donc sont donnés implicitement

La théorie des automates a également connu de grands développements du point de vue mathématique, en liaison étroite avec la théorie des monoïdes finis, principalement sous l'impulsion de M. P. Schützenberger. Elle a aussi des liens profonds avec les théories logiques.

2.2 Automates finis

2.2.1 Définition

Un *automate fini* sur un alphabet fini A est composé d'un ensemble fini Q d'*états*, d'un ensemble $I \subset Q$ d'états *initiaux* , d'un ensemble $T \subset Q$ d'états *terminaux* ou *finals* et d'un ensemble $\mathcal{F} \subset Q \times A \times Q$ de *flèches.* Un automate est habituellement noté

$$\mathcal{A} = (Q, I, T, \mathcal{F})$$

Parfois, on écrit plus simplement $\mathcal{A} = (Q, I, T)$, lorsque l'ensemble des flèches est sous-entendu, mais cette notation est critiquable car l'ensemble des flèches est essentiel. L'*étiquette* d'une flèche $f = (p, a, q)$ est la lettre a. Un *calcul* de longueur n dans $\mathcal{A}$ est une suite $c = f_1 \cdots f_n$ de flèches consécutives $f_i = (p_i, a_i, q_i)$, c'est-à-dire telles que $q_i = p_{i+1}$ pour $i = 1, \ldots, n-1$. L'*étiquette* du calcul c est $|c| = a_1 \cdots a_n$. On écrit également, si $w = |c|$,

$$c : p_1 \to q_n \quad \text{ou} \quad c : p_1 \xrightarrow{w} q_n$$

Par convention, il existe un calcul vide $1_q : q \to q$ d'étiquette ε ou 1 (le mot vide) pour chaque état q. Les calculs peuvent être composés. Etant donnés deux calculs $c : p \to q$ et $d : q \to r$, le calcul $cd : p \to r$ est défini par concaténation. On a bien entendu $|cd| = |c|\,|d|$.

Un calcul $c : i \to t$ est dit *réussi* si $i \in I$ et $t \in T$. Un mot est *reconnu* s'il est l'étiquette d'un calcul réussi. Le *langage reconnu* par l'automate $\mathcal{A}$ est l'ensemble des mots reconnus par $\mathcal{A}$, soit

$$L(\mathcal{A}) = \{w \in A^* \mid \exists c : i \to t,\ i \in I,\ t \in T,\ w = |c|\}$$

Une partie $X \subset A^*$ est *reconnaissable* s'il existe un automate fini $\mathcal{A}$ sur A telle que $X = L(\mathcal{A})$. La famille de toutes les parties reconnaissables de A^* est notée $\mathrm{Rec}(A^*)$.

La terminologie qui vient d'être introduite suggère d'elle-même une représentation graphique d'un automate fini par ce qui est appelé son *diagramme d'états* : les états sont représentés par les sommets d'un graphe (plus précisément d'un multigraphe). Chaque flèche (p, a, q) est représentée par un arc étiqueté qui relie l'état de départ p à l'état d'arrivée q, et qui est étiqueté par l'étiquette a de la flèche. Un calcul n'est autre qu'un chemin dans le multigraphe, et l'étiquette du calcul est la suite des étiquettes des arcs composant le chemin. Dans les figures, on attribue un signe distinctif aux états initiaux et terminaux. Un état initial est muni d'une flèche qui pointe sur lui, un état final est repéré par une flèche qui le quitte. Parfois, nous convenons de réunir en un seul arc plusieurs arcs étiquetés ayant les mêmes extrémités. L'arc porte alors l'ensemble de ces étiquettes. En particulier, s'il y a une flèche (p, a, q) pour toute lettre $a \in A$, on tracera une flèche unique d'étiquette A.

2.2.2 Exemples

L'automate de la figure 2.1 est défini sur l'alphabet $A = \{a, b\}$.

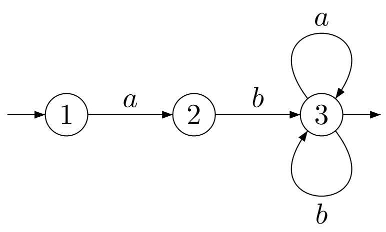

FIG. 2.1 – Automate reconnaissant le langage abA^*.

L'état initial est l'état 1, le seul état final est 3. Tout calcul réussi se factorise en

$$1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{w} 3$$

avec $w \in A^*$. Le langage reconnu est donc bien abA^*. Toujours sur l'alphabet $A = \{a, b\}$, considérons l'automate de la figure 2.2.

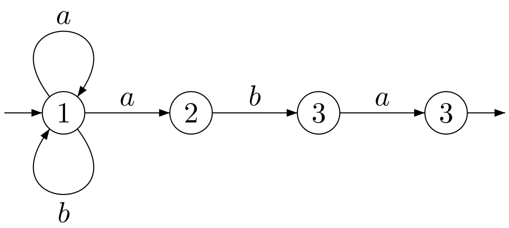

FIG. 2.2 – Automate reconnaissant le langage A^*aba.

Cet automate reconnaît l'ensemble A^*aba des mots qui se terminent par aba. L'automate de la figure 2.3 est défini sur l'alphabet $A = \{a, b\}$.

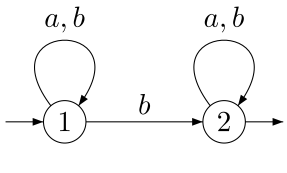

FIG. 2.3 – Automate reconnaissant les mots contenant au moins un b.

Tout calcul réussi contient exactement une fois la flèche $(1, b, 2)$. Un mot est donc reconnu si et seulement s'il contient au moins une fois la lettre b.

L'automate de la figure 2.4 reconnaît l'ensemble des mots contenant un nombre impair de a. Un autre exemple est l'automate vide, ne contenant pas d'états. Il reconnaît le

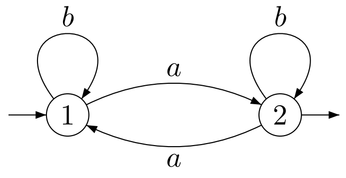

FIG. 2.4 – Automate reconnaissant les mots contenant un nombre impair de a.

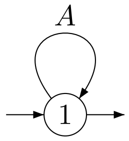

FIG. 2.5 – Tous les mots sont reconnus.

langage vide. A l'inverse, l'automate de la figure 2.5 ayant un seul état qui est à la fois initial et terminal, et une flèche pour chaque lettre $a \in A$ reconnaît tous les mots.

Enfin, le premier des deux automates de la figure 2.6 ne reconnaît que le mot vide, alors que le deuxième reconnaît tous les mots sur A sauf le mot vide, c'est-à-dire le langage A^+.

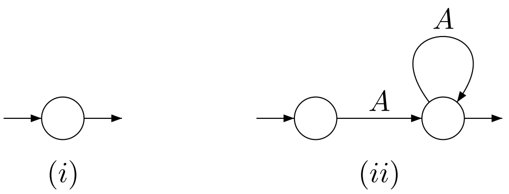

FIG. 2.6 – Automates reconnaissant (i) le mot vide et (ii) tous les mots sauf le mot vide.

2.2.3 Automates déterministes

Un état $q \in Q$ d'un automate $\mathcal{A} = (Q, I, T, \mathcal{F})$ est *accessible* s'il existe un calcul $c : i \to q$ avec $i \in I$. De même, l'état q est *coaccessible* s'il existe un calcul $c : q \to t$ avec $t \in T$. Un automate est *émondé* si tous ses états sont accessibles et coaccessibles. Soit P l'ensemble des états qui sont à la fois accessibles et coaccessibles, et soit $\mathcal{A}^0 = (P, I \cap P, T \cap P, \mathcal{F} \cap P \times A \times P)$. Il est clair que $\mathcal{A}^0$ est émondé. Comme tout calcul réussi de $\mathcal{A}$ ne passe que par des états accessibles et coaccessibles, on a $L(\mathcal{A}^0) = L(\mathcal{A})$.

Emonder un automate fini revient à calculer l'ensemble des états qui sont descendants d'un état initial et ascendants d'un état final. Cela peut se faire en temps linéaire en fonction du nombre de flèches (d'arcs), par les algorithmes de parcours de graphes.

Dans la pratique, les automates déterministes que nous définissons maintenant sont les plus importants, notamment parce qu'ils sont faciles à implémenter.

Un automate $\mathcal{A} = (Q, I, T, \mathcal{F})$ est *déterministe* s'il possède un seul état initial (c'est-à-dire $|I| = 1$) et si

$$(p, a, q), (p, a, q') \in \mathcal{F} \Rightarrow q = q'$$

Ainsi, pour tout $p \in Q$ et tout $a \in A$, il existe au plus un état q dans Q tel que $(p, a, q) \in \mathcal{F}$. On pose alors, pour $p \in Q$ et $a \in A$,

$$p \cdot a = \begin{cases} q & \text{si } (p, a, q) \in \mathcal{F}, \\ \emptyset & \text{sinon.} \end{cases}$$

On définit ainsi une fonction partielle

$$Q \times A \to Q$$

appelée la *fonction de transition* de l'automate déterministe. On l'étend aux mots en posant, pour $p \in Q$,

$$p \cdot \varepsilon = p$$

et pour $w \in A^*$, et $a \in A$,

$$p \cdot wa = (p \cdot w) \cdot a$$

Cette notation signifie que $p \cdot wa$ est défini si et seulement si $p \cdot w$ et $(p \cdot w) \cdot a$ sont définis, et dans l'affirmative, $p \cdot wa$ prend la valeur indiquée.

Avec cette notation on a, avec $I = \{i\}$,

$$L(\mathcal{A}) = \{w \in A^* \mid i \cdot w \in T\}.$$

Un automate est dit *complet* si, pour tout $p \in Q$ et $a \in A$, il existe au moins un état $q \in Q$ tel que $(p, a, q) \in \mathcal{F}$. Si un automate fini $\mathcal{A}$ n'est pas complet, on peut le compléter sans changer le langage reconnu en ajoutant un nouvel état non final s, et en ajoutant les flèches (p, a, s) pour tout couple (p, a) tel que $(p, a, q) \notin \mathcal{F}$ pour tout $q \in Q$. Rendre un automate déterministe, c'est-à-dire le *déterminiser*, est plus difficile, et donne lieu à un algorithme intéressant.

Théorème 2.1. *Pour tout automate fini $\mathcal{A}$, il existe un automate fini déterministe et complet $\mathcal{B}$ tel que*

$$L(\mathcal{A}) = L(\mathcal{B}).$$

Preuve. Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$. On définit un automate déterministe $\mathcal{B}$ qui a pour ensemble d'états l'ensemble $\wp(Q)$ des parties de Q, pour état initial I, et pour ensemble d'états terminaux $V = \{S \subset Q \mid S \cap T \neq \emptyset\}$. On définit enfin la fonction de transition de $\mathcal{B}$ pour $S \in \wp(Q)$ et $a \in A$ par

$$S \cdot a = \{q \in Q \mid \exists s \in S : (s, a, q) \in \mathcal{F}\}.$$

Nous prouvons par récurrence sur la longueur d'un mot w que

$$S \cdot w = \{q \in Q \mid \exists s \in S : s \xrightarrow{w} q\}.$$

Ceci est clair si $w = \varepsilon$, et est vrai par définition si w est une lettre. Posons $w = va$, avec $v \in A^*$ et $a \in A$. Alors comme par définition $S \cdot w = (S \cdot v) \cdot a$, on a $q \in S \cdot w$ si et seulement s'il existe une flèche (p, a, q), avec $p \in S \cdot v$, donc telle qu'il existe un calcul $s \xrightarrow{v} p$ pour un $s \in S$. Ainsi $q \in S \cdot w$ si et seulement s'il existe un calcul $s \xrightarrow{w} q$ avec $s \in S$. Ceci prouve l'assertion.

Maintenant, $w \in L(\mathcal{A})$ si et seulement s'il existe un calcul réussi d'étiquette w, ce qui signifie donc que $I \cdot w$ contient au moins un état final de $\mathcal{A}$, en d'autres termes que $I \cdot w \cap T \neq \emptyset$. Ceci prouve l'égalité $L(\mathcal{A}) = L(\mathcal{B})$. ∎

La démonstration du théorème est constructive. Elle montre que pour un automate $\mathcal{A}$ à n états, on peut construire un automate fini déterministe reconnaissant le même langage et ayant 2^n états. La construction est appelée la *construction par sous-ensembles* (en anglais la « subset construction »).

Exemple 2.2. Sur l'alphabet $A = \{a, b\}$, considérons l'automate $\mathcal{A}$ reconnaissant le langage A^*ab des mots se terminant par ab (figure 2.7).

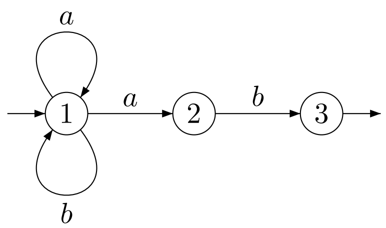

Fig. 2.7 – Un automate reconnaissant le langage A^*ab.

L'application stricte de la preuve du théorème conduit à l'automate déterministe à 8 états de la figure 2.8.

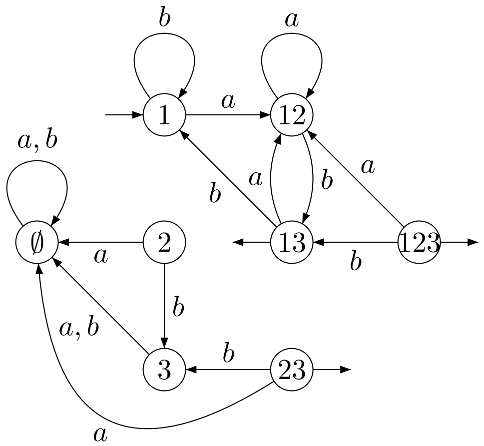

Fig. 2.8 – L'automate déterminisé.

Cet automate a beaucoup d'états inaccessibles. Il suffit de se contenter des états accessibles, et si l'on n'a pas besoin d'un automate complet, il suffit de considérer la partie émondée ; dans notre exemple, on obtient l'automate complet à trois états de la figure 2.9.

◁

En pratique, pour déterminiser un automate $\mathcal{A}$, on ne construit pas l'automate déterministe de la preuve en entier avant de l'émonder ; on combine plutôt les deux étapes en une seule, en faisant une recherche des descendants de l'état initial de l'automate

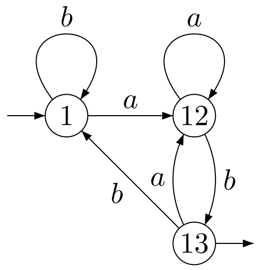

FIG. 2.9 – L'automate déterministe et émondé.

$\mathcal{B}$ pendant sa construction. Ce parcours peut se faire en largeur ou en profondeur, de façon itérative ou récursive.

Pour un parcours en largeur itératif, on maintient un ensemble R d'états de $\mathcal{B}$ déjà construits, et un ensemble E de *transitions* (S, a) qui restent à explorer et qui sont composées d'un état de $\mathcal{B}$ déjà construit et d'une lettre (cet ensemble est organisé en file, par exemple). A chaque étape, on choisit un couple (S, a) dans E et on construit l'état $S \cdot a$ de $\mathcal{B}$ en se servant de l'automate de départ $\mathcal{A}$. Si l'état $S' = S \cdot a$ est connu parce qu'il figure dans R, on passe à la transition suivante ; sinon, on l'ajoute à R et on ajoute à l'ensemble E des transitions à explorer les couples (S', a), pour $a \in A$.

EXEMPLE 2.3. Reprenons l'exemple ci-dessus. Au départ, l'ensemble d'états de l'automate déterministe est réduit à $\{1\}$, et la liste E des transitions à explorer est $(\{1\}, a)$, $(\{1\}, b)$. On débute donc avec

$$R = \{\{1\}\} \qquad E = (\{1\}, a), (\{1\}, b)$$

Choisissons la première transition, ce qui donne le nouvel état $\{1, 2\}$ et les nouvelles transitions $(\{1, 2\}, a), (\{1, 2\}, b)$, d'où

$$R = \{\{1\}, \{1, 2\}\} \qquad E = (\{1\}, b), (\{1, 2\}, a), (\{1, 2\}, b)$$

Les étapes suivantes sont (noter que l'on a le choix de la transition à traiter, ici on les considère dans l'ordre d'arrivée) :

$$\begin{array}{ll}
R = \{\{1\}, \{1, 2\}\} & E = (\{1, 2\}, a), (\{1, 2\}, b) \\
R = \{\{1\}, \{1, 2\}\} & E = (\{1, 2\}, b) \\
R = \{\{1\}, \{1, 2\}, \{1, 3\}\} & E = (\{1, 3\}, a), (\{1, 3\}, b) \\
R = \{\{1\}, \{1, 2\}, \{1, 3\}\} & E = (\{1, 3\}, b) \\
R = \{\{1\}, \{1, 2\}, \{1, 3\}\} & E = \emptyset
\end{array}$$

◁

Comme le suggère la construction, il existe des automates à n états pour lesquels tout automate déterministe équivalent possède de l'ordre de 2^n états. Voici, pour tout entier positif n fixé, un automate à $n + 1$ états (figure 2.10) qui reconnaît le langage $L_n = A^* a A^{n-1}$ sur l'alphabet $A = \{a, b\}$.

Nous allons démontrer que tout automate déterministe reconnaissant L_n a au moins 2^n états. Soit en effet $\mathcal{A} = (Q, i, T)$ un automate déterministe reconnaissant L_n. Alors on a, pour $v, v' \in A^n$:

$$i \cdot v = i \cdot v' \Rightarrow v = v' \tag{2.1}$$

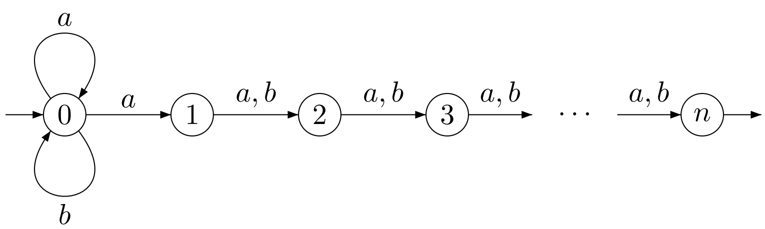

FIG. 2.10 – Un automate à $n+1$ états.

En effet, supposons $v \neq v'$. Il existe alors des mots x, x', w tels que $v = xaw$ et $v' = x'bw$ ou vice-versa. Soit y un mot quelconque tel que wy est de longueur $n-1$. Alors $vy = xawy \in L_n$ et $v'y = x'bwy \notin L_n$, alors que l'égalité $i \cdot v = i \cdot v'$ implique que $i \cdot vy = i \cdot v'y$, et donc que vy et vy' soit sont tous deux dans L_n, soit sont tous deux dans le complément de L_n. D'où la contradiction. L'implication (2.1) montre que $\mathcal{A}$ a au moins 2^n états.

2.2.4 Automates asynchrones

Nous introduisons maintenant une généralisation des automates finis, dont nous montrons qu'en fait ils reconnaissent les mêmes langages. L'extension réside dans le fait d'autoriser également le mot vide comme étiquette d'une flèche. L'avantage de cette convention est une bien plus grande souplesse dans la construction des automates. Elle a en revanche l'inconvénient d'accroître le nondéterminisme des automates.

Un *automate fini asynchrone* $\mathcal{A} = (Q, I, T, \mathcal{F})$ est un automate tel que

$$\mathcal{F} \subset Q \times (A \cup \varepsilon) \times Q\,.$$

Certaines flèches peuvent donc être étiquetées par le mot vide. Les notions de *calcul*, d'*étiquette*, de mot et de langage reconnu s'étendent de manière évidente aux automates asynchrones. La terminologie provient de l'observation que, dans un automate asynchrone, la longueur d'un calcul peut être supérieure à la longueur du mot qui est son étiquette. Dans un automate usuel en revanche, lecture d'une lettre et progression dans l'automate sont rigoureusement synchronisées.

EXEMPLE 2.4. L'automate asynchrone de la figure 2.11 reconnaît le langage a^*b^*.

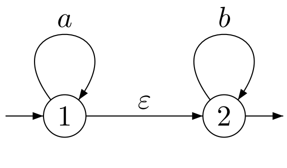

FIG. 2.11 – Un automate asynchrone.

Soit $\mathcal{A} = (Q, I, T)$ un automate asynchrone. La *clôture* d'un état p, notée $C(p)$, est l'ensemble des états accessibles de p par un chemin dont l'étiquette est le mot vide :

$$C(p) = \{q \in Q \mid p \xrightarrow{\varepsilon} q\}\,.$$

Proposition 2.5. *Pour tout automate fini asynchrone $\mathcal{A}$, le langage $L(\mathcal{A})$ est reconnaissable.*

Preuve. Soit $\mathcal{A} = (Q, I, T)$ un automate fini asynchrone sur A. Soit $\mathcal{B} = (P, I, T')$ l'automate fini défini comme suit. L'ensemble P des états de $\mathcal{B}$ est formé de I et de tous les états de $\mathcal{A}$ qui sont extrémités terminales d'une flèche de $\mathcal{A}$ étiquetée par une lettre. Un triplet (p, a, q) est une flèche de $\mathcal{B}$ s'il existe un état $r \in C(p)$ et une flèche (r, a, q) dans $\mathcal{A}$. Les états initiaux de $\mathcal{B}$ sont les états initiaux de $\mathcal{A}$. Enfin, un état $t' \in P$ est final dans $\mathcal{B}$ si $C(p) \cap T \neq \emptyset$. On a alors $L(\mathcal{A}) = L(\mathcal{B})$. ■

Reprenons notre exemple de l'automate de la figure 2.11. La construction de la preuve conduit à l'automate « synchrone » de la figure 2.12.

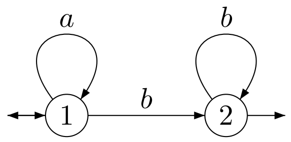

Fig. 2.12 – L'automate précédent « synchrone ».

En effet, il existe dans l'automate d'origine un calcul de longueur 2, de l'état 1 à l'état 2, l'un portant l'étiquette b. Cet automate reconnaît le langage a^*b^* : le mot vide est reconnu par le nouvel état terminal 1.

2.3 Langages rationnels

Dans cette section, nous définissons une autre famille de langages, les langages rationnels, et nous prouvons qu'ils coïncident avec les langages reconnaissables. Grâce à ce résultat, dû à Kleene, on dispose de deux caractérisations très différentes d'une même famille de langages.

2.3.1 Langages rationnels : définitions

Soit A un alphabet. Les *opérations rationnelles* sur les parties de A^* sont les opérations suivantes :

union	$X \cup Y$;
produit	$XY = \{xy \mid x \in X,\ y \in Y\}$;
étoile	$X^* = \{x_1 \cdots x_n \mid n \geq 0, x_1, \ldots, x_n \in X\}$.

Une famille de parties de A^* est *rationnellement fermée* si elle est fermée pour les trois opérations rationnelles. Les *langages rationnels* de A^* sont les éléments de la plus petite famille rationnellement fermée de A^* qui contient les singletons (c'est-à-dire les langages réduits à un seul mot) et le langage vide. Cette famille est notée $\mathrm{Rat}(A^*)$. Une expression d'un langage comme combinaison finie d'unions, de produits et d'étoiles de singletons est une *expression rationnelle.* Considérons quelques exemples.

Le langage abA^*, avec $A = \{a, b\}$, est rationnel : il est le produit des singletons a et b par l'étoile de A qui est lui-même l'union des deux lettres a et b. De même, le langage

A^*aba est rationnel. Toujours sur $A = \{a, b\}$, le langage L des mots qui contiennent un nombre pair de a est rationnel : c'est le langage $L = (ab^*a \cup b)^*$.

2.3.2 Expressions rationnelles

Le fait qu'un langage est rationnel est immédiat si l'on dispose d'une expression rationnelle qui le décrit.

Dans ce paragraphe, nous étudions les expressions rationnelles en elles-mêmes, en faisant une distinction entre une expression et le langage qu'elle décrit, un peu comme l'on fait la différence entre une expression arithmétique et la valeur numérique qu'elle représente. L'approche est donc purement syntaxique, et le langage représenté par une expression peut être considéré comme résultant d'une évaluation de l'expression.

Les manipulations (algébriques ou combinatoires) d'expressions permettent de construire des automates efficaces reconnaissant le langage représenté par des opérations qui sont proches de l'analyse syntaxique, et qui ne font pas intervenir le langage lui-même. Elles se prêtent donc bien à une implémentation effective. Une autre application est la recherche de motifs dans un texte, comme elle est réalisée dans un traitement de texte par exemple.

Soit A un alphabet fini, soient 0 et 1 deux symboles ne figurant pas dans A, et soient $+, \cdot, ^*$ trois symboles de fonctions, les deux premiers binaires, le dernier unaire. Les *expressions* sur $A \cup \{0, 1, +, \cdot, ^*\}$ sont les termes bien formés sur cet ensemble de symboles.

Exemple 3.1. Sur $A = \{a, b\}$ le terme $a((a + a \cdot b)^*b)$ est une expression. Comme souvent, il convient de représenter une expression par un arbre qui, sur cet exemple, est l'arbre de la figure 3.1.

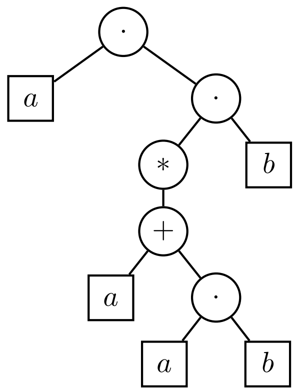

Fig. 3.1 – L'expression rationnelle $a((a + a \cdot b)^*b)$.

◁

Les parenthèses ne sont utilisées que lorsqu'elles sont nécessaires, en convenant que l'opération * a priorité sur $\cdot$ qui a priorité sur $+$; on omet également le signe de multiplication quand cela ne prête pas à confusion.

Souvent, deux expressions ne diffèrent que par des différences mineures, comme les expressions $0 + a$ et a, ou les expressions $a + b$ et $b + a$. Formellement, elles sont distinctes, mais pour la manipulation de tous les jours il convient d'identifier deux expressions aussi semblables. Les *expressions rationnelles* sur A sont les éléments de l'algèbre quotient obtenue en convenant que

(i) l'opération $+$ est idempotente, associative et commutative,

(ii) l'opération $\cdot$ est associative et distributive par rapport à l'opération $+$,

(iii) les équations suivantes sont vérifiées pour tout terme e :

$$\begin{aligned} 0 + e = e &= e + 0 \\ 1 \cdot e = e &= e \cdot 1 \\ 0 \cdot e = 0 &= e \cdot 0 \end{aligned}$$

(iv) on a :

$$0^* = 1^* = 1$$

Exemple 3.2. Par exemple, $(0 + a(1 + b \cdot 1))^*$ et $(ab + a)^*$ sont la *même* expression rationnelle, alors que $1 + a^*a$ et a^* sont deux expressions différentes (même si elles décrivent le même langage).

On note $\mathcal{E}(A)$ l'algèbre des expressions rationnelles ainsi obtenue. La convention d'identifier certaines expressions selon les règles que nous venons d'édicter allège très considérablement l'écriture et l'exposé ; tout ce qui suit pourrait se faire également en ne raisonnant que sur les termes ou les arbres.

L'important est toutefois que l'*égalité* de deux expressions est *facilement décidable* sur les expressions elles-mêmes. Pour ce faire, on met une expression, donnée disons sous forme d'arbre, en « forme normale » en supprimant d'abord les feuilles 0 ou 1 quand c'est possible en appliquant les égalités (iii) ci-dessus, puis en distribuant le produit par rapport à l'addition pour faire « descendre » au maximum les produits. Lorsque les deux expressions sont en forme normale, on peut décider récursivement si elles sont égales : il faut et il suffit pour cela qu'elles aient même symbole aux racines, et si les *suites* d'expressions des fils obtenues en faisant jouer l'associativité de $\cdot$ et les *ensembles* d'expressions des fils modulo l'associativité, l'idempotence et la commutativité de $+$ sont égaux.

Exemple 3.3. En appliquant les règles de simplification à l'expression $(0+a(1+b\cdot 1))^*$, on obtient $(a(1 + b))^*$. Par distributivité et simplification, elle donne $(a + ab)^*$, et les deux expressions fils de la racine sont les mêmes dans cette expression, et dans $(ab+a)^*$. $\triangleleft$

Remarque. Dans ce qui précède ne figure pas l'opération $e \mapsto e^+$. C'est pour ne pas alourdir l'exposé que nous considérons cette opération comme une abréviation de ee^* ; on pourrait aussi bien considérer cette opération comme opération de base.

A chaque expression rationnelle sur A est associé naturellement un langage rationnel sur A, par l'application

$$L : \mathcal{E}(A) \to \wp(A^*)$$

définie par récurrence comme suit :

$$\begin{aligned}
&L(0) = \emptyset, \quad L(1) = \{\varepsilon\}, \quad L(a) = \{a\}, \\
&L(e + e') = L(e) \cup L(e'), \quad L(e \cdot e') = L(e)L(e'), \\
&L(e^*) = L(e)^*
\end{aligned}$$

Pour une expression e, le langage $L(e)$ est le langage *décrit* ou *dénoté* par e. Deux expressions e et e' sont dites *équivalentes* lorsqu'elles dénotent le même langage, c'est-à-dire lorsque $L(e) = L(e')$. On écrit alors $e \approx e'$.

PROPOSITION 3.4. *Pour toutes expressions rationnelles e et f, on a les formules suivantes :*

$$\begin{aligned}
(ef)^* &\approx 1 + e(fe)^*f \\
(e + f)^* &\approx e^*(fe^*)^* \\
e^* &\approx (1 + e + \ldots + e^{p-1})(e^p)^* \qquad p > 1
\end{aligned}$$

Cette proposition est un corollaire immédiat du lemme suivant :

LEMME 3.5. *Quelles que soient les parties X et Y de A^*, on a*

$$(XY)^* = \varepsilon \cup X(YX)^*Y \tag{3.1}$$

$$(X \cup Y)^* = X^*(YX^*)^* \tag{3.2}$$

$$X^* = (\varepsilon \cup X \cup \cdots \cup X^{p-1})(X^p)^* \qquad p > 1 \tag{3.3}$$

Preuve. Pour établir (3.1), montrons d'abord que $(XY)^* \subset \varepsilon \cup X(YX)^*Y$. Pour cela, soit $w \in (XY)^*$. Alors $w = x_1y_1 \cdots x_ny_n$ pour $n \geq 0$ et $x_i \in X$, $y_i \in Y$. Si $n = 0$, alors $w = \varepsilon$, sinon $w = x_1vy_n$, avec $v = y_1 \cdots x_n \in (YX)^*$. Ceci prouve l'inclusion. L'inclusion réciproque se montre de manière similaire.

Pour prouver (3.2), il suffit d'établir

$$(X \cup Y)^* \subset \varepsilon \cup X^*(YX^*)^*$$

l'inclusion réciproque étant évidente. Soit $w \in (X \cup Y)^*$. Il existe $n \geq 0$ et $z_1, \ldots, z_n \in X \cup Y$ tels que $w = z_1 \cdots z_n$. En groupant les z_i consécutifs qui appartiennent à X, ce produit peut s'écrire

$$w = x_0y_1x_1 \cdots y_mx_m$$

avec $x_0, \ldots, x_m \in X^*$ et $y_0, \ldots, y_m \in Y$. Par conséquent $w \in \varepsilon \cup X^*(YX^*)^*$.

Considérons enfin (3.3). On a

$$(\varepsilon \cup X \cup \cdots \cup X^{p-1})(X^p)^* = \bigcup_{k=0}^{p-1} \bigcup_{m \geq 0} X^{k+mp} = X^* \qquad \blacksquare$$

2.3.3 Algorithme de Thompson

Nous étudions ici un procédé de calcul efficace d'un automate reconnaissant le langage dénoté par une expression rationnelle, appelé algorithme de Thompson. Soit A un alphabet, et soit e une expression rationnelle sur A. La *taille* de e, notée $|e|$, est le nombre de symboles figurant dans e. Plus précisément, on a

$$|0| = |1| = |a| = 1 \quad \text{pour} \quad a \in A$$
$$|e + f| = |e \cdot f| = 1 + |e| + |f|, \quad |e^*| = 1 + |e|$$

Nous allons construire, pour toute expression e, un automate reconnaissant $L(e)$ qui a des propriétés particulières. Un automate asynchrone $\mathcal{A}$ est dit *normalisé* s'il vérifie les conditions suivantes :

(i) il existe un seul état initial, et un seul état final, et ces deux états sont distincts ;
(ii) aucune flèche ne pointe sur l'état initial, aucune flèche ne sort de l'état final ;
(iii) tout état est soit l'origine d'exactement une flèche étiquetée par une lettre, soit l'origine d'au plus deux flèches étiquetées par le mot vide ε.

Notons que le nombre de flèches d'un automate normalisé est au plus le double du nombre de ses états.

PROPOSITION 3.6. *Pour toute expression rationnelle e de taille m, il existe un automate normalisé reconnaissant $L(e)$, et dont le nombre d'états est au plus $2m$.*

Preuve. Elle est constructive. On procède par récurrence sur la taille de e.

Pour $e = 0$, $e = 1$, et $e = a$, où $a \in A$, les automates de la figure 3.2 donnent des automates normalisés ayant 2 états.

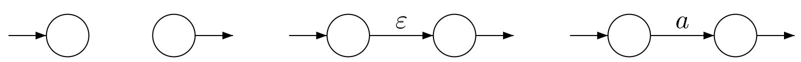

FIG. 3.2 – Automates pour l'ensemble vide, pour ε et pour a.

Si $e = e' + e''$, soient $\mathcal{A}' = (Q', i', t')$ et $\mathcal{A}'' = (Q'', i'', t'')$ deux automates normalisés reconnaissant des langages $X' = L(e')$ et $X'' = L(e'')$. On suppose Q' et Q'' disjoints.

L'automate

$$\mathcal{A} = (Q' \cup Q'' \cup \{i, t\}, i, t)$$

où i et t sont deux nouveaux états distincts et dont les flèches sont, en plus de celles de $\mathcal{A}'$ et de A'', les quatre flèches (i, ε, i'), (i, ε, i''), (t', ε, t), (t'', ε, t) reconnaît $X' \cup X'' = L(e)$ (voir figure 3.3). L'automate est normalisé, et $|Q| \leq 2|e|$. Si $e = e' \cdot e''$, considérons l'automate

$$\mathcal{A} = ((Q' \setminus t') \cup Q'', i', t'')$$

obtenu en « identifiant » t' et i'', c'est-à-dire en remplaçant toute flèche aboutissant en t' par la même flèche, mais aboutissant en i'' (voir figure 3.3). Cet automate reconnaît

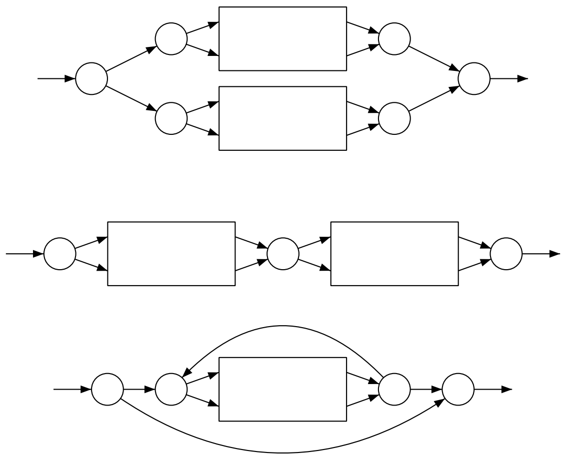

FIG. 3.3 – Les automates pour l'union, le produit et l'étoile.

le langage $X'X''$; clairement, son nombre d'états est majoré par $2|e|$. Enfin, si $e = e'^*$, le troisième automate de la figure 3.3 est un automate

$$\mathcal{A} = (Q' \cup \{i, t\}, i, t)$$

qui, en plus des flèches de $\mathcal{A}'$, possède les quatre flèches (i, ε, i'), (i, ε, t), (t', ε, i'), (t', ε, t) reconnaît le langage $X'^* = L(e)$. Il est normalisé et a 2 états de plus que $\mathcal{A}'$. ■

Notons le corollaire suivant de cette construction.

PROPOSITION 3.7. *Soit A un alphabet. Tout langage rationnel de A^* est reconnaissable :* $\mathrm{Rat}(A^*) \subset \mathrm{Rec}(A^*)$. ■

EXEMPLE 3.8. Pour l'expression $(a + b)^*b(1 + a)(a + b)^*$, la construction de la preuve produit l'automate de la figure 3.4, dans laquelle les flèches non marquées portent comme étiquette le mot vide. Il a 21 états et 27 flèches. Observons qu'il existe de

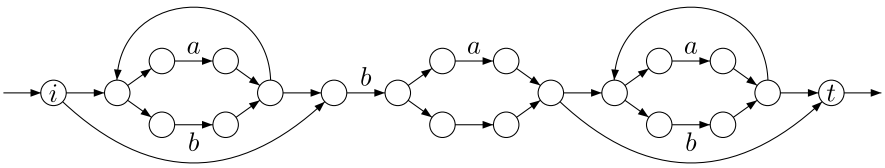

FIG. 3.4 – L'automate pour l'expression $(a + b)^*b(a + 1)(a + b)^*$.

nombreux chemins dont l'étiquette est vide. ◁

2.3.4 Le théorème de Kleene

Le théorème suivant est dû à Kleene :

THÉORÈME 3.9 (de Kleene). *Soit A un alphabet fini. Alors les langages rationnels et reconnaissables sur A coïncident :* $\mathrm{Rat}(A^*) = \mathrm{Rec}(A^*)$.

Cet énoncé est remarquable dans la mesure où il donne deux caractérisations très différentes d'une même famille de langages : l'automate fini est un moyen de calcul, et se donner un langage par un automate fini revient à se donner un algorithme pour vérifier l'appartenance de mots au langage. Au contraire, une expression rationnelle décrit la structure syntaxique du langage. Elle permet en particulier des manipulations, et des opérations entre langages.

La démonstration du théorème de Kleene est en deux parties. Une première partie consiste à montrer que tout langage rationnel est reconnaissable, c'est-à-dire à prouver l'inclusion $\mathrm{Rat}(A^*) \subset \mathrm{Rec}(A^*)$. Cette inclusion a été prouvée dans la section précédente.

L'inclusion opposée, à savoir, $\mathrm{Rec}(A^*) \subset \mathrm{Rat}(A^*)$ se montre en exhibant, pour tout automate fini, une expression rationnelle. Il existe plusieurs algorithmes pour obtenir cette expression.

Nous présentons une méthode appelée l'algorithme BMC (d'après leurs auteurs, Brzozowski et McCluskey).

Algorithme BMC. Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$ un automate. On cherche une expression dénotant le langage reconnu par $\mathcal{A}$. On procède par suppression successive de flèches et d'états, en remplaçant d'autres étiquettes par des expressions rationnelles.
(1) Ajouter à $\mathcal{A}$ deux nouveaux états, notés α et ω, et les flèches (α, ε, i) pout $i \in I$ et (t, ε, ω) pour $t \in T$.
(2) Itérer les réductions suivantes tant que possible :
 – s'il existe deux flèches $p(, e, q)$ et (p, f, q), les remplacer par la flèche $(p, e + f, q)$
 – supprimer un état q (autre que α et ω) et remplacer, pour tous états $p, r \neq q$, les flèches (p, e, q), (q, f, q), (q, g, r), par la flèche (p, ef^*g, r)

Cet algorithme termine, parce que l'on diminue le nombre de flèches et d'états, jusqu'à obtenir une seule flèche (α, e, ω). Il est clair que e est une expression pour le langage $L(\mathcal{A})$.

EXEMPLE 3.10. Considérons l'automate de la figure 3.5.

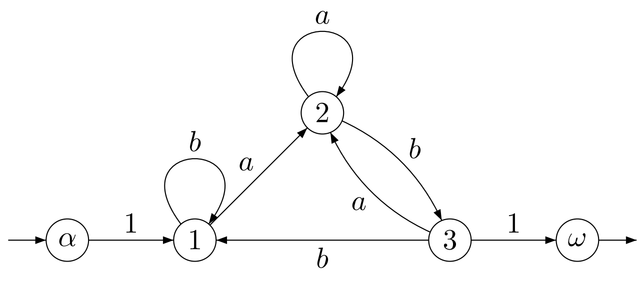

FIG. 3.5 – Un automate, augmenté de deux états α et ω.

Supprimons l'état 2. Les couples de flèches concernées sont $(1, a, 2)$ et $(2, b, 3)$ d'une part et $(3, a, 2)$ et $(2, b, 3)$ d'autre part. Le premier couple produit une flèche de 1 vers 3

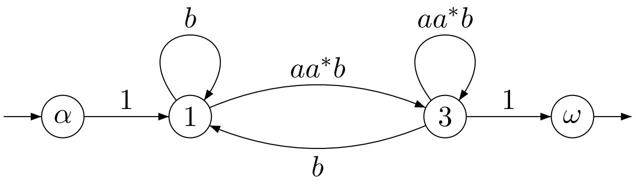

FIG. 3.6 – L'automate, après suppression de l'état 2.

d'étiquette aa^*b, le deuxième une boucle autour de 3, aussi avec l'étiquette aa^*b, ce qui donne l'automate de la figure 3.6. Nous pouvons maintenant supprimer par exemple l'état 1. Cela donne une flèche de 1 à 3 d'étiquette ba^+b (on écrit a^+ à la place de aa^*), et une boucle supplémentaire, de cette étiquette, autour de 3, soit l'automate de la figure 3.7 Les deux boucles sont combinées en une seule, dont l'étiquette est

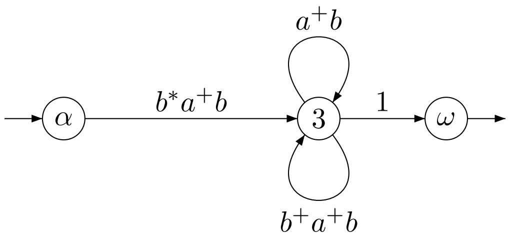

FIG. 3.7 – L'automate précédent, après suppression de l'état 1.

$b^+a^+b + a^+b$. En fait, cette expression dénote le même langage que b^*a^+b. On a donc plus simplement l'automate de la figure 3.8. Il ne reste plus qu'à supprimer l'état 3. On

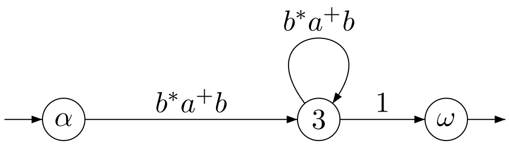

FIG. 3.8 – L'automate précédent, après sommation des flèches.

obtient l'automate de la figure 3.9.

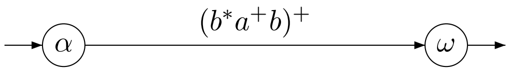

FIG. 3.9 – L'automate complètement réduit.

Le langage reconnu est donc $(b^*a^+b)^+$. Pour transformer cette expression en une expression plus simple, on observe que $(b^*a^+b)^+ = (b^*a^+b)^*b^*a^+b$ et que $(b^*a^+b)^*b^* = (b + a^+b)^*$. Comme $b + a^+b = a^*b$, on a donc $(b^*a^+b)^+ = (a^*b)^*a^+b = (a^*b)^*a^*ab = (a + b)^*ab$. $\triangleleft$

2.3.5 Complément : l'algorithme de McNaughton et Yamada

Voici une autre preuve de la proposition :

PROPOSITION 3.11. *Soit A un alphabet fini. Tout langage reconnaissable de A^* est rationnel :* $\mathrm{Rec}(A^*) \subset \mathrm{Rat}(A^*)$.

Preuve. Soit $\mathcal{A} = (Q, I, T)$ un automate fini sur A, et soit X le langage reconnu. Numérotons les états de manière que $Q = \{1, \ldots, n\}$. Pour i, j dans Q, soit $X_{i,j} = \{w \mid i \xrightarrow{w} j\}$, et pour $k = 0, \ldots, n$, soit $X_{i,j}^{(k)}$ l'ensemble des étiquettes des calculs de longueur strictement positive de la forme

$$i \to p_1 \to \ldots \to p_s \to j, \qquad s \geq 0, \quad p_1, \ldots, p_s \leq k$$

Observons que $X_{i,j}^{(0)} \subset A$, et donc que chaque $X_{i,j}^{(0)}$ est une partie rationnelle, parce que l'alphabet est fini. Ensuite, on a

$$X_{i,j}^{(k+1)} = X_{i,j}^{(k)} \cup X_{i,k+1}^{(k)} \left(X_{k+1,k+1}^{(k)}\right)^* X_{k+1,j}^{(k)} \tag{3.4}$$

Cette formule montre, par récurrence sur k, que chacun des langages $X_{i,j}^{(k)}$ est rationnel. Or

$$X_{i,j} = \begin{cases} X_{i,j}^{(n)} & \text{si } i \neq j \\ \varepsilon \cup X_{i,j}^{(n)} & \text{si } i = j \end{cases} \tag{3.5}$$

et

$$X = \bigcup_{\substack{i \in I \\ t \in T}} X_{i,t} \tag{3.6}$$

et par conséquent les langages $X_{i,j}$ sont rationnels. Donc X est également rationnel. ∎

Les formules (3.4)–(3.6) permettent de calculer effectivement une expression (expression rationnelle) pour le langage reconnu par un automate fini. Cette méthode est appelée l'algorithme de MacNaughton et Yamada, d'après ses créateurs. Il est très simple, mais pas très efficace dans la mesure où il faut calculer les $O(n^3)$ expressions $X_{i,j}^{(k)}$ pour un automate à n états.

2.3.6 Systèmes d'équations linéaires

Une façon parfois plus commode — surtout pour les calculs à la main — de déterminer l'expression rationnelle du langage reconnu par un automate consiste à résoudre un système d'équations linéaires naturellement associé à tout automate fini.

Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$ un automate fini sur A. Le *système d'équations* (linéaire droit) associé à $\mathcal{A}$ est

$$X_p = \bigcup_{q \in Q} E_{p,q} X_q \cup \delta_{p,T} \qquad p \in Q$$

où

$$E_{p,q} = \{a \in A \mid (p,a,q) \in \mathcal{F}\}, \qquad p,q \in Q$$
$$\delta_{p,T} = \begin{cases} \{\varepsilon\} & \text{si } p \in T \\ \emptyset & \text{sinon} \end{cases}$$

Considérons par exemple l'automate de la figure 3.10.

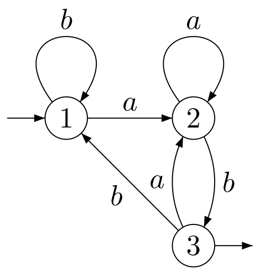

FIG. 3.10 – Quel est le langage reconnu par cet automate ?

Le système d'équations associé à cet automate est

$$\begin{aligned} X_1 &= bX_1 \cup aX_2 \\ X_2 &= \phantom{bX_1 \cup{}} aX_2 \cup bX_3 \\ X_3 &= bX_1 \cup aX_2 \phantom{{}\cup bX_3} \cup \varepsilon \end{aligned}$$

Une *solution* du système d'équations est une famille $(L_p)_{p\in Q}$ de langages qui vérifie le système d'équations.

PROPOSITION 3.12. *Soit $\mathcal{A} = (Q, I, T)$ un automate fini sur A, et posons*

$$L_p(\mathcal{A}) = \{w \in A^* \mid \exists c : p \to t \in T,\ |c| = w\}$$

Alors la famille $(L_p(\mathcal{A}))_{p\in Q}$ est l'unique solution du système d'équations associé à $\mathcal{A}$.

Le langage $L_p(\mathcal{A})$ est l'ensemble des mots reconnus par $\mathcal{A}$ en prenant p pour état initial, de sorte que

$$L(\mathcal{A}) = \bigcup_{p\in I} L_p(\mathcal{A})$$

Nous commençons par le cas particulier d'un système d'équations réduit à une seule équation. La démarche sera la même dans le cas général.

LEMME 3.13 (Lemme d'Arden). *Soient E et F deux langages. Si $\varepsilon \notin E$, alors l'équation $X = EX \cup F$ possède une solution unique, à savoir le langage E^*F.*

Preuve. Posons $L = E^*F$. Alors $EL \cup F = EE^*F \cup F = (EE^* \cup \varepsilon)F = E^*F = L$, ce qui montre que L est bien solution de l'équation. Supposons qu'il y ait deux solutions L et L'. Comme

$$L - L' = (EL \cup F) - L' = EL - L' \subset EL - EL'$$

et que $EL - EL' \subset E(L - L')$, on a $L - L' \subset E(L - L') \subset E^n(L - L')$ pour tout $n > 0$. Mais comme le mot vide n'appartient pas à E, cela signifie qu'un mot de $L - L'$ a pour longueur au moins n pour tout n. Donc $L - L'$ est vide, et par conséquent $L \subset L'$ et de même $L' \subset L$. ■

Preuve de la proposition. Pour montrer que les langages $L_p(\mathcal{A})$, $(p \in Q)$, constituent une solution, posons

$$L'_p = \bigcup_{q \in Q} E_{p,q} L_q(\mathcal{A}) \cup \delta_{p,T}$$

et vérifions que $L_p(\mathcal{A}) = L'_p$ pour $p \in Q$. Si $\varepsilon \in L_p(\mathcal{A})$, alors $p \in T$, donc $\delta_{p,T} = \{\varepsilon\}$ et $\varepsilon \in L'_p$, et réciproquement. Soit $w \in L_p(\mathcal{A})$ de longueur > 0 ; il existe un calcul $c : p \to t$ pour un $t \in T$ d'étiquette w. En posant $w = av$, avec a une lettre et v un mot, le calcul c se factorise en $p \xrightarrow{a} q \xrightarrow{v} t$ pour un état q. Mais alors $a \in E_{p,q}$ et $v \in L_q(\mathcal{A})$, donc $w \in L'_p$. L'inclusion réciproque se montre de la même façon.

Pour prouver l'unicité de la solution, on procède comme dans la preuve du lemme d'Arden. Soient $(L_p)_{p \in Q}$ et $(L'_p)_{p \in Q}$ deux solutions du système d'équations, et posons $M_p = L_p - L'_p$ pour $p \in Q$. Alors

$$M_p = \bigcup_{q \in Q} E_{p,q} L_q \cup \delta_{p,T} - L'_p \subset \bigcup_{q \in Q} (E_{p,q} L_q - E_{p,q} L'_q) \subset \bigcup_{q \in Q} E_{p,q} M_q$$

A nouveau, ces inclusions impliquent que $M_p = \emptyset$ pour tout p. Supposons en effet le contraire, soit w un mot de longueur minimale dans

$$M' = \bigcup_{p \in Q} M_p$$

et soit p un état tel que $w \in M_p$. Alors

$$w \in \bigcup_{q \in Q} E_{p,q} M_q$$

donc w est le produit d'une lettre et d'un mot v de M' ; mais alors v est plus court que w, une contradiction. ■

Pour *résoudre* un système d'équations, on peut procéder par élimination de variables (c'est la méthode de Gauss). Dans notre exemple, on substitue la troisième équation dans la deuxième, ce qui donne

$$X_2 = (a \cup ba) X_2 \cup b^2 X_1 \cup b$$

qui, par le lemme d'Arden, équivaut à l'équation

$$X_2 = (a \cup ba)^* (b^2 X_1 \cup b)$$

Cette expression pour X_2 est substituée dans la première équation du système ; on obtient

$$X_1 = (b \cup a(a \cup ba)^* b^2) X_1 \cup a(a \cup ba)^* b$$

d'où, à nouveau par le lemme d'Arden,

$$X_1 = (b \cup a(a \cup ba)^* b^2)^* a(a \cup ba)^* b$$

Il n'est pas du tout évident que cette dernière expression soit égale à $\{a,b\}^*ab$. Pour le montrer, observons d'abord que

$$(a \cup ba)^* = a^*(ba^+)^*$$

(rappelons que $X^+ = XX^* = X^*X$ pour tout X) en utilisant la règle $(U \cup V)^* = U^*(VU^*)^*$, d'où

$$a(a \cup ba)^*b = a^+(ba^+)^*b = (a^+b)^+$$

Il en résulte que

$$b \cup a(a \cup ba)^*b^2 = [\varepsilon \cup (a^+b)^+]b = (a^+b)^*b$$

d'où

$$X_1 = [(a^+b)^*b]^*(a^+b)^*(a^+b) = (a^+b \cup b)^*(a^+b) = (a^*b)^*a^*ab = \{a,b\}^*ab$$

2.4 Le lemme d'itération

Bien entendu, tout langage ne peut être reconnu par un automate fini. On peut prouver directement que certains langages ne sont par reconnaissables, par une discussion sur la nature des automates qui les reconnaîtraient. Il existe une propriété, connue sous le terme de *lemme d'itération* ou *lemme de l'étoile* (en anglais "pumping lemma") qui donne une condition nécessaire pour qu'un langage soit reconnaissable. Il est alors en général assez commode, à l'aide de cette propriété, de prouver qu'un langage n'est pas reconnaissable.

La propriété d'itération encapsule le fait que, dans automate fini, tout calcul assez long doit contenir un circuit.

Proposition 4.1. (Lemme d'itération) *Soit K un langage reconnu par un automate à N états. Pour tout mot $z \in K$, et pour toute factorisation $z = xyx'$ telle que $|y| \geq N$, il existe une factorisation $y = uvw$ telle que*

(i) $|uv| \leq N$
(ii) $v \neq \varepsilon$
(iii) $xuv^nwx \in K$ *pour tout* $n \geq 0$

On peut illustrer l'emploi de cette proposition par un protocole entre deux acteurs, disons Alice et Bob. Alice veut convaincre Bob que le langage K qu'elle considère est reconnaissable, et Bob veut plutôt prouver le contraire. La discussion suivante s'engage entre Alice et Bob :

Alice	Bob
K est reconnaissable !	Combien d'états ?
N états	Voici un mot z
z est dans K ? ok	Voici une factorisation $z = xyx'$
$\|y\| \geq N$? ok	
Voici une factorisation $y = uvw$	$\|uv\| \leq N$? ok
	v non vide ? ok
	$xuv^nwx' \in K$ pour tout $n \geq 0$?

Preuve. Soit $\mathcal{A} = (Q, I, T)$ un automate à N états reconnaissant le langage K, soit $z = xyx'$ un mot de K avec $|y| \geq N$. On pose $y = a_1 a_2 \cdots a_N y'$, avec $a_1, \ldots, a_N$ des lettres. Le mot z étant reconnu, il existe un chemin réussi $c : i \to t$ dans $\mathcal{A}$ d'étiquette z. Décomposons ce chemin :

$$c : i \xrightarrow{x} q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots q_{N-1} \xrightarrow{a_N} q_N \xrightarrow{y'} p \xrightarrow{z'} t$$

Parmi les $n+1$ états $q_0, \ldots, q_N$, il y en a deux d'égaux par le principe des tiroirs. Soient k, ℓ avec $0 \leq k < \ell \leq N$ tels que $q_k = q_\ell$. Posons alors $u = a_1 \cdots a_k$, $v = a_{k+1} \cdots a_\ell$, $w = a_{\ell+1} \cdots a_N y'$. On a donc $q_0 \xrightarrow{u} q_k$, $q_k \xrightarrow{v} q_\ell$, $q_\ell \xrightarrow{w} p$. Il en résulte qu'il existe des chemins $q_0 \to p$ d'étiquette $uv^n w$ pour tout entier $n \geq 0$. Par conséquent, tous les mots $xuv^n wx'$, pour $n \geq 0$ sont reconnus. Par ailleurs, $|v| = \ell - k > 0$, donc v n'est pas le mot vide, et $|uv| = \ell \leq N$. ∎

Voici deux exemples d'utilisation du lemme d'itération pour prouver qu'un langage n'est pas reconnaissable.

Exemple 4.2. *Le langage $\{a^n b^n \mid n \geq 0\}$ n'est pas reconnaissable.* Appelons K ce langage, et soit N l'entier du lemme d'itération. Soit $z = a^N b^N \in K$ et soit $x = \varepsilon$, $y = a^N$, $x' = b^N$. On a $|y| = N$ et il existe donc une factorisation $y = uvw$ telle que $uv^n wx' \in K$ pour tout n. Or u, v, w sont des puissances de la lettre a. Posons $u = a^k$, $v = a^\ell$ et $w = a^m$. On a $k + \ell + m = N$ et $\ell \neq 0$. Mais alors $uv^2 wx' = a^{N+\ell} b$ n'est pas dans K. D'où la contradiction qui montre que K n'est pas reconnaissable.

Exemple 4.3. *L'ensemble des écritures binaires des nombres premiers n'est pas un langage reconnaissable.* Soit L l'ensemble des écritures binaires des nombres premiers. Pour tout mot binaire w, on note $n(w)$ le nombre décrit par w. Ainsi, $n(100) = 4$. Supposons que L soit reconnaissable par un automate à N états, et soit p un nombre premier tel que $p > 2^N$. Soit z l'écriture binaire de p, de sorte que $n(z) = p$, et utilisons le lemme d'itération avec $y = z$. On a $|z| \geq N$ parce que $p > 2^N$. Il existe une factorisation $z = uvw$ telle que $n(uv^n w)$ est un nombre premier pour tout $n \geq 0$. Maintenant, le nombre $q = n(uv^p w)$ vérifie

$$n(uv^p w) = n(u)2^{|w|+p|v|} + n(v)2^{|w|}(1 + 2^{|v|} + 2^{2|v|} + \cdots + 2^{(p-1)|v|}) + n(w)$$

Il est commode de poser $k = 2^{|v|}$. On a alors l'écriture plus simple

$$n(uv^p w) = n(u)2^{|w|}k^p + n(v)2^{|w|}(1 + k + k^2 + \cdots + k^{(p-1)}) + n(w)$$

Comme $2 \leq k \leq 2^N < p$, on peut utiliser le petit théorème de Fermat qui dit que $k^p \equiv k \bmod p$, parce que p est premier. Il en résulte que $1+k+k^2+\cdots+k^{(p-1)} \equiv 1 \bmod p$ (en effet, posons $f = 1 + k + k^2 + \cdots + k^{(p-1)}$; alors $(k-1)f = k^p - 1 \equiv k - 1 \bmod p$, donc $f \equiv 1 \bmod p$). Il en résulte que

$$n(uv^p w) \equiv n(u)2^{|w|}k + n(v)2^{|w|} + n(w) = p \equiv 0 \bmod p$$

et donc que p divise q et q n'est pas premier.

2.5 Opérations

PROPOSITION 5.1. *La famille des langages reconnaissables sur un alphabet A est fermée pour les opérations booléennes, c'est-à-dire pour l'union, l'intersection et la complémentation.*

Preuve. Soient X et X' deux langages reconnaissables de A^*, et soient $\mathcal{A} = (Q, i, T)$ et $\mathcal{A}' = (Q', i', T')$ deux automates finis déterministes complets tels que $X = L(\mathcal{A})$ et $X' = L(\mathcal{A}')$. Considérons l'automate déterministe complet

$$\mathcal{B} = (Q \times Q', (i, i'), S)$$

dont la fonction de transition est définie par

$$(p, p') \cdot a = (p \cdot a, p' \cdot a)$$

pour tout couple d'états (p, p') et toute lettre $a \in A$. Alors on a

$$(p, p') \cdot w = (p \cdot w, p' \cdot w)$$

pour tout mot w, comme on le vérifie immédiatement en raisonnant par récurrence sur la longueur de w. Il résulte de cette équation que pour $S = T \times T'$, on obtient $L(\mathcal{B}) = X \cap X'$, et pour $S = (T \times Q') \cup (Q \times T')$, on obtient $L(\mathcal{B}) = X \cup X'$. Enfin, pour $S = T \times (Q' - T')$, on a $L(\mathcal{B}) = X - X'$. ∎

COROLLAIRE 5.2. *Un langage X est reconnaissable si et seulement si $X - \{\varepsilon\}$ est reconnaissable.* ∎

PROPOSITION 5.3. *Si X est reconnaissable, alors X^* est reconnaissable ; si X et Y sont reconnaissables, alors XY est reconnaissable.*

Cette proposition est une conséquence immédiate du théorème de Kleene. Nous en donnons ici une autre preuve, plus constructive.

Preuve. Soit X un langage reconnaissable. Soit $\mathcal{A} = (Q, I, T, \mathcal{F})$ un automate fini reconnaissant X, et soit $\mathcal{B} = (Q, I, T)$ l'automate asynchrone ayant pour flèches

$$\mathcal{F} \cup (T \times \{\varepsilon\} \times I)$$

Montrons que l'on a $X^+ = L(\mathcal{B})$. Il est clair en effet que $X^+ \subset L(\mathcal{B})$. Réciproquement, soit $c : i \xrightarrow{w} t$ un calcul réussi dans $\mathcal{B}$. Ce calcul peut se décomposer en

$$c : i_1 \xrightarrow{w_1} t_1 \xrightarrow{\varepsilon} i_2 \xrightarrow{w_2} t_2 \xrightarrow{\varepsilon} \cdots i_n \xrightarrow{w_n} t_n$$

avec $i = i_1$, $t = t_n$, et où aucun des calculs $c_k : i_k \xrightarrow{w_k} t_k$ ne contient de flèche étiquetée par le mot vide. Alors $w_1, w_2, \ldots, w_n \in X$, et donc $w \in X^+$. Il en résulte évidemment que $X^* = X^+ \cup 1$ est reconnaissable.

Considérons maintenant deux automates finis $\mathcal{A} = (Q, I, T, \mathcal{F})$ et $\mathcal{B} = (P, J, R, \mathcal{G})$ reconnaissant respectivement les langages X et Y. On peut supposer les ensembles d'états Q et P disjoints. Soit alors $\mathcal{C} = (Q \cup P, I, R)$ l'automate dont les flèches sont

$$\mathcal{F} \cup \mathcal{G} \cup (T \times \{\varepsilon\} \times J)$$

Alors on vérifie facilement que $L(\mathcal{C}) = XY$. ∎

2.5.1 Morphismes et substitutions

Soient A et B deux alphabets. Une application $f : A^* \to B^*$ est un *morphisme* si $f(xy) = f(x)f(y)$ pour tous mots $x, y \in A^*$. En particulier, on a $f(\varepsilon) = \varepsilon$, car $f(\varepsilon) = f(\varepsilon\varepsilon) = f(\varepsilon)f(\varepsilon)$, et ε est le seul mot égal à son carré. Il résulte de la définition que si $w = a_1 \cdots a_n$ où $a_1, \ldots, a_n \in A$, alors $f(w) = f(a_1) \cdots f(a_n)$. Ainsi, un morphisme est entièrement déterminé par sa donnée sur l'alphabet.

Exemple 5.4. Soient $A = \{a, b, c\}$ et $B = \{0, 1\}$, et soit f donnée par

$$f : \quad \begin{aligned} a &\mapsto 0 \\ b &\mapsto 01 \\ c &\mapsto 10 \end{aligned}$$

On a par exemple $f(abcba) = 00110010$. Notons que $f(ac) = f(ba) = 010$, et donc que f n'est pas injectif.

Exemple 5.5. Soient $A = \{0, 1, \ldots, F\}$ l'alphabet hexadécimal et $B = \{0, 1\}$. Le morphisme

$$\begin{aligned} 0 &\mapsto 0000 \\ 1 &\mapsto 0001 \\ &\cdots \\ 9 &\mapsto 1001 \\ A &\mapsto 1010 \\ &\cdots \\ F &\mapsto 1111 \end{aligned}$$

qui remplace chaque symbole hexadécimal par son écriture binaire est injectif.

Un morphisme $f : A^* \to B^*$ est *non effaçant* si l'image d'une lettre n'est pas le mot vide. Il est *littéral* si l'image d'une lettre est une lettre. Un morphisme littéral "préserve les longueurs", c'est-à-dire que la longueur de l'image d'un mot est égale à la longueur du mot.

Soit $f : A^* \to B^*$. Pour toute partie K de A^*, on note

$$f(K) = \bigcup_{w \in K} f(w)$$

Par exemple, dans l'exemple 5.4, on a $f(a^*b) = 0^*01$. Si f est un morphisme, on a les formules suivantes, pour $K, K' \subset A^*$:

$$\begin{aligned} f(K \cup K') &= f(K) \cup f(K') \\ f(KK') &= f(K)f(K') \\ f(K^*) &= (f(K))^* \end{aligned}$$

Une substitution est une généralisation des morphismes. Une *substitution* σ de A^* dans B^* est une application de A^* dans l'ensemble $\wp(B^*)$ des parties de B^* vérifiant, pour tous $x, y \in A^*$

$$\sigma(xy) = \sigma(x)\sigma(y)$$

et, de plus

$$\sigma(\varepsilon) = \{\varepsilon\}\,.$$

Bien noter que, dans la première formule, le produit dans le membre droit est le produit d'ensembles. A nouveau, $\sigma(a_1 \cdots a_n) = \sigma(a_1) \cdots \sigma(a_n)$, et une substitution est entièrement déterminée par la donnée sur l'alphabet.

Une substitution est dite *régulière* si les langages $\sigma(a)$, pour $a \in A$, sont réguliers. Un morphisme est une substitution particulière, où l'image de chaque lettre est réduite à un mot. Un morphisme est une substitution régulière.

Exemple 5.6. Considérons le langage a^*b sur $A = \{a, b\}$, et définissons une substitution σ de A^* dans B^*, avec $B = \{0, 1\}$, par

$$\sigma : \begin{array}{l} a \mapsto 10^* \\ b \mapsto 1 \end{array}$$

C'est une substitution régulière. On obtient $\sigma(b) = \{1\}$, $\sigma(ab) = 10^*1$, $\sigma(a^2b) = 10^*10^*1$ et finalement $\sigma(a^*b) = 1B^*1 \cup \{1\}$. ◁

Exemple 5.7. Considérons l'alphabet $A = \{a, b\}$, et deux substitutions σ et τ de A^* dans lui-même définies par

$$\sigma : \begin{array}{l} a \mapsto a^+ \\ b \mapsto b \end{array} \qquad \tau : \begin{array}{l} a \mapsto \{a^n b^n \mid n \geq 1\} \\ b \mapsto b \end{array}$$

La substitution σ est régulière, la substitution τ ne l'est pas. On a

$$\sigma(\{a^n b^n \mid n \geq 1\}) = \{a^{n_1} b a^{n_2} b \cdots a^{n_k} b^{1+k} \mid k \geq 1, n_1, \ldots, n_k \geq 1\}$$

et

$$\tau(a^+ b^+) = \{a^{n_1} b^{n_1} a^{n_2} b^{n_2} \cdots a^{n_k} b^{n_k+m} \mid k \geq 1, n_1, \ldots, n_k \geq 1, m \geq 1\}$$

◁

Proposition 5.8. *L'image d'un langage régulier par une substitution régulière est un langage régulier.*

Preuve. Soit K un langage régulier, et e une expression régulière qui le dénote. Soit σ une substitution régulière, et associons à toute lettre a, une expression régulière e_a dénotant le langage $\sigma(a)$. Alors le langage $\sigma(K)$ est dénoté par l'expression régulière obtenue en remplaçant, dans e, toutes les occurrences de lettres par l'expression associée. ■

Exemple 5.9. Pour illustrer la construction de la proposition, considérons la substitution régulière

$$\sigma : \begin{array}{l} a \mapsto a^* \\ b \mapsto b \end{array}$$

voisine de celle de l'exemple précédent. Pour le langage K des mots de longueur paire par exemple, on a l'expression régulière $((a + b)(a + b))^*$. Le langage $\sigma(K)$ a pour expression régulière $((a^* + b)(a^* + b))^*$. ◁

COROLLAIRE 5.10. *L'image homomorphe d'un langage régulier est un langage régulier.* ∎

PROPOSITION 5.11. *Soit $f : A^* \to B^*$ un morphisme. Si L est un langage régulier sur B, alors $f^{-1}(L)$ est un langage régulier sur A.*

Preuve. Soit $\mathcal{B} = (Q, I, T)$ un automate fini sur B reconnaissant L. On construit un automate $\mathcal{A} = (Q, I, T)$ sur A par (p, a, q) est une flèche de $\mathcal{A}$ si et seulement s'il existe un chemin de p à q dans $\mathcal{B}$ d'étiquette $f(a)$. Soit $K = L(\mathcal{A})$. On a $K = f^{-1}(L)$.

Montrons d'abord que $K \subset f^{-1}(L)$. Soit $w \in K$. Si w est le mot vide, alors $I \cap T \neq \emptyset$ et $\varepsilon \in L$. Si $i \xrightarrow{w} y$ est un chemin réussi pour w dans $\mathcal{A}$, on remplace chaque flèche (p, a, q) de ce chemin par le chemin $p \xrightarrow{f(a)} q$ dans $\mathcal{B}$. Ceci donne un chemin réussi $i \xrightarrow{f(w)} t$ dans $\mathcal{B}$. Donc $f(w) \in L$ et $w \in f^{-1}(L)$.

Réciproquement soit $w \in f^{-1}(L)$. Il existe $x \in L$ tel que $f(w) = x$. Posons $w = a_1 \cdots a_n$, où $a_1, \ldots, a_n$ sont des lettres. On a $x = x_1 \cdots x_n$, avec $x_k = f(a_k)$ pour $k = 1, \ldots, n$. Un chemin réussi $i \xrightarrow{x} t$ dans $\mathcal{B}$ se factorise en

$$i \xrightarrow{x_1} p_1 \xrightarrow{x_2} p_2 \cdots p_{n-1} \xrightarrow{x_n} t$$

Pour chacun des chemins $i \xrightarrow{x_1} p_1$, $p_1 \xrightarrow{x_2} p_2$, …, $p_{n-1} \xrightarrow{x_n} t$ de $\mathcal{B}$, il existe par construction une flèche (i, a_1, p_1), (p_1, a_2, p_2), …, (p_{n-1}, a_n, t) dans l'automate $\mathcal{A}$. Il existe donc un chemin réussi $i \xrightarrow{w} t$ dans $\mathcal{A}$, ce qui montre que $w \in K$, d'où l'inclusion réciproque et l'égalité. ∎

2.5.2 La puissance des $L_{p,q}$

De nombreuses propriétés de fermeture des langages réguliers viennent d'une étude des langages reconnus par un automate fini lorsque l'on change d'état initial ou terminal.

Commençons par introduire les notations : Soit $\mathcal{A} = (Q, I, T)$ un automate fini sur A. Pour tout couple d'états (p, q) on pose $L_{p,q}(\mathcal{A})$, ou $L_{p,q}$ lorsque le contexte le permet, le langage

$$L_{p,q} = \{w \in A^* \mid \exists c : p \to q, w = |c|\}$$

Ainsi, $L_{p,q}$ est l'ensemble des mots reconnus par l'automate en prenant p comme unique état initial et q comme unique état terminal. Bien entendu, on a

$$L(\mathcal{A}) = \bigcup_{i \in I, t \in T} L_{i,t}$$

Une formule utile est

$$L_{p,q} = \bigcup_{r \in Q} L_{p,r} L_{r,q}$$

En effet, si $x \in L_{p,r}$ et $y \in L_{r,q}$, alors $xy \in L_{p,q}$, d'où l'inclusion du membre droit dans le membre gauche. Réciproquement, on a $\varepsilon \in L_{p,p}$ pour tout p, donc $L_{p,q} \subset L_{p,p} L_{p,q}$.

PROPOSITION 5.12. *L'ensemble des préfixes, l'ensemble des suffixes, l'ensemble des facteurs d'un langage régulier sont des langages réguliers.*

Preuve. Avec les notations ci-dessus, et en supposant l'automate émondé, l'ensemble des préfixes est

$$\bigcup_{i\in I, t\in Q} L_{i,t}$$

l'ensemble des suffixes est

$$\bigcup_{i\in Q, t\in T} L_{i,t}$$

et l'ensemble des facteurs est

$$\bigcup_{i,t\in Q} L_{i,t}$$ ∎

PROPOSITION 5.13. *Le langage obtenu en supprimant les premières lettres des mots d'un langage régulier est un langage régulier.*

Preuve. Considérons un langage $L_{i,t}$. On a

$$L_{i,t} \setminus \varepsilon = \bigcup_{(i,a,p)\in\mathcal{F}} aL_{p,t}$$

et le langage recherché est donc

$$\bigcup_{(i,a,p)\in\mathcal{F}, t\in T} L_{p,t}$$ ∎

De nombreuses autres propriétés peuvent se prouver de cette manière. En voici une. La *fermeture circulaire* d'un langage K est l'ensemble $\hat{K} = \{uv \in A^* \mid vu \in K\}$. Par exemple, si $K = a^+b^+$, toutes les permutations circulaires que l'on peut faire mettront une plage de b devant la plage de a ou une plage de a derrière la plage de b. On a donc $\hat{K} = a^+b^+a^* \cup b^+a^+b^*$.

PROPOSITION 5.14. *La fermeture circulaire d'un langage régulier est un langage régulier.*

Preuve. On a

$$\hat{K} = \bigcup_{p\in Q, i\in I, t\in T} L_{p,t}L_{i,p}$$

En effet, soit $x \in \hat{K}$. Alors $x = uv$ pour un couple (u, v) tel que $vu \in K$. Il existe donc un état p tel que $v \in L_{i,p}$ et $u \in L_{p,t}$ pour des états $i \in I$, $t \in T$. Ceci prouve l'inclusion dans un sens. La réciproque se montre de la même façon. ∎

2.6 Automate minimal

Déterminer un automate ayant un nombre minimum d'états pour un langage rationnel donné est très intéressant du point de vue pratique. Il est tout à fait remarquable que tout langage rationnel possède un automate déterministe minimal unique, à une numérotation des états près. Ce résultat n'est plus vrai si l'on considère des automates qui ne sont pas nécessairement déterministes.

Il y a deux façons de définir l'automate déterministe minimal reconnaissant un langage rationnel donné. La première est intrinsèque ; elle est définie à partir du langage par une opération appelée le *quotient*. La deuxième est plus opératoire ; on part d'un automate déterministe donné, et on le réduit en identifiant des états appelés *inséparables*. Les algorithmes de minimisation utilisent la deuxième définition.

2.6.1 Quotients

Soit A un alphabet. Pour deux mots u et v, on pose

$$u^{-1}v = \{w \in A^* \mid uw = v\}, \qquad uv^{-1} = \{w \in A^* \mid u = wv\}$$

Un tel ensemble, appelé *quotient gauche* respectivement *droit* est, bien entendu, soit vide soit réduit à un seul mot qui, dans le premier cas est un suffixe de v, et dans le deuxième cas un préfixe de u.

La notation est étendue aux parties en posant, pour $X, Y \subset A^*$

$$X^{-1}Y = \bigcup_{x\in X}\bigcup_{y\in Y} x^{-1}y, \qquad XY^{-1} = \bigcup_{x\in X}\bigcup_{y\in Y} xy^{-1}$$

Les quotients sont un outil important pour l'étude des automates finis et des langages rationnels. On utilise principalement les quotients gauches par un mot, c'est-à-dire les ensembles

$$u^{-1}X = \{w \in A^* \mid uw \in X\}$$

En particulier $\varepsilon^{-1}X = X$ pour tout ensemble X, et $(uv)^{-1}X = v^{-1}(u^{-1})X$. Pour toute partie X de A^*, on pose

$$Q(X) = \{u^{-1}X \mid u \in A^*\} \tag{6.1}$$

Exemple 6.1. Sur $A = \{a, b\}$, soit X l'ensemble des mots qui contiennent au moins une fois la lettre a. Alors on a :

$$\varepsilon^{-1}X = X, \quad a^{-1}X = A^*, \quad b^{-1}X = X, \quad a^{-1}A^* = b^{-1}A^* = A^*$$

donc $Q(X) = \{X, A^*\}$.

Proposition 6.2. *Soit $X \subset A^*$ le langage reconnu par un automate fini déterministe, accessible et complet $\mathcal{A} = (Q, i, T)$; pour $q \in Q$, soit $L_q(\mathcal{A}) = \{w \in A^* \mid q \cdot w \in T\}$. Alors*

$$\{L_q(\mathcal{A}) \mid q \in Q\} = Q(X) \tag{6.2}$$

Preuve. Montrons d'abord que $Q(X)$ est contenu dans $\{L_q(\mathcal{A}) \mid q \in Q\}$. Soit $u \in A^*$, et soit $q = i \cdot u$ (cet état existe parce que $\mathcal{A}$ est complet). Alors $u^{-1}X = L_q(\mathcal{A})$, puisque

$$w \in u^{-1}X \iff uw \in X \iff i \cdot uw \in T \iff q \cdot w \in T \iff w \in L_q(\mathcal{A})$$

Pour montrer l'inclusion réciproque, soit $q \in Q$ et soit $u \in A^*$ tel que $q = i \cdot u$ (un tel mot existe parce que l'automate est accessible). Alors $L_q(\mathcal{A}) = u^{-1}X$. Ceci prouve la proposition. ■

EXEMPLE 6.3. Considérons l'automate $\mathcal{A}$ donné dans la figure 6.1. Un calcul rapide montre que

$$L_1(\mathcal{A}) = L_3(\mathcal{A}) = L_5(\mathcal{A}) = L_6(\mathcal{A}) = \{a,b\}^*$$
$$L_0(\mathcal{A}) = L_2(\mathcal{A}) = L_4(\mathcal{A}) = b^*a\{a,b\}^*$$

L'équation (6.2) est bien vérifiée.

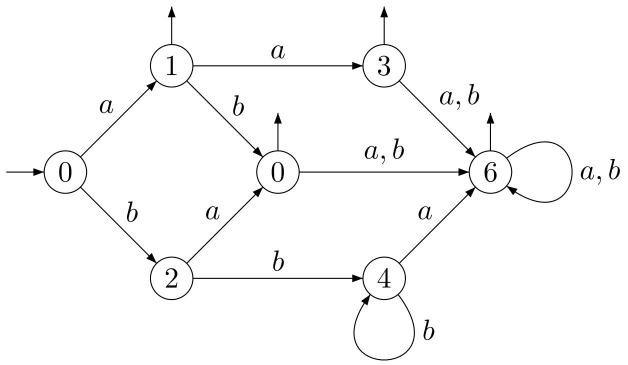

FIG. 6.1 – Un automate reconnaissant $X = b^*a\{a,b\}^*$.

◁

On déduit de cette proposition que, pour un langage reconnaissable X, l'ensemble des quotients gauches $Q(X)$ est fini ; nous allons voir dans un instant que la réciproque est vraie également. L'équation (6.2) montre par ailleurs que tout automate déterministe, accessible et complet reconnaissant X possède au moins $|Q(X)|$ états. Nous allons voir que ce minimum est atteint, et même, au paragraphe suivant, qu'il existe un automate unique, à un isomorphisme près, qui a $|Q(X)|$ états et qui reconnaît X.

Soit $X \subset A^*$. On appelle *automate minimal* de X l'automate déterministe

$$\mathcal{A}(X) = (Q(X), X, T(X))$$

dont l'ensemble d'états est donné par (6.1), ayant X comme état initial, l'ensemble

$$T(X) = \{u^{-1}X \mid u \in X\} = \{u^{-1}X \mid 1 \in u^{-1}X\}$$

comme ensemble d'états terminaux, la fonction de transition étant définie, pour $Y \in Q$ et $a \in A$ par

$$Y \cdot a = a^{-1}Y$$

Notons que si $Y = u^{-1}X$, alors $Y \cdot a = a^{-1}(u^{-1}X) = (ua)^{-1}X$, donc la fonction de transition est bien définie, et l'automate est complet.

EXEMPLE 6.4. L'automate $\mathcal{A}(X)$ pour le langage $X = b^*a\{a,b\}^*$ a les deux états X et $\{a,b\}^*$, le premier est initial, le deuxième est final. L'automate est donné dans la figure 6.2. ◁

PROPOSITION 6.5. *Le langage reconnu par l'automate $\mathcal{A}(X)$ est X.*

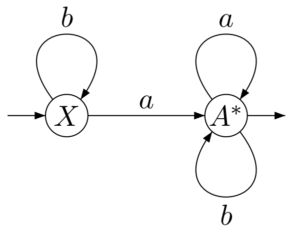

FIG. 6.2 – L'automate minimal pour $X = b^*a\{a,b\}^*$.

Preuve. Montrons d'abord que, pour tout $w \in A^*$ et pour tout $Y \in Q(X)$, on a $Y \cdot w = w^{-1}Y$. En effet, ceci est vrai si w est une lettre ou le mot vide. Si $w = ua$, avec a une lettre, alors $Y \cdot ua = (Y \cdot u) \cdot a = (u^{-1}Y) \cdot a = a^{-1}(u^{-1}Y) = (ua)^{-1}Y$. Ceci prouve la formule. Il en résulte que

$$w \in L(\mathcal{A}(X)) \iff X \cdot w \in T \iff w^{-1}X \in T \iff w \in X$$

Donc $\mathcal{A}(X)$ reconnaît X. ∎

COROLLAIRE 6.6. *Une partie $X \subset A^*$ est reconnaissable si et seulement si l'ensemble $Q(X)$ est fini.*

Preuve. Si X est reconnaissable, alors (6.2) montre que $Q(X)$ est fini. La réciproque découle de la proposition précédente. ∎

On peut calculer l'ensemble $Q(X)$ pour un langage rationnel, à partir d'une expression rationnelle pour X, à l'aide des formules de la proposition suivante. Cela ne résout pas complètement le problème du calcul de l'automate minimal, parce qu'une difficulté majeure demeure, à savoir tester si deux expressions sont équivalentes. On en parlera plus loin.

PROPOSITION 6.7. *Soit a une lettre, et soient X et Y des langages. Alors on a*

$$\begin{aligned}
a^{-1}\emptyset &= a^{-1}1 = \emptyset \\
a^{-1}a &= 1 \\
a^{-1}b &= \emptyset \quad (b \neq a) \\
a^{-1}(X \cup Y) &= a^{-1}X \cup a^{-1}Y \\
a^{-1}(XY) &= (a^{-1}X)Y \cup (X \cap 1)a^{-1}Y \qquad (6.3) \\
a^{-1}X^* &= (a^{-1}X)X^* \qquad (6.4)
\end{aligned}$$

Preuve. Les quatre premières formules sont évidentes. Prouvons la formule 6.3. Si $w \in a^{-1}(XY)$, alors $aw \in XY$; il existe donc $x \in X$, $y \in Y$ tels que $aw = xy$. Si $x = 1$, alors $aw = y$, donc $w \in (X \cap 1)a^{-1}Y$; sinon, $x = au$ pour un préfixe u de w, et $u \in a^{-1}X$, d'où $w \in (a^{-1}X)Y$. L'inclusion réciproque se montre de la même manière.

Considérons la dernière formule. Soit $w \in a^{-1}X^*$. Alors $aw \in X^*$, donc $aw = xx'$, avec $x \in X$, $x \neq 1$, et $x' \in X^*$. Mais alors $x = au$, avec $u \in a^{-1}X$, donc $w \in (a^{-1}X)X^*$. Réciproquement, on a par la formule (6.3), l'inclusion $(a^{-1}X)X^* \subset a^{-1}(XX^*)$, donc $(a^{-1}X)X^* \subset a^{-1}X^*$. ∎

EXEMPLE 6.8. Calculons, à l'aide de ces formules, le langage $a^{-1}(b^*aA^*)$. On a

$$a^{-1}(b^*aA^*) = a^{-1}(b^*(aA^*)) = (a^{-1}b^*)aA^* \cup a^{-1}(aA^*)$$

par (6.3) ; or le premier terme de l'union est vide par (6.3), d'où

$$a^{-1}(b^*aA^*) = (a^{-1}a)A^* \cup (a \cap 1)a^{-1}A^* = (a^{-1}a)A^* = A^*$$

2.6.2 Equivalence de Nerode

Dans ce paragraphe, tous les automates considérés sont déterministes, accessibles et complets.

Soit $\mathcal{A} = (Q, i, T)$ un automate sur un alphabet A reconnaissant un langage X. Pour tout état q, on pose

$$L_q(\mathcal{A}) = \{w \in A^* \mid q \cdot w \in T\}$$

C'est donc l'ensemble des mots reconnus par l'automate $\mathcal{A}$ en prenant q comme état initial. Bien entendu, le langage X reconnu par $\mathcal{A}$ coïncide avec $L_i(\mathcal{A})$. On a vu, au paragraphe précédent, que

$$\{L_q(\mathcal{A}) \mid q \in Q\} = Q(X)$$

La correspondance s'établit par

$$L_q(\mathcal{A}) = u^{-1}X \quad \text{si } i \cdot u = q$$

Notons que, plus généralement,

$$L_{q \cdot v}(\mathcal{A}) = v^{-1}L_q(\mathcal{A}) \tag{6.5}$$

En effet, $w \in L_{q\cdot v}(\mathcal{A})$ si et seulement si $(q \cdot v) \cdot w \in T$ donc si et seulement si $vw \in L_q(\mathcal{A})$. Lorsque l'automate est fixé, on écrira L_q au lieu de $L_q(\mathcal{A})$. Deux états $p, q \in Q$ sont dits *inséparables* si $L_p = L_q$, ils sont *séparables* sinon. Ainsi, p et q sont séparables si et seulement s'il existe un mot w tel que $p \cdot w \in T$ et $q \cdot w \notin T$ ou vice-versa. Si w est un mot ayant cette propriété, on dit qu'il *sépare* les états p et q. L'*équivalence de Nerode* sur Q (ou sur $\mathcal{A}$) est la relation $\sim$ définie par

$$p \sim q \iff p \text{ et } q \text{ sont inséparables}$$

PROPOSITION 6.9. *Dans l'automate minimal, deux états distincts sont séparables, et l'équivalence de Nerode est l'égalité.*

Preuve. Soit $Y = u^{-1}X$ un état de l'automate minimal $\mathcal{A}(X)$ du langage X. Comme $Y = X \cdot u$, on a $L_Y = u^{-1}X = Y$. Par conséquent, deux états distincts ne sont pas équivalents. ∎

EXEMPLE 6.10. Reprenons l'automate donné dans la figure 6.3 ci-dessous, et que nous avons déjà considéré dans le paragraphe précédent. Puisque

$$L_1 = L_3 = L_5 = L_6 = \{a, b\}^*$$
$$L_0 = L_2 = L_4 = b^*a\{a, b\}^*$$

l'équivalence de Nerode a donc les deux classes $\{0, 2, 4\}$ et $\{1, 3, 5, 6\}$. Le mot vide sépare deux états pris dans des classes distinctes. ◁

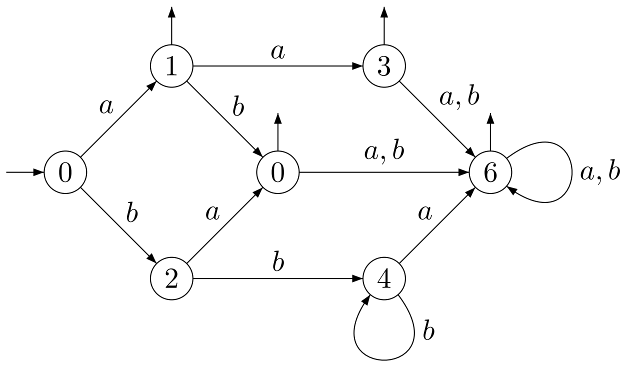

FIG. 6.3 – Un automate qui n'est pas minimal.

PROPOSITION 6.11. *L'équivalence de Nerode est une relation d'équivalence régulière à droite, c'est-à-dire vérifiant*

$$p \sim q \Rightarrow p \cdot u \sim q \cdot u \qquad (u \in A^*)$$

De plus, une classe de l'équivalence ou bien ne contient pas d'états terminaux, ou bien ne contient que des états terminaux.

Preuve. Soit $\mathcal{A} = (Q, i, T)$ un automate. Il est clair que la relation $\sim$ est une relation d'équivalence. Pour montrer sa régularité, supposons $p \sim q$, et soit $u \in A^*$. En vue de (6.5), on a $L_{q \cdot u} = u^{-1}L_q = u^{-1}L_p = L_{p \cdot u}$, donc $p \cdot u \sim q \cdot u$. Supposons enfin $p \sim q$ et p terminal. Alors $1 \in L_p$, et comme $L_p = L_q$, on a $1 \in L_q$, donc q est terminal. ∎

L'équivalence de Nerode sur un automate $\mathcal{A} = (Q, i, T)$ étant régulière à droite, on peut définir un *automate quotient* en confondant les états d'une même classe. Plus précisément, notons $[q]$ la classe d'un état q dans l'équivalence. L'automate quotient est alors défini par

$$\mathcal{A}/\!\sim = (Q/\!\sim, [i], \{[t] : t \in T\})$$

avec la fonction de transition

$$[q \cdot a] = [q] \cdot a \tag{6.6}$$

qui justement est bien définie (indépendante du représentant choisi dans la classe de q) parce que l'équivalence est régulière à droite.

Dans l'exemple ci-dessus, l'automate quotient a deux états : l'état $\{0, 2, 4\}$ est initial, et $\{1, 3, 5, 6\}$ est l'état final. La formule (6.6) permet de calculer les transitions. On obtient l'automate de la figure 6.4.

Comme le suggère cet exemple, l'automate quotient est en fait l'automate minimal, à une renumérotation des états près. Nous avons besoin, pour le prouver, de la notion d'isomorphisme d'automates. Soient $\mathcal{A} = (Q, i, T)$ et $\mathcal{A}' = (Q', i', T')$ deux automates sur A. Ils sont dits *isomorphes* s'il existe une bijection

$$\alpha : Q \to Q'$$

telle que $\alpha(i) = i'$, $\alpha(T) = T'$, et $\alpha(q \cdot a) = \alpha(q) \cdot a$ pour tout $q \in Q$ et $a \in A$.

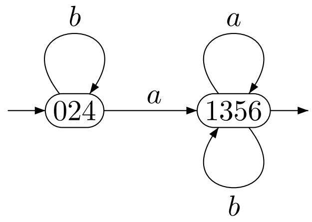

FIG. 6.4 – Un automate quotient.

PROPOSITION 6.12. *Soit $\mathcal{A} = (Q, i, T)$ un automate sur un alphabet A reconnaissant un langage X. Si l'équivalence de Nerode de $\mathcal{A}$ est l'égalité, alors $\mathcal{A}$ et l'automate minimal $\mathcal{A}(X)$ sont isomorphes.*

Preuve. Soit $\alpha : Q \to Q(X)$ définie par $\alpha(q) = L_q(\mathcal{A})$. Comme l'équivalence de Nerode est l'égalité, α est une bijection. Par ailleurs, $\alpha(i) = L_i = X$, et $t \in T$ si et seulement si $1 \in L_t(\mathcal{A})$, donc si et seulement si $L_t(\mathcal{A})$ est état final de $\mathcal{A}(X)$. Enfin, on a

$$L_{q \cdot a}(\mathcal{A}) = L_q(\mathcal{A}) \cdot a$$

montrant que α est bien un morphisme. ∎

De cette propriété, on déduit une conséquence importante :

THÉORÈME 6.13. *L'automate minimal d'un langage X est l'automate ayant le moins d'états parmi les automates déterministes complets qui reconnaissent X. Il est unique à un isomorphisme près.*

Preuve. Soit $\mathcal{B}$ un automate reconnaissant X et ayant un nombre minimal d'états. Alors son équivalence de Nerode est l'égalité, sinon on pourrait passer au quotient par son équivalence de Nerode et trouver un automate plus petit. Par la proposition précédente, $\mathcal{B}$ est isomorphe à $\mathcal{A}(X)$. ∎

L'unicité de l'automate minimal ne vaut que pour les automates déterministes. Il existe des automates non déterministes, non isomorphes, ayant un nombre minimal d'états, et reconnaissant un même langage (voir exercices).

2.7 Calcul de l'automate minimal

Le calcul de l'automate minimal peut se faire par un procédé appelé l'algorithme de Moore, et que nous exposons maintenant. Dans le premier paragraphe, on montre comment calculer l'équivalence de Nerode de façon itérative. Ce calcul est traduit en un algorithme explicite dans le deuxième paragraphe.

2.7.1 Calcul de Moore

Soit $\mathcal{A} = (Q, i, T)$ un automate déterministe, accessible et complet sur un alphabet A. Pour calculer l'automate minimal, il suffit de calculer l'équivalence de Nerode définie, rappelons-le, par

$$p \sim q \iff L_p = L_q$$

où $L_p = \{w \in A^* \mid p \cdot w \in T\}$. Pour calculer cette équivalence, on procède par approximations successives. On considère pour ce faire l'équivalence suivante, où k est un entier naturel :

$$p \sim_k q \iff L_p^{(k)} = L_q^{(k)}$$

avec

$$L_p^{(k)} = \{w \in L_p \mid |w| \leq k\}$$

L'équivalence de Nerode est l'intersection de ces équivalences. La proposition suivante exprime l'équivalence $\sim_k$ au moyen de l'équivalence $\sim_{k-1}$. Elle permettra de prouver que l'on obtient bien « à la limite » l'équivalence de Nerode, et elle donne aussi un procédé de calcul.

PROPOSITION 7.1. *Pour tout entier $k \geq 1$, on a*

$$p \sim_k q \iff p \sim_{k-1} q \quad \text{et} \quad (\forall\, a \in A, \quad p \cdot a \sim_{k-1} q \cdot a)$$

Preuve. On a

$$\begin{aligned} L_p^{(k)} =& \{w \mid p \cdot w \in T \;\text{ et }\; |w| \leq k\} \\ =& \{w \mid p \cdot w \in T \;\text{ et }\; |w| \leq k-1\} \\ & \qquad \cup \bigcup_{a \in A} a\{v \mid (p \cdot a) \cdot v \in T \;\text{ et }\; |v| \leq k-1\} \\ =& L_p^{(k-1)} \cup \bigcup_{a \in A} a L_{p \cdot a}^{(k-1)} \end{aligned}$$

L'observation n'est qu'une traduction de ces égalités. ∎

COROLLAIRE 7.2. *Si les équivalences $\sim_k$ et $\sim_{k+1}$ coïncident, les équivalences $\sim_{k+l}$ ($l \geq 0$) sont toutes égales, et égales à l'équivalence de Nerode.*

Preuve. De la proposition, il résulte immédiatement que l'égalité des équivalences $\sim_k$ et $\sim_{k+1}$ entraîne celle des équivalences $\sim_{k+1}$ et $\sim_{k+2}$. D'où la première assertion. Comme $p \sim q$ si et seulement si $p \sim_k q$ pour tout $k \geq 0$, la deuxième assertion s'en déduit. ∎

PROPOSITION 7.3. *Si $\mathcal{A}$ est un automate à n états, l'équivalence de Nerode de $\mathcal{A}$ est égale à $\sim_{n-2}$.*

Preuve. Si, pour un entier $k > 0$, les équivalences $\sim_{k-1}$ et $\sim_k$ sont distinctes, le nombre de classes de l'équivalence $\sim_k$ est $\leq k+2$. ∎

2.7.2 Algorithme de Moore

Dans la pratique, on applique le procédé décrit ci-dessous qui consiste à essayer de briser des classes des équivalences $\sim_{k-1}$. Pour ce faire, on considère une classe P de $\sim_{k-1}$, et on calcule les classes auxquelles appartiennent les états $p \cdot a$ pour $p \in P$ et $a \in A$. D'après la Proposition 7.1, deux états p et p' de P sont dans deux classes différentes de $\sim_k$ dès que $p \cdot a \not\sim_{k-1} p' \cdot a$ pour une lettre a.

De manière effective, chaque classe d'une équivalence $\sim_{k-1}$ est numérotée. On débute avec l'équivalence $\sim_0$ dont les deux classes contiennent les états terminaux respectivement les états nonterminaux. Les images $p \cdot a$ sont calculées, et les classes correspondantes sont numérotées.

Lorsque les classes de $\sim_{k-1}$ ont été calculées, on calcule les états $p{\cdot}a$ et on note les classes des états d'arrivée. Ensuite, on inspecte, pour chaque état p, les suites de numéros de classes de p et des états $p \cdot a$, pour $a \in A$. Deux suites identiques indiquent deux états équivalents pour $\sim_k$. Le nombre de suites différentes est donc le nombre de classes de l'équivalence $\sim_k$. On numérote ces classes, et on termine lorsque le nombre de classes n'augmente plus.

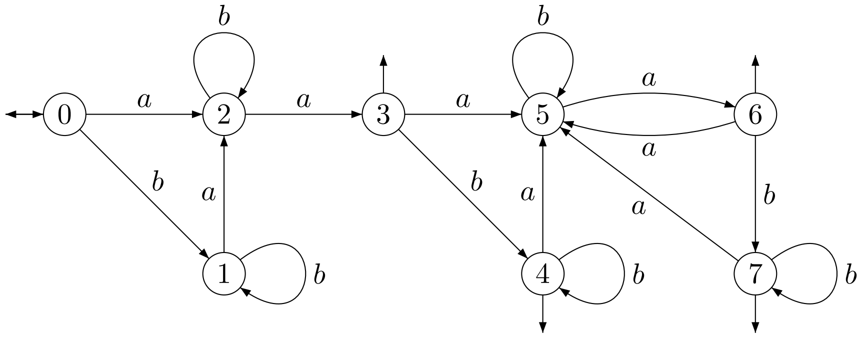

Fig. 7.1 – Un automate à minimiser.

Voici un exemple détaillé. On part de l'automate de la figure 7.1 dont la fonction de transition est la suivante :

	a	b
0	2	1
1	2	1
2	3	2
3	5	4
4	5	4
5	6	5
6	5	7
7	6	7

On crée un tableau dont les lignes sont les différents états de l'automate. Chaque colonne contient le numéro d'une classe à laquelle appartient un état.

La première colonne, indicée par $\sim_0$, contient, dans chaque ligne, le numéro de la classe à laquelle appartient l'état qui indice la ligne. Cette partition s'obtient en séparant les états terminaux des autres. On obtient donc deux classes notées I et II.

	$\sim_0$
0	I
1	II
2	II
3	I
4	I
5	II
6	I
7	I

Les colonnes suivantes sont indicées par les lettres de l'alphabet. Pour chaque ligne q et chaque colonne a, l'entrée contient la classe à laquelle appartient l'état $q \cdot a$. Dans notre cas, on obtient :

	$\sim_0$	a	b
0	I	II	II
1	II	II	II
2	II	I	II
3	I	II	I
4	I	II	I
5	II	I	II
6	I	II	I
7	I	II	I

Il y a 4 triplets de numéros de classes différents, à savoir (I, II, II), (II, II, II), (II, I, II) et (I, II, I). L'équivalence $\sim_1$ a donc les 4 classes 0/1/25/3467. On renumérote les classes, et on obtient le tableau suivant :

	$\sim_0$	a	b	$\sim_1$
0	I	II	II	I
1	II	II	II	II
2	II	I	II	III
3	I	II	I	IV
4	I	II	I	IV
5	II	I	II	III
6	I	II	I	IV
7	I	II	I	IV

Nous procédons maintenant de même avec la nouvelle partition : pour chaque état q et chaque colonne a, on note le numéro de la classe à laquelle appartient l'état $q \cdot a$. On obtient le tableau souvant :

	$\sim_0$	a	b	$\sim_1$	a	b
0	I	II	II	I	III	II
1	II	II	II	II	III	II
2	II	I	II	III	IV	III
3	I	II	I	IV	III	IV
4	I	II	I	IV	III	IV
5	II	I	II	III	IV	III
6	I	II	I	IV	III	IV
7	I	II	I	IV	III	IV

On constate que pour deux états d'une même classe de $\sim_1$, les couples de classes des états d'arrivée par les lettres a et b sont les mêmes : pour les états de la classe III, c'est toujours (IV, III), et pour les états de la classe IV, c'est toujours (III, IV). En d'autres termes, les équivalences $\sim_1$ et $\sim_2$ sont les mêmes. On le voit en numérotant comme précédemment les classes :

	$\sim_0$	a	b	$\sim_1$	a	b	$\sim_2$
0	I	II	II	I	III	II	I
1	II	II	II	II	III	II	II
2	II	I	II	III	IV	III	III
3	I	II	I	IV	III	IV	IV
4	I	II	I	IV	III	IV	IV
5	II	I	II	III	IV	III	III
6	I	II	I	IV	III	IV	IV
7	I	II	I	IV	III	IV	IV

Ainsi, l'équivalence de Nerode est égale à $\sim_2$. L'automate minimal s'obtient directement à partir du dernier tableau : voir la figure 7.2.

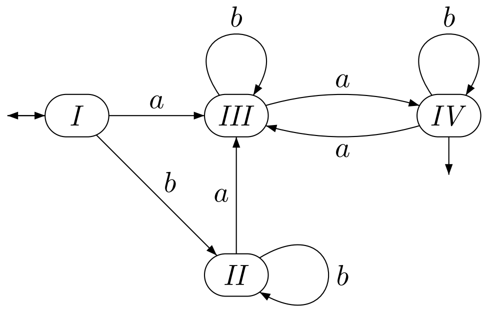

FIG. 7.2 – L'automate minimisé.

2.8 Exemples

Dans cette section, nous présentons quelques exemples "pratiques" d'usage des automates.

2.8.1 Editeurs

Les automates jouent un rôle omniprésent dans les interfaces graphiques sous la forme d'indicateurs ("flags" en anglais). Ces indicateurs constituent une aide pour l'utilisateur, en décrivant la situation où il se trouve.

Prenons l'exemple d'un éditeur de texte. Un document en cours de traitement peut être dans deux états : modifié M ou inchangé I. On passe de l'état I à l'état M par toute modification – insertion, suppression de caractère, collage, etc. On passe de l'état M

à l'état I essentiellement par la sauvegarde (certains éditeurs permettent de revenir à l'état I par annulation de commandes). On est donc en présence d'un automate très simple, donné dans la figure 8.1. En fait, la boucle autour de l'état I n'est pas toujours présente. Au contraire, les "bons" traitements de texte ne sauvegardent pas à nouveau le texte s'il n'a pas été modifié.

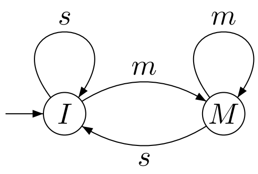

Fig. 8.1 – L'automate de sauvegarde d'un texte.

De manière plus générale, toute la batterie des "boutons" indicateurs, ou des entrées de menus à cocher, dans un éditeur ou un autre logiciel renvoie à des automates à deux états. Pour chacun, la nature des actions qui permettent de passer d'un état à l'autre doit être spécifiée avec soin.

Quand on veut décrire globalement l'état d'un éditeur, on doit donner la suite des valeurs de chacun des automates élémentaires. On fait donc le produit des automates. En voici un exemple.

Les traitements de texte avec *styles* permettent de mettre du texte en gras, en italique, en souligné, par action sur des boutons appropriés. On est alors en présence de trois petits automates, décrits dans la figure 8.2.

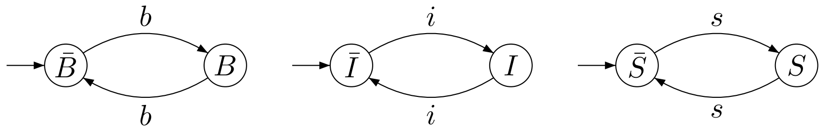

Fig. 8.2 – Les automates pour mettre en gras, en italique, en souligné.

Le produit de ces trois automates donne l'automate à 8 états de la figure 8.3.

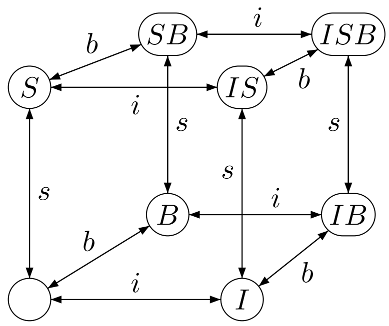

Fig. 8.3 – L'automate produit des automates précédents.

Dans cet automate, le nom d'un état est l'ensemble des options valides.

Dans les éditeurs de dessin – ou des logiciels graphiques – le nombre de paramètres est souvent grand (épaisseur du trait, nature du trait (plein ou pointillé), couleur du trait, couleur de fond, police de caractère, etc). L'ensemble des valeurs des paramètres est appelé l'*état graphique*. Pour les valeurs numériques, la présentation par automate fini n'est pas adaptée.

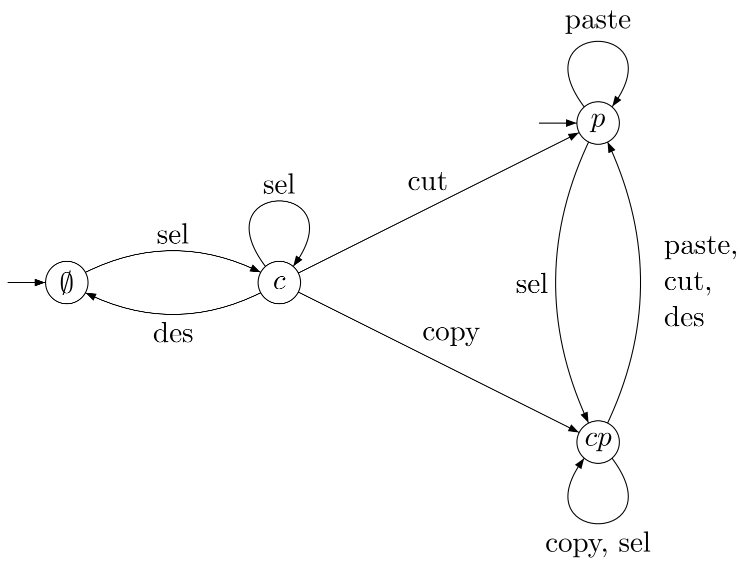

FIG. 8.4 – Automate du "couper-coller".

L'automate de la figure 8.4 décrit le comporterment du "couper-coller" dans un éditeur de texte. Cet automate décrit les possibilités offertes par les diverses entrées du menu d'édition d'un éditeur, en fonction des actions entreprises. Tout le monde sait que toute opération de copier-coller n'est pas toujours disponible. Un peu de réflexion conduit au constat que le "couper" et le "copier" sont toujours simultanément actifs ou inactifs. Par ailleurs, on ne peut couper que si l'on a sélectionné, et on ne peut coller que si on a procédé auparavant à une mémorisation, dans le presse-papier, notamment par un couper ou un coller.

L'automate a deux états initiaux. En effet, lorque l'on ouvre une fenêtre texte, soit rien n'a été copié, soit on a copié quelque chose d'une fenêtre ou d'un autre dispositif. Selon le cas, on est dans l'état $\emptyset$ ou dans l'état p.

2.8.2 Digicode

Le digicode est un dispositif bien connu. C'est en fait un cas particulier de la recherche d'un motif dans un texte.

L'alphabet d'un digicode contient les douze symboles $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B\}$. Le mécanisme déclenche l'ouverture lorsque l'acteur a entré une séquence correcte de 4 ou de 5 symboles. Supposons que le code est $12A73$. Tout mot sur l'alphabet S qui se termine par $12A73$ déclenche l'ouverture, et l'on peut même taper des symboles après l'ouverture sans que cela ne provoque la fermeture. Ainsi, tout mot contenant $12A73$ déclenche l'ouverture. En fait, on constate que la plupart des digicodes autorisent la répétition d'une même touche. Par exemple, si l'on tape $11222A773$, la porte s'ouvrira aussi. L'ensemble des séquences correctes est donc $S^*1^+2^+A^+7^+3^+S^*$, et l'automate – pour ce code particulier – est donné dans la figure 8.5. Chaque flèche vers l'état 0 a

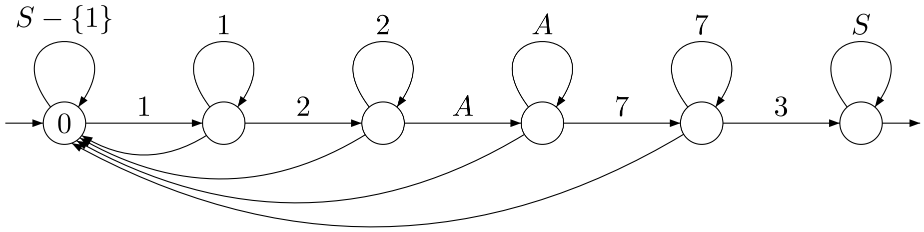

FIG. 8.5 – L'automate du digicode pour le code d'entrée $12A73$.

pour étiquettes les symboles restants.

Chapitre 3

Grammaires

Dans ce chapitre, nous présentons les grammaires algébriques ou context-free. Nous montrons comment transformer les grammaires, de donnons en particulier diverses formes normales. Nous présentons enfin les automates à pile.

3.1 Généralités

3.1.1 Introduction

Une des applications principales des langages algébriques est dans la définition des langages de programmation, et dans l'analyse de la syntaxe des programmes. Dans un compilateur, l'analyseur syntaxique reçoit le programme sous la forme d'une suite de lexèmes, produits par l'analyseur lexical. Le rôle de l'analyseur syntaxique est de déduire, de cette séquence, la structure syntaxique d'un programme, c'est-à-dire de regrouper des suites de lexèmes en unités syntaxiques de plus en plus grandes.

Dans les langages de programmation impératifs, les unités syntaxiques sont les variables, expressions, instructions, suites d'instructions, blocs, déclarations, définitions.

L'analyseur syntaxique reconnaît la structure syntaxique d'un programme, et la représente de manière appropriée à un traitement ultérieur, par d'autres parties du compilateur. Une présentation possible est un arbre syntaxique, qui peut ensuite être décoré ou transformé et finalement traduit dans un langage assembleur.

La structure syntaxique des programmes d'un langage de programmation peut être décrite, en partie, à l'aide d'une grammaire algébrique. A partir d'une telle grammaire, on engendre un analyseur syntaxique qui est un automate à pile. Pour des raisons d'efficacité, on se restreint dans la pratique aux grammaires analysables de manière déterministe, et même à des grammaires encore plus particulières.

3.1.2 Définition

Une *grammaire algébrique* ou *context-free* $G = (V, P, S)$ sur un alphabet A est composée

– d'un alphabet fini V, disjoint de A, appelé alphabet des *variables* ou *non-terminaux* ;
– d'un élément distingué S de V appelé l'axiome ;
– d'un ensemble fini $P \subset V \times (V \cup A)^*$ de *règles* ou *productions*.

Les symboles non-terminaux représentent (d'une manière qui sera rendue précise plus loin) des ensembles de mots, à savoir les mots qu'ils produisent, ou que l'on peut dériver à partir d'eux. Dans les langages de programmation, les non-terminaux représentent les unités syntaxiques, et les lettres de l'alphabet A (alphabet *terminal)* les lexèmes du langage.

Soit règle $p \in P$ est un couple $p = (X, \alpha)$, avec $X \in V$ et $\alpha \in (V \cup A)^*$. On écrit souvent

$$X \to \alpha$$

au lieu de (X, α), et on appelle X le *membre gauche*, et α le membre droit de p. On dit aussi que c'est une règle *de* X ou une X-règle.

Les exemples de grammaires ont souvent de nombreuses règles. Il est pour cela utile d'écrire

$$X \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

pour

$$X \to \alpha_1, X \to \alpha_2, \cdots, X \to \alpha_n \ .$$

On dit que ces règles sont les X-règles ou les règles *de* la variable X.

Exemple 1.1. Dans le langage de programmation Pascal, les *instructions* sont décrites par les règles suivantes :

$$\begin{array}{lcl}
\textit{instruction} & \to & \varepsilon \mid \textit{variable}\ \texttt{:=}\textit{expression} \\
& & \mid \texttt{begin}\ \textit{liste-instructions}\ \texttt{end} \\
& & \mid \texttt{if}\ \textit{expression}\ \texttt{then}\ \textit{instruction} \\
& & \mid \texttt{if}\ \textit{expression}\ \texttt{then}\ \textit{instruction}\ \texttt{else}\ \textit{instruction} \\
& & \mid \texttt{case}\ \textit{expression}\ \texttt{of}\ \textit{liste-case}\ \texttt{end} \\
& & \mid \texttt{while}\ \textit{expression}\ \texttt{do}\ \textit{instruction} \\
& & \mid \texttt{repeat}\ \textit{instruction}\ \texttt{until}\ \textit{expression} \\
& & \mid \texttt{for}\ \textit{varid}\ \texttt{:=}\textit{liste-pour}\ \texttt{do}\ \textit{instruction} \\
& & \mid \textit{identificateur-procédure} \\
& & \mid \textit{identificateur-procédure}\texttt{(}\textit{liste-expressions}\texttt{)} \\
& & \mid \texttt{goto}\ \textit{étiquette} \\
& & \mid \texttt{with}\ \textit{liste-variables-record}\ \texttt{do}\ \textit{instruction} \\
& & \mid \textit{étiquette}\ \texttt{:}\textit{instruction}
\end{array}$$

Les symboles `begin`, `end`, `if`, `then` etc constituent les lettres terminales, et les règles font appel à des variables définies par ailleurs, comme *expression*, *liste-case* etc. La grammaire Pascal contient plus de 300 règles.

Exemple 1.2. Une forme simplifiée des *expressions arithmétiques* se décrit par la grammaire que voici :

$$\begin{aligned}
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid a \mid b \mid c
\end{aligned}$$

Ici, l'alphabet des variables est $\{E, T, F\}$, l'alphabet terminal est $\{+, *, (,), a, b, c\}$ et l'axiome est E. $\triangleleft$

Formellement, les mots engendrés par une grammaire se définissent comme suit. Soit $G = (V, P, S)$ une grammaire sur A. Soient $u, v \in (V \cup A)^*$. On dit que u *dérive directement* en v, et on écrit

$$u \to v$$

s'il existe des factorisations $u = xXy$ et $v = x\alpha y$, avec $(X, \alpha) \in P$. Soit $k \geq 0$ un entier. On dit que u *dérive à l'ordre* k en v, et on écrit

$$u \xrightarrow{k} v$$

s'il existe des mots $w_0, w_1, \ldots, w_k \in (V \cup A)^*$ tels que

$$u = w_0, \quad w_{i-1} \to w_i \ (1 \leq i \leq k), \quad w_k = v\ .$$

La suite $w_0, \ldots, w_k$ est alors appelée une *dérivation* de u en v. On dit que u *dérive* en v et on écrit

$$u \xrightarrow{*} v$$

lorsqu'il existe un entier $k \geq 0$ tel que u dérive à l'ordre k en v. On dit que u *dérive proprement* en v et on écrit

$$u \xrightarrow{+} v$$

lorsqu'il existe un entier $k > 0$ tel que u dérive à l'ordre k en v. Les relations $\xrightarrow{+}$ et $\xrightarrow{*}$ sont évidemment la fermeture transitive (resp. transitive et réflexive) de la relation $\longrightarrow$.

Le *langage engendré* par la grammaire G est l'ensemble

$$L(G) = \{w \in A^* \mid S \xrightarrow{*} w\}$$

Il est commode de pouvoir décrire les mots qui dérivent d'autres variables que l'axiome. On pose

$$L_G(X) = \{w \in A^* \mid X \xrightarrow{*} w\}$$

et plus généralement, pour $\alpha \in (V \cup A)^*$,

$$L_G(\alpha) = \{w \in A^* \mid \alpha \xrightarrow{*} w\}\ .$$

Bien évidemment, on a $L(G) = L_G(S)$, et $L_G(w) = \{w\}$, si $w \in A^*$.

Un mot $\alpha \in (V \cup A)^*$ tel que $S \xrightarrow{*} \alpha$ est un *mot élargi* ou une *forme* engendrée par S. Dans une dérivation, tous les mots à l'exception éventuelle du dernier sont des formes.

Exemple 1.3. (suite de l'exemple 1.2) On a

$$E \xrightarrow{*} a * b + c$$

car en effet

$$\begin{aligned} E &\to E + T \to T + T \to T * F + T \to T * b + T \to F * b + T \\ &\to F * b + F \to a * b + F \to a * b + c\,. \end{aligned}$$

3.1.3 Arbre de dérivation

Soit $G = (V, P, S)$ une grammaire. Soit D un arbre ordonné (les fils d'un sommet sont ordonnés). On étiquette les feuilles de D avec des éléments de $A \cup \{\varepsilon\}$ et les nœuds avec des lettres dans V. L'étiquette d'une feuille est le mot vide seulement si la feuille est fille unique. L'arbre D est un *arbre de dérivation* pour un mot w à partir de $X \in V$ si les conditions suivantes sont remplies :

(1) pour tout nœud, si Y est l'étiquette du nœud, et si $Z_1, \ldots, Z_n$ sont les nœuds de ses fils, dans cet ordre, alors $Y \to Z_1 \cdots Z_n$ est une règle ;

(2) le *mot des feuilles* de D, c'est-à-dire le mot obtenu en concaténant les étiquettes des feuilles de la gauche vers la droite, est le mot w ;

(3) l'étiquette de la racine est X.

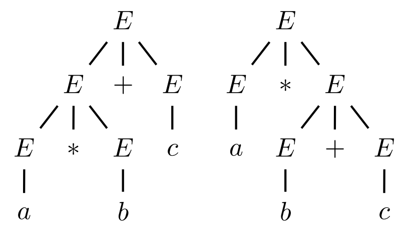

FIG. 1.1 – Deux arbres de dérivation pour l'expression $a * b + c$.

EXEMPLE 1.4. Considérons la grammaire

$$E \to E + E \mid E * E \mid (E) \mid a \mid b \mid c$$

Le mot $a * b + c$ possède deux arbres de dérivation, à savoir ceux de la figure 1.1. ◁

Une grammaire G est dite *ambiguë* pour un mot x s'il existe plus d'un arbre de dérivation pour le mot x. Elle est ambiguë si elle est ambiguë pour au moins un mot. Une grammaire qui n'est pas ambiguë est dite *inambiguë* ou non-ambiguë.

3.1.4 Dérivations gauche et droite

Soit $w_0, \ldots, w_k$ une dérivation de u en v ($u = w_0$, $v = w_k$). La dérivation est *gauche* si, à chaque étape de la dérivation, c'est la variable la plus à gauche qui est dérivée : si $w_i = x_i X_i y_i$ et $w_{i+1} = x_i \alpha_i y_i$, alors $x_i \in A^*$, donc x_i ne contient pas de variable. De même, une dérivation *droite* est une dérivation où, à chaque pas, c'est la variable la plus à droite qui est dérivée.

PROPOSITION 1.5. *Il y a bijection entre les dérivations gauches d'un mot x à partir de S et les arbres de dérivation pour x à partir de S.*

Preuve. Considérons un arbre de dérivation D pour x à partir de S. On obtient une dérivation gauche en appliquant les règles dans un parcours en profondeur gauche de l'arbre. Réciproquement, étant donné une dérivation gauche, on construit facilement un arbre dont la dérivation est la dérivation gauche associée. ∎

COROLLAIRE 1.6. *Pour tout mot $x \in L(G)$, il y a autant de dérivations gauches que de dérivations droites.* ∎

On écrit

$$S \xrightarrow[g]{*} x, \qquad S \xrightarrow[d]{*} x$$

pour dire que x dérive à gauche resp. à droite de x. Si x est terminal ($x \in A^*$), alors l'existence d'une dérivation implique l'existence d'une dérivation gauche et d'une dérivation droite. Si x est une forme, il n'en est plus ainsi.

EXEMPLE 1.7. Revenons sur la grammaire de l'exemple 1.2, et sur la dérivation

$$\begin{aligned} E &\to E+T \to T+T \to T*F+T \to T*b+T \to F*b+T \\ &\to F*b+F \to a*b+F \to a*b+c\,. \end{aligned}$$

Ce n'est pas une dérivation gauche (à cause de l'étape $T*F+T \to T*b+T$), ni une dérivation droite pour la même raison. L'arbre de dérivation correspondant à cette dérivation est donnée dans la figure 1.2.

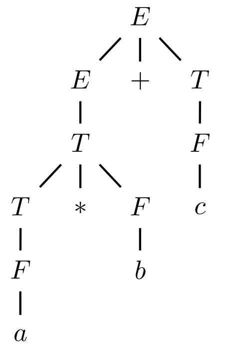

FIG. 1.2 – Un arbre de dérivation pour l'expression $a*b+c$.

A partir de cet arbre, on construit directement une dérivation gauche (ou droite) par un parcours en profondeur. Cela donne

$$\begin{aligned} E &\to E+T \to T+T \to F*T+T \to a*T+T \to a*F+T \\ &\to a*b+T \to a*b+F \to a*b+c\,. \end{aligned}$$

et

$$\begin{aligned} E &\to E+T \to E+F \to E+c \to T+c \to T*F+c \\ &\to T*b+c \to F*b+c \to a*b+c\,. \end{aligned}$$

3.1.5 Dérivations et analyse grammaticale

L'analyse grammaticale de la langue française procède de manière très similaire à l'analyse formelle. Une phrase est décomposée en constituants, et la dépendance de ces composants est représentée par exemple sus une forme arborescente. Dans la figure 1.3, on donne un exemple. L'écriture de l'arbre est inversée, mais la structure est la même.

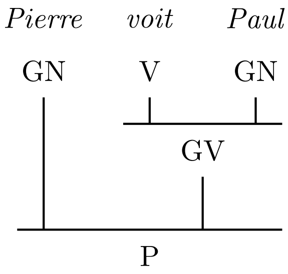

FIG. 1.3 – Un arbre d'analyse grammaticale (GN : groupe nominal, GV : groupe verbal, V : verbe).

On peut compliquer à souhait. Voici une deuxième analyse.

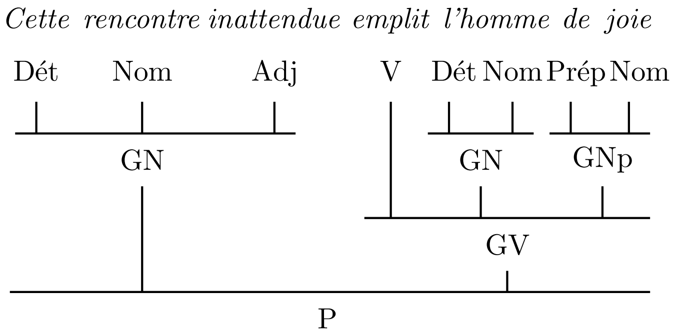

FIG. 1.4 – Analyse d'une phrase plus complexe (GNp : groupe nominal prépositionnel).

3.2 Opérations

3.2.1 Lemmes

Soit $G = (V, P, S)$ une grammaire sur l'alphabet A. On pose, pour $x \in (V \cup A)^*$,

$$L_G(x) = \{w \in A^* \mid x \xrightarrow{*} w\}\,.$$

Bien entendu, on a $L_G(S) = L(G)$, et $L_G(x) = \{x\}$ si $x \in A^*$.

LEMME 2.1. *Si $x \to y$, alors $L_G(x) \supset L_G(y)$.* ∎

LEMME 2.2. *Soit x un mot qui n'est pas dans A^*, et soient $y_1, \ldots, y_n$ les mots qui dérivent immédiatement de x. Alors*

$$L_G(x) = L_G(y_1) \cup \cdots \cup L_G(y_n)\,.$$ ∎

COROLLAIRE 2.3. *Soit X une variable, et soient $X \to \alpha_1 \mid \cdots \mid \alpha_n$ les règles de X. Alors*

$$L_G(X) = L_G(\alpha_1) \cup \cdots \cup L_G(\alpha_n)\,.$$ ∎

Nous en venons maintenant à la concaténation. Le lemme qui suit explique en quoi le terme "context-free" est justifié : lorsque l'on dérive un produit, la dérivation d'un facteur n'influe pas sur l'autre.

LEMME 2.4. *Soient u, v, w des mots. On a*

$$uv \to w$$

si et seulement s'il existe u', v' tels que

$$w = u'v'$$

et

$$u \to u', v = v' \text{ ou } u = u', v \to v'.$$

Preuve. La condition est suffisante : supposons que $u \to u'$ et $v = v'$ (l'autre cas se traite de façon symétrique). Alors $uv \to u'v = u'v' = w$.

Réciproquement, si $uv \to w$, il existe une factorisation $uv = xXy$ et une règle (X, α) telles que $w = x\alpha y$. Ou bien xX est préfixe de u, ou bien Xy est suffixe de v. Dans le premier cas, il existe un mot t tel que

$$u = xXt,\ tv = y$$

et alors $w = x\alpha tv$. Posons $u' = x\alpha t$, $v' = v$. On a bien $u \to u'$ et $w = u'v'$. Le deuxième cas se traite de façon symétrique. ∎

LEMME 2.5. *Soient u_1, u_2, w des mots et $k \geq 0$ un entier. On a*

$$u_1 u_2 \xrightarrow{k} w$$

si et seulement s'il existe des mots v_1, v_2 et deux entiers $k_1, k_2 \geq 0$ tels que

$$u_1 \xrightarrow{k_1} v_1,\ u_2 \xrightarrow{k_2} v_2,\ k_1 + k_2 = k,\ w = v_1 v_2.$$

Preuve. Si $u_1 \xrightarrow{k_1} v_1$ et $u_2 \xrightarrow{k_2} v_2$, alors

$$u_1u_2 \xrightarrow{k_1} v_1u_2 \xrightarrow{k_2} v_1v_2$$

donc

$$u_1u_2 \xrightarrow{k_1+k_2} v_1v_2$$

Réciproquement, supposons que $u_1u_2 \xrightarrow{k} w$. Si $k = 0$, il n'y a rien à prouver. Si $k > 0$, il existe un mot w' tel que $u_1u_2 \to w'$ et $w' \xrightarrow{k-1} w$. Par le lemme précédent, il existe une factorisation $w' = x_1x_2$ telle que $u_1 \to x_1$ et $u_2 = x_2$ ou $u_1 = x_1$ et $u_2 \to x_2$. Comme $x_1x_2 \xrightarrow{k-1} w$, il existe, par récurrence, deux entiers ℓ_1, ℓ_2 et deux mots v_1, v_2 tels que

$$x_1 \xrightarrow{\ell_1} v_1,\ x_2 \xrightarrow{\ell_2} v_2,\ \ell_1 + \ell_2 = k - 1,\ w = v_1v_2.$$

Si $u_1 \to x_1$ et $u_2 = x_2$, alors

$$u_1 \xrightarrow{\ell_1+1} v_1,\ u_2 \xrightarrow{\ell_2} v_2$$

et si $u_1 = x_1$ et $u_2 \to x_2$

$$u_1 \xrightarrow{\ell_1} v_1,\ u_2 \xrightarrow{\ell_2+1} v_2$$

Dans les deux cas, le lemme est vérifié. ∎

Corollaire 2.6. *Soient $u_1, \ldots, u_n, w$ des mots et $k \geq 0$ un entier. On a*

$$u_1 \cdots u_n \xrightarrow{k} w$$

si et seulement s'il existe des mots $v_1, \ldots, v_n$ et des entiers $k_1, \ldots, k_n \geq 0$ tels que

$$u_i \xrightarrow{k_i} v_i, (1 \leq i \leq n), \quad k_1 + \cdots + k_n = k, \quad w = v_1 \cdots v_n.$$

Preuve. Elle est immédiate par récurrence sur n. ∎

Lemme 2.7. `(Lemme fondamental)` *Soient $u_0, \ldots, u_n \in A^*$ et $X_1, \ldots, X_n$ des variables. Soit w un mot et $k \geq 0$ un entier. Alors*

$$u_0X_1u_1X_2 \cdots X_nu_n \xrightarrow{k} w$$

si et seulement s'il existe des mots $v_1, \ldots, v_n$ et des entiers $k_1, \ldots, k_n \geq 0$ tels que

$$X_i \xrightarrow{k_i} v_i\ (1 \leq i \leq n), \quad k_1 + \cdots + k_n = k, \quad w = u_0v_1u_1v_2 \cdots v_nu_n.$$

Preuve. Le corollaire précédent s'applique ici directement. Comme les u_i ne contiennent pas de variable, la seule dérivation à partir d'un u_i est d'ordre 0. ∎

Ces résultats impliquent le corollaire suivant :

Proposition 2.8. *Quels que soient les mots x et y, on a $L_G(xy) = L_G(x)L_G(y)$.*

Preuve. On a $w \in L_G(xy)$ si et seulement si $xy \xrightarrow{*} w$. Or, par le corollaire, ceci se produit si et seulement s'il existe une factorisation $w = uv$ telle que $x \xrightarrow{*} u$ et $y \xrightarrow{*} v$, donc si et seulement s'il existe une factorisation $w = uv$ avec $u \in L_G(x)$ et $v \in L_G(y)$. ∎

3.2.2 Opérations

Un langage $L \subset A^*$ est *algébrique* ou *context-free* s'il existe une grammaire G sur A telle que $L = L(G)$. On note $\mathrm{Alg}(A^*)$ la famille des langages algébriques sur A.

THÉORÈME 2.9. *La famille des langages algébriques est fermée par union, produit, étoile, image miroir.*

Preuve. Soient L_1 et L_2 deux langages algébriques sur A, et soient $G_1 = (V_1, P_1, S_1)$ et $G_2 = (V_2, P_2, S_2)$ deux grammaires engendrant respectivement L_1 et L_2. En renommant si nécessaire les variables, on peut supposer V_1 et V_2 disjoints. Soit alors S une nouvelle variable et posons $V = V_1 \cup V_2 \cup \{S\}$.
La grammaire

$$(V, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$$

engendre le langage $L_1 \cup L_2$. La grammaire

$$(V, P_1 \cup \{S \to S_1S_2\}, S)$$

engendre le langage L_1L_2. La grammaire

$$(V, P_1 \cup P_2 \cup \{S \to SS_1, S \to \varepsilon\}, S)$$

engendre le langage L_1^*. La grammaire

$$(V_1, \{(X, \tilde{\alpha}) \mid (X, \alpha) \in P_1\}, S_1)$$

enfin engendre le langage $\tilde{L}_1$. ∎

EXEMPLE 2.10. Le langage

$$\{a^{n_1}b^{n_1}a^{n_2}b^{n_2}\cdots a^{n_k}b^{n_k} \mid k \geq 0, n_1, \ldots, n_k \geq 0\}$$

est l'étoile du langage $\{a^nb^n \mid n \geq 0\}$ et donc est algébrique, engendré par la grammaire

$$S \to ST \mid \varepsilon, \quad T \to aTb \mid \varepsilon$$

Cette grammaire est ambiguë. ◁

COROLLAIRE 2.11. *Tout langage rationnel est algébrique.* ∎

Soient A et B deux alphabets. Un *morphisme* de A^* dans B^* est une application $f : A^* \to B^*$ telle que $f(uv) = f(u)f(v)$ pour tous $u, v \in A^*$. Une *substitution* de A^* dans B^* est une application f de A^* dans l'ensemble $\wp(B^*)$ des parties de B^* telle que $f(uv) = f(u)f(v)$ pour tous $u, v \in A^*$, et $f(\varepsilon) = \{\varepsilon\}$. Une substitution est un morphisme du monoïde A^* dans le monoïde des parties de B^*, pour le produit. On étend f aux parties par

$$f(X) = \bigcup_{x \in X} f(x)$$

Une substitution est *rationnelle* resp. *algébrique* si les langages $f(a)$, pour $a \in A$, sont rationnels, respectivement algébriques. Il en résulte alors que $f(w)$ est rationnel, resp. algébrique, pour tout mot $w \in A^*$, et aussi que $f(X)$ est rationnel resp. algébrique pour toute partie finie de A^*.

Théorème 2.12. **(Théorème de substitution)** *Soit f une substitution algébrique de A^* dans B^*. Pour tout langage algébrique L sur A, le langage $f(L)$ est algébrique sur B.*

Preuve. Pour toute lettre $a \in A$, soit $G_a = (V_a, P_a, S_a)$ une grammaire algébrique sur B qui engendre $f(a) = L(G_a) = L_{G_a}(S_a)$. On peut supposer les alphabets de variables des grammaires G_a disjoints. Soit $G = (V, P, S)$ une grammaire engendrant L, avec V disjoint des V_a. On considère alors la grammaire $H = (W, Q, S)$, avec

$$W = V \cup \bigcup_{a \in A} V_a, \quad Q = P \cup \bigcup_{a \in A} P_a$$

On a $L_H(a) = L_{G_a}(S_a) = f(a)$ pour $a \in A$, et pour $X \in V$

$$L_H(X) = \bigcup_{(X,\alpha) \in P} L_H(\alpha) = f(L_G(X)) \ . \qquad \blacksquare$$

3.3 Complément : systèmes d'équations

Dans cette section, nous décrivons rapidement le lien entre les langages algébriques et les systèmes d'équations polynomiales, lien qui a donné aux langages leur nom.

Soit $G = (V, P, S)$ une grammaire. Il est utile ici de numéroter les variables :

$$V = \{X_1, \ldots, X_N\}$$

Posons

$$P_i = \{\alpha \mid (X_i, \alpha) \in P\} \quad (1 \le i \le N).$$

Le *système d'équations* associé à la grammaire G est le système d'équations

$$X_i = P_i \qquad (i = 1, \ldots, N)$$

Une suite $L = (L_1, \ldots, L_N)$ de parties de A^* est une *solution* du système si

$$L_i = P_i(L) \quad (1 \le i \le N).$$

Dans cette écriture, on associe, à chaque partie finie (polynôme) de $(V \cup A)^*$ une application (fonction polynôme) des N-uplets L de parties de $(V \cup A)^*$ dans les parties de $(V \cup A)^*$ comme suit :

$$\begin{array}{lll}
(1) & P(L) = \bigcup_{w \in P} w(L) & P \subset (V \cup A)^* \\
(2) & uv(L) = u(L)v(L) & u, v \in (V \cup A)^* \\
(3) & X_i(L) = L_i & i = 1, \ldots, N \\
(4) & a(L) = a & a \in A \\
(5) & \varepsilon(L) = \{\varepsilon\} &
\end{array}$$

Théorème 3.1. *Le N-uplet $L = (L_G(X_1), \ldots, L_G(X_N))$ est la plus petite solution, pour l'inclusion, du système d'équations associé à la grammaire.*

Preuve. Montrons d'abord que $L = (L_G(X_1), \ldots, L_G(X_N))$ est une solution. Pour cela, nous constatons que, par les relation (3) – (5) ci-dessus, on a $L_G(X_i) = X_i(L)$ pour $i = 1, \ldots, N$ et pour $w \in A^*$, $L_G(w) = \{w\} = w(L)$. Il en résulte par (2) que, pour tout mot $\alpha \in (V \cup P)^*$, on a $L_G(\alpha) = \alpha(L)$. Or, par le corollaire 2.3, on a $L_G(X_i) = L_G(P_i)$. Il en résulte que

$$L_G(X_i) = \bigcup_{\alpha \in P_i} L_G(\alpha) = \bigcup_{\alpha \in P_i} \alpha(L) = P_i(L)$$

Montrons maintenant que cette solution est la plus petite. Pour cela, considérons une autre solution $M = (M_1, \ldots, M_N)$. Nous montrons que pour tout mot $w \in (V \cup A)^*$, on a

$$L_G(w) = w(L) \subset w(M) \tag{3.1}$$

Pour ce faire, nous montrons d'abord

$$w \to v \implies v(M) \subset w(M) \tag{3.2}$$

En effet, comme $w \to v$, il existe une règle (X_i, α) et deux mots x, y tels que

$$w = xX_iy, \quad v = x\alpha y.$$

Il s'en suit que $v(M) = x(M)\alpha(M)y(M)$, et comme $\alpha \in P_i$, $v(M) \subset x(M)P_i(M)y(M)$. Or, M est solution, donc $P_i(M) = M_i$. Il en résulte que

$$v(M) = x(M)\alpha(M)y(M) \subset x(M)M_iy(M) = w(M)$$

Ceci montre 3.2. Par récurrence sur l'ordre de la dérivation, on en déduit que

$$w \xrightarrow{*} v \implies v(M) \subset w(M)$$

Soit maintenant $u \in L_G(w)$. Alors $w \xrightarrow{*} u$, et par ce qui précède, on a $u(M) \subset w(M)$. Comme $u \in A^*$, on a $u(M) = \{u\}$, et par conséquent $u \in w(M)$. Ceci prouve 3.1 et par là même le théorème. ∎

3.4 Vérifications

Dans cette section, nous présentons quelques procédés qui permettent de simplifier des grammaires. Le premier s'apparente à la construction d'un automate émondé, la deuxième à l'élimination des ε-transitions dans un automate. La dernière étape – la suppression des règles unitaires – est aussi de cette nature.

Ces opération produisent des grammaires appelées *réduites* et *propres*. Leur intérêt est de faciliter les tâches suivantes, la mise sous forme normale.

3.4.1 Grammaires réduites

Soit $G = (V, P, S)$ une grammaire sur A. Une variable X est

- *productive* si $L_G(X) \neq \emptyset$;
- *accessible* s'il existe des mots α, β tels que $S \xrightarrow{*} \alpha X \beta$.

- *utile* si elle est productive et il existe des mots α, β tels que $S \xrightarrow{*} \alpha X \beta$ et α, β ne contiennent que des variables productives.

Une grammaire est *réduite* si toutes ses variables sont utiles. En supprimant les variables inutiles dans une grammaire, on ne change pas le langage engendré — sauf si l'axiome lui-même est inutile, c'est-à-dire si la grammaire engendre le langage vide.

L'algorithme suivant calcule l'ensemble des variables productives par une méthode qui rappelle un calcul de descendants.

Algorithme de calcul des variables productives.

1. Calculer l'ensemble V_0 des variables X pour lesquelles il existe une règle $X \to \alpha$ avec $\alpha \in A^*$.
2. Calculer l'ensemble V_{i+1} formé de V_i et des variables X pour lesquelles il existe une règle $X \to \alpha$ avec $\alpha \in (A \cup V_i)^*$.
3. Arrêter lorsque $V_{i+1} = V_i$. Cet ensemble est l'ensemble des variables productives.

L'algorithme suivant calcule l'ensemble des variables accessibles à partir de l'axiome.

Algorithme de calcul des variables accessibles.

1. Poser $W_0 = \{S\}$.
2. Calculer l'ensemble W_{i+1} formé de W_i et des variables X telles qu'il existe une règle $Y \to \alpha X \beta$ avec $Y \in W_i$.
3. Arrêter lorsque $W_{i+1} = W_i$. Cet ensemble est l'ensemble des variables accessibles.

Pour déterminer l'ensemble des variables utiles, il ne suffit pas de faire l'intersection les ensembles calculés par ces deux algorithmes. Il convient d'appliquer l'algorithme suivant qui sert à *réduire* une grammaire.

Algorithme de réduction d'une grammaire.

1. On détermine les variables productives, par l'algorithme ci-dessus.
2. On supprime les variables improductives, et les règles où elles figurent.
3. Si l'axiome S est improductif, la grammaire réduite a pour seule variable S, et un ensemble de règles vide.
4. Si l'axiome S est productif, on détermine toutes les variables accessibles de S. Ceci donne les variables utiles.
5. On supprime les autres variables, et les règles où elles figurent. La grammaire obtenue est la grammaire réduite.

Exemple 4.1. On considère la grammaire

$$\begin{aligned} S &\to a \mid X \\ X &\to XY \\ Y &\to b \end{aligned}$$

Les variables productives sont Y et S, et X est donc improductif. Après suppression de X et des règles où X apparaît, il reste la grammaire

$$\begin{aligned} S &\to a \\ Y &\to b \end{aligned}$$

Evidemment, seule S est accessible et la grammaire réduite est

$$S \to a\,.$$

Si l'on calcule les variables accessibles à partir de la grammaire de départ, on trouve que toutes les variables sont accessibles. L'intersection des variables accessibles et des variables productives de la grammaire d'origine ne fournit donc pas les variables utiles. ◁

Exemple 4.2. On considère la grammaire

$$\begin{aligned}
&S \to XYZW \\
&X \to cX \\
&Y \to ab \\
&Z \to cYa \mid WSW \\
&W \to \varepsilon
\end{aligned}$$

Les variables productives sont Y, Z, W. En particulier, S est improductif. ◁

3.4.2 Grammaires propres

Soit $G = (V, P, S)$ une grammaire sur A. Une *ε-règle* est une règle de la forme

$$X \to \varepsilon$$

donc une règle dont le membre droit est le mot vide. Un règle *unitaire* est une règle de la forme

$$X \to Y \qquad Y \in V$$

donc une règle dont le membre droit est une variable.

Une grammaire est *propre* si elle n'a ni ε-règle ni règle unitaire. Nous allons montrer, dans cette section, que l'on peut toujours rendre une grammaire propre. De manière plus précise, pour toute grammaire G, il existe une grammaire G' qui est propre et qui lui est équivalente, c'est-à-dire qui engendre le même langage, au mot vide près. Il est bien clair qu'une grammaire sans ε-règle ne pourra jamais engendrer le mot vide. L'équivalence est donc forcément au mot vide près.

Une variable X est *annulable* si $X \xrightarrow{*} \varepsilon$ ou si, de manière équivalente, $\varepsilon \in L_G(X)$. Pour qu'une variable X soit annulable, il suffit qu'il existe une règle $X \to \varepsilon$, ou qu'il existe une règle $X \to Y_1 \cdots Y_n$ où $Y_1, \ldots, Y_n$ sont toutes des variables annulables. Ces conditions sont aussi nécessaires, et sont à la base de l'algorithme suivant.

Algorithme de calcul des variables annulables.

1. Calculer l'ensemble N_0 des variables X telles qu'il existe une règle $X \to \varepsilon$.
2. Calculer l'ensemble N_{i+1} formé de N_i et des variables X telles qu'il existe une règle $X \to \alpha$ avec $\alpha \in N_i^*$.
3. Arrêter lorsque $N_{i+1} = N_i$. Cet ensemble est l'ensemble des variables annulables.

Une fois que les variables annulables sont connues, on modifie la grammaire comme suit.

Algorithme d'élimination des ε-règles.

1. Calculer l'ensemble N des variables annulables.
2. Remplacer chaque règle $X \to \alpha$ par toutes les règles obtenues en remplaçant, de toutes les façons possibles, les occurrences de variables annulables par le mot vide. S'il y a n occurrences de variables annulables dans α, cela donne 2^n règles.
3. Supprimer les ε-règles. La grammaire obtenue est équivalente à la grammaire de départ au mot vide près, et est sans ε-règle.

EXEMPLE 4.3. On considère la grammaire du langage de Dyck $S \to aSbS \mid \varepsilon$. Remplacer la variable S par le mot vide de toutes les façons possibles dans $aSbS$ donne les 4 mots $aSbS$, abS, aSb et ab. La grammaire propre obtenue est

$$S \to aSbS \mid abS \mid aSb \mid ab\,.$$

EXEMPLE 4.4. Une autre grammaire du langage de Dyck est $S \to aSb \mid SS \mid \varepsilon$. La première règle donne $S \to aSb \mid ab$, la deuxième $S \to SS$ car la règle $S \to S$, qui est générée deux fois par notre algorithme, est inutile. La grammaire propre obtenue est donc

$$S \to aSb \mid SS \mid ab\,.$$

Venons-en aux productions unitaires. On suppose maintenant la grammaire sans ε-règles. En vertu des lemmes prouvés plus haut, on a $L_G(X) \supset L_G(Y)$ si $X \to Y$. Si $X \xrightarrow{*} Y$, alors les règles employées sont toutes unitaires parce que la grammaire est sans ε-règles. On définit une relation $\leq$ sur les variables par

$$X \geq Y \iff X \xrightarrow{*} Y$$

Cette relation n'est pas (encore) un ordre, et on pose $X \sim Y$ si $X \geq Y$ et $Y \geq X$. Si $X \sim Y$, on a $L_G(X) = L_G(Y)$, et on peut donc identifier deux telles variables. Après une telle identification, la relation que nous avons définie devient une relation d'ordre. Il en résulte que si $X \to Y$ est une règle, alors $X > Y$. En particulier, les éléments *minimaux* pour cette relation d'ordre ne sont pas membres gauches de règles unitaires. On peut donc, en procédant à partir des éléments minimaux, remplacer toute règle $X \to Y$ par les règles $X \to \alpha$, pour tous les α tels que $Y \to \alpha$. C'est le procédé que nous décrivons plus formellement maintenant.

Algorithme d'élimination des règles unitaires.

1. Calculer la relation $\geq$ définie par $X \geq Y$ ssi $X \xrightarrow{*} Y$.
2. Calculer la relation d'équivalence de cette relation, définie par $X \sim Y$ ssi $X \geq Y$ et $Y \geq X$.
3. Choisir une variable par classe d'équivalence, et remplacer toutes les occurrences de toutes les variables équivalentes par ce représentant. Supprimer toutes les règles unitaires entre variables équivalentes.

4. En commançant par les variables qui sont minimales dans l'ordre $\geq$, remplaces les règles unitaires $X \to Y$ par toutes les règles $X \to \alpha$, pour tous les α tels que $Y \to \alpha$. La grammaire obtenue est sans règles unitaires.

Exemple 4.5. Revenons à la grammaire des expressions arithmétiques (exemple 1.2) :

$$\begin{aligned}&E \to E + T \mid T\\&T \to T * F \mid F\\&F \to (E) \mid a \mid b \mid c\end{aligned}$$

Il y a deux règles unitaires : $E \to T$ et $T \to F$. L'ordre est $E > T > F$. On substitue à $T \to F$ les règles $T \to (E) \mid a \mid b \mid c$, et de même pour E. On obtient finalement :

$$\begin{aligned}&E \to E + T \mid T * F \mid (E) \mid a \mid b \mid c\\&T \to T * F \mid (E) \mid a \mid b \mid c\\&F \to (E) \mid a \mid b \mid c\end{aligned}$$

◁

3.5 Formes normales

Plusieurs formes normales existent. Nous en présentons deux. La première, la forme normale de Chomsky, est très ancienne, et est donnée à titre historique. La deuxième, la forme normale de Greibach, est plus sophistiquée. Elle a un intérêt théorique, même si dans les applications, en analyse syntaxique par exemple, on ne l'utilise pas.

En effet, dans les applications, la simplicité d'une grammaire est bien plus importante qu'une forme normale. Un telle faorme normale, en ajoutant des variables auxiliaires, et en remplaçant des règles par d'autres, peut profondément modifier l'intuition que l'utilisateur de la grammaire déduit d'une structure plus naturelle.

3.5.1 Forme normale de Chomsky

Une grammaire est en *forme normale de Chomsky*, aussi appelée forme quadratique de Chomsky, si ses règles sont toutes de la forme

$$X \to YZ \quad \text{ou} \quad X \to a$$

où Y, Z sont des variables et $a \in A$ est une lettre terminale. En d'autres termes, les membres droits de règles sont de longueur 1 ou 2. S'ils sont de longueur 1, ce sont des lettres, s'ils sont de longueur 2, ce sont de mots formés de deux variables.

Algorithme de mise en forme normale de Chomsky.

On part d'une grammaire propre.

1. On introduit un nouvel ensemble de variables $Z = \{Z_a \mid a \in A\}$ en bijection avec A, et on ajoute les règles $Z_a \to a$ pour $a \in A$.
2. Toute règle $X \to \alpha$ où α est de longueur 1 est conservée.

3. Toute règle $X \to \alpha$ où $|\alpha| \geq 2$ est transformée en deux étapes
   - toute lettre terminale a dans α est remplacée par la variable Z_a ;
   - si $|\alpha| > 2$, soit $\alpha = Y_1 \cdots Y_m$, on introduit de nouvelles variables $T_1 \ldots, T_{m-2}$, et on remplace la règle $X \to Y_1 \cdots Y_m$ par les $m-1$ règles

$$X \to Y_1T_1,\ T_1 \to Y_2T_2,\ \ldots,\ T_{m-3} \to Y_{m-2}T_{m-2},\ T_{m-2} \to Y_{m-1}Y_m$$

4. La grammaire obtenue est en forme normale de Chomsky

La grammaire ainsi obtenue peut contenir des règles inutiles de la forme $Z_a \to a$.

Exemple 5.1. Considérons la grammaire des mots de Dyck

$$S \to aSbS \mid abS \mid aSb \mid ab$$

On introduit d'abord deux variables Z_a et Z_b, et on remplace les lettres terminales. La grammaire devient

$$\begin{aligned} S &\to Z_aSZ_bS \mid Z_aZ_bS \mid Z_aSZ_b \mid Z_aZ_b \\ Z_a &\to a \\ Z_b &\to b \end{aligned}$$

On introduit maintenant des variables et règles $S \to Z_aT_1$, $T_1 \to ST_2$ et $T_2 \to Z_bS$. On utilise de plus $T_3 \to Z_aZ_b$. La grammaire devient (l'écriture n'est évidemment pas unique) :

$$\begin{aligned} S &\to Z_aT_1 \mid Z_aT_2 \mid Z_aT_3 \mid Z_aZ_b \\ T_1 &\to ST_2 \\ T_2 &\to Z_bS \\ T_3 &\to SZ_b \\ Z_a &\to a \\ Z_b &\to b \end{aligned}$$

3.5.2 Forme normale de Greibach

Une grammaire $G = (V, S, P)$ sur l'alphabet A est en *forme normale de Greibach* si toutes ses règles sont de la forme

$$X \to aY_1 \cdots Y_m, \quad a \in A,\ Y_i \in V\,.$$

Une telle règle se réduit à $X \to a$ si $m = 0$.

Exemple 5.2. La grammaire

$$S \to aSS \mid b$$

du langage de Lukasiewicz est en forme normale de Greibach.

Une grammaire en forme normale de Greibach et toujours propre. Nous allons prouver le théorème suivant

Théorème 5.3. *Pour toute grammaire propre, il existe une grammaire équivalente en forme normale de Greibach.*

La preuve est effective.

On utilise, dans la construction, de façon répétée deux opérations que nous décrivons auparavant, et qui – tout en transformant la grammaire – ne modifient pas le langage. Nous les formulons sous forme de lemmes. Le premier est clair.

LEMME 5.4. *Soient $Y \to \beta_1, \dots, Y \to \beta_r$ les règles de Y. On ne change pas le langage engendré par la grammaire en remplaçant une règle $X \to \alpha Y \alpha'$ par les règles $X \to \alpha\beta_1\alpha' \dots, X \to \alpha\beta_r\alpha'$.* ∎

Le deuxième traite de la récursivité gauche : une variable X est *récursive gauche* s'il existe une règle $X \to X\alpha$. Le lemme qui suit montre comment supprimer la récursivité gauche.

LEMME 5.5. *Si les règles de X sont*

$$\begin{array}{l} X \to X\alpha_1 \mid X\alpha_2 \mid \cdots \mid X\alpha_r \\ X \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_s \end{array}$$

on ne change pas le langage engendré en introduisant une nouvelle variable Y, et en remplaçant les règles par les règles suivantes :

$$\begin{array}{ll} X \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_s & \quad Y \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_r \\ X \to \beta_1 Y \mid \beta_2 Y \mid \cdots \mid \beta_s Y & \quad Y \to \alpha_1 Y \mid \alpha_2 Y \mid \cdots \mid \alpha_r Y \end{array}$$

Preuve. Considérons une dérivation $X \xrightarrow{*} w$ dans la première grammaire. Il lui correspond un arbre de dérivation comme celui décrit dans la partie gauche de la figure 5.1.

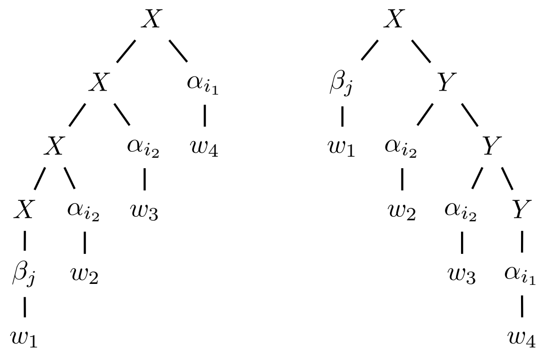

FIG. 5.1 – Arbres de dérivation pour $w = w_1w_2w_3w_4$ avec élimination de la récursivité gauche.

De manière plus précise, une dérivation gauche de X en w se décompose en

$$\begin{array}{rcl} X & \to & X\alpha_{i_1} \\ & \to & X\alpha_{i_2}\alpha_{i_1} \\ & & \cdots \\ & \to & X\alpha_{i_m}\cdots\alpha_{i_2}\alpha_{i_1} \\ & \to & \beta_j\alpha_{i_m}\cdots\alpha_{i_2}\alpha_{i_1} \\ & \xrightarrow{*} & w_0\cdots w_m \end{array}$$

Il lui correspond, dans la grammaire transformée, une dérivation droite

$$\begin{aligned}
X &\to \beta_j Y \\
&\to \beta_j \alpha_{i_m} Y \\
&\to \beta_j \alpha_{i_m} \alpha_{i_{m-1}} Y \\
&\quad \cdots \\
&\to \beta_j \alpha_{i_m} \alpha_{i_{m-1}} \cdots \alpha_{i_2} Y \\
&\to \beta_j \alpha_{i_m} \alpha_{i_{m-1}} \cdots \alpha_{i_2} \alpha_{i_1} \\
&\xrightarrow{*} w_0 \cdots w_m
\end{aligned}$$

Ceci prouve le lemme. ∎

Nous sommes prêts pour la description de l'algorithme de mise en forme normale de Greibach.

Algorithme de mise en forme normale de Greibach.

Soit $G = (V, P, S)$ une grammaire sur A. On numérote les variables : soient $X_1, \ldots, X_m$ les variables numérotées. Les règles sont classées en trois catégories :
– type 1 : $X_i \to a\alpha$ pour une lettre terminale a.
– type 2 : $X_i \to X_j\alpha$ avec $j > i$.
– type 3 : $X_i \to X_j\alpha$ avec $j \leq i$.
Les variables X_j des règles de type 2 ou 3 sont appelées variables de tête.

1. On élimine les règles de type 3, par indice i croissant. Si on a éliminé les règles de type 3 pour $X_1, \ldots, X_{i-1}$, on les élimine pour X_i en deux étapes :
   – On substitue à chaque X_j de tête, avec $j < i$, toutes les règles pour X_j précédemment construites. Après cette substitution, les variables en tête ont un indice strictement plus grand que j. On continue jusqu'à ne plus avoir de règles de type 3, sauf avec $j = i$, c'est-à-dire des règles récursives gauches.
   – On élimine toutes les règles récursives gauche pour X_i par la méthode exposée plus haut. Ceci introduit de nouvelles variables.
2. Les règles sont maintenant toutes de type 1 ou 2, sauf les règles pour les nouvelles variables. On substitue maintenant, par indice de variable décroissant, aux occurrences des variables X_j en tête des membres droits de règles les règles, les membres droits des X_j-règles. Ceci fait disparaître les variables en tête dans les règles de type 2.
3. Les membres droits de règles des nouvelles variables commencent tous par des variables anciennes ou des lettres. Si ce sont des variables, on leur substitue leurs membres droits de règles.

Exemple 5.6. Voici une grammaire, avec les règles déjà numérotées

$$\begin{aligned}
X_1 &\to X_2X_3 \\
X_2 &\to X_3X_1 \mid b \\
X_3 &\to X_1X_2 \mid a
\end{aligned}$$

La seule règle de type 3 est la règle $X_3 \to X_1X_2$. On remplace cette règle par un ensemble de règles : d'abord, on substitue à la variable de tête X_1 son membre droit de règle. On obtient

$$X_3 \to X_2X_3X_2\,,$$

puis la variable X_2 en tête est remplacée par ses deux membres droits de règle : on obtient finalement

$$X_3 \to X_3X_1X_3X_2 \mid bX_3X_2\,.$$

La grammaire s'écrit donc

$$\begin{aligned}
X_1 &\to X_2X_3\\
X_2 &\to X_3X_1 \mid b\\
X_3 &\to X_3X_1X_3X_2 \mid bX_3X_2 \mid a
\end{aligned}$$

On élimine maintenant la récursivité gauche par l'algorithme de bascule : on introduit une variable nouvelle Y, et les règles

$$\begin{array}{ll}
X_3 \to bX_3X_2 \mid a & \qquad Y \to X_1X_3X_2\\
X_3 \to bX_3X_2Y \mid aY & \qquad Y \to X_1X_3X_2Y
\end{array}$$

Il en résulte la grammaire :

$$\begin{aligned}
X_1 &\to X_2X_3\\
X_2 &\to X_3X_1 \mid b\\
X_3 &\to bX_3X_2 \mid a \mid bX_3X_2Y \mid aY\\
Y &\to X_1X_3X_2 \mid X_1X_3X_2Y
\end{aligned}$$

Cette grammaire est sans règles de type 3. On peut donc faire disparaître les règles de type 2, en substituant aux occurrences de X_3 en tête ses règles, puis de même pour X_2. On obtient successivement

$$X_2 \to bX_3X_2X_1 \mid aX_1 \mid bX_3X_2YX_1 \mid aYX_1 \mid b$$

et

$$X_1 \to bX_3X_2X_1X_3 \mid aX_1X_3 \mid bX_3X_2YX_1X_3 \mid aYX_1X_3 \mid bX_3$$

d'où la grammaire

$$\begin{aligned}
X_1 &\to bX_3X_2X_1X_3 \mid aX_1X_3 \mid bX_3X_2YX_1X_3 \mid aYX_1X_3 \mid bX_3\\
X_2 &\to bX_3X_2X_1 \mid aX_1 \mid bX_3X_2YX_1 \mid aYX_1 \mid b\\
X_3 &\to bX_3X_2 \mid a \mid bX_3X_2Y \mid aY\\
Y &\to X_1X_3X_2 \mid X_1X_3X_2Y
\end{aligned}$$

Il reste à remodeler les règles de Y, en substituant les règles de X_1 en tête. D'où finalement

$$\begin{aligned}
X_1 &\to bX_3X_2X_1X_3 \mid aX_1X_3 \mid bX_3X_2YX_1X_3 \mid aYX_1X_3 \mid bX_3\\
X_2 &\to bX_3X_2X_1 \mid aX_1 \mid bX_3X_2YX_1 \mid aYX_1 \mid b\\
X_3 &\to bX_3X_2 \mid a \mid bX_3X_2Y \mid aY\\
Y &\to bX_3X_2X_1X_3X_3X_2 \mid aX_1X_3X_3X_2 \mid bX_3X_2YX_1X_3 \mid aYX_1X_3X_3X_2 \mid bX_3X_3X_2 \mid\\
&\quad\ bX_3X_2X_1X_3X_3X_2Y \mid aX_1X_3X_3X_2Y \mid bX_3X_2YX_1X_3Y \mid aYX_1X_3X_3X_2Y \mid\\
&\quad\ bX_3X_3X_2Y
\end{aligned}$$

∎

EXEMPLE 5.7. Voici, comme deuxième exemple, peut-être plus simple, la grammaire de la forme postfixée du langage de Lukasiewicz

$$S \to SSa \mid b$$

On élimine la récursivité gauche par l'algorithme de bascule :

$$\begin{aligned} S &\to b \mid bT \\ T &\to Sa \mid SaT \end{aligned}$$

Ensuite, il suffit de remplace la variable S en tête des règles de T par ses membres droits de règle. Ceci donne

$$\begin{aligned} S &\to b \mid bT \\ T &\to ba \mid bTa \mid baT \mid bTaT \end{aligned}$$

On ne s'étonne pas d'obtenir, pour T un langage de Dyck retourné : ceci est bien la relation entre le langages de Lukasiewicz et le langage de Dyck. $\triangleleft$

3.6 Automates à pile

3.6.1 Définition et exemples

Il existe plusieurs variations sur les automates à pile. Nous en présentons les plus courantes dans la théorie. Dans les applications, à l'analyse syntaxique notamment, les automates sont légèrement différents.

Nous introduisons d'abord les *machines à pile*, et ensuite les automates à pile. Les automates à pile sont des machines à pile dotées d'une condition d'acceptation.

Une *machine à pile* $\mathcal{M} = (Q, Z, i, R)$ sur un alphabet A est composée des données suivantes :
– un ensemble fini d'états Q,
– un *alphabet de pile* Z,
– une *configuration initiale* $i \in Q \times Z$,
– un ensemble *fini* de *transitions* $R \subset Q \times Z \times (A \cup \{\varepsilon\}) \times Q \times Z^*$.

De manière imagée, une machine à pile est composée de trois unités :
– une unité centrale, dont la configuration est symbolisée par un état, élément de Q ;
– deux canaux, l'un de lecture, l'autre de lecture-écriture ; le premier contient un mot à analyser ; le deuxième est organisé en pile, et sert à contenir de l'information auxiliaire, en quantité non bornée.

La *pile* contient, à tout moment, un mot h sur Z. La machine se trouve dans un état q. Le couple (q, h) est appelé une *configuration* de la machine.

En fonction de q et h, et de ce qui peut être vu sur le canal d'entrée, la machine applique une transition appropriée prise dans R.

Une *transition* (p, z, s, q, d) comporte un état de départ p, un état d'arrivée q. Elle lit un symbole s (lettre ou mot vide), dépile le symbole z, et empile le mot d. Au lieu d'écrire $(p, z, s, q, d) \in R$, on écrira parfois $(q, d) \in T(p, z, s)$, en considérant T comme une application qui donne, pour un triplet de données (p, z, s) toutes les situations (q, d) possibles.

Dans une description graphique d'une machine, les états sont représentés par des sommets, et une transition (p, z, s, q, d) est représentée par une flèche de l'état p vers l'état q. L'étiquette contient deux parties : la première est le symbole s lu, et la deuxième est formée du couple z, d. Voir la figure 6.1.

$$\begin{tikzcd} \textcircled{p} \arrow[rr, "s/z{,}d"] & & \textcircled{q} \end{tikzcd}$$

FIG. 6.1 – Représentation d'une transition (p, z, s, q, d) dans une machine à pile.

L'ensemble des *configurations* est $Q \times Z^*$. La configuration initiale $i = (q_0, z_0) \in QtimesZ$ est formée d'un *état initial* et d'un *lettre de pile initiale* . Dans la configuration initiale, la pile n'est donc pas vide, mais contient une lettre de l'alphabet de pile.

Un mouvement de la machine représente le passage d'une configuration à une autre. Voici une description précise. Il y a *mouvement* de la configuration (p, h) vers la configuration (q, h'), par lecture de $s \in A \cup \{\varepsilon\}$, et on écrit

$$(p, h) \stackrel{s}{\longmapsto} (q, h')$$

s'il existe un mot $f \in Z^*$ et une transition (p, z, s, q, d) dans R tels que $h = fz$ et $h' = fd$. Un mouvement consiste donc en quatre parties :

(1) Suppression, dans la pile, de la lettre en haut de pile, notée z ;
(2) Écriture, dans la pile, du mot d ;
(3) Lecture du symbole s sur la bande d'entrée ;
(4) Passage de l'état p à l'état suivant q.

Chaque écriture est donc précédée d'un effacement ; ceci n'est qu'une convention, et sert surtout à décrire simplement le fait que la machine se bloque lorsque la pile est vide, puisqu'alors il n'y a rien à effacer.

Chaque effacement est suivi de l'écriture d'un mot d. Si ce mot commence par la lettre que l'on vient d'effacer, c'est comme si l'on n'avait pas dépilé. Si le mot d est vide, cela revient à dépiler seulement. Il est possible, dans ce modèle, d'empiler plusieurs lettres en une seule étape.

Le symbole "lu" sur la bande d'entrée peut être le mot vide. Dans ce cas, cela signifie que rien n'est lu, et en particulier que la tête de lecture n'avance pas.

Enfin, notons qu'il est fort possible que plusieurs règles s'appliquent à une configuration donnée ; dans ce cas, la machine n'est pas déterministe.

REMARQUE 6.1. Comme il ressort de la définition, nous avons "couché la pile" vers la droite : le symbole au sommet de la pile est la dernière lettre du mot de pile. Nous aurions pu aussi bien coucher la pile vers la gauche : le sommet de pile aurait alors été la première lettre. Dans ce cas, la transition se définit formellement par

$$(p, h) \stackrel{s}{\longmapsto} (q, h')$$

s'il existe un mot $f \in Z^*$ et une transition (p,z,s,q,d) dans R tels que $h = zf$ et $h' = df$. Une telle machine est une machine *à pile à gauche*, la première est une machine à pile à droite ou machine à pile tout court. On passe de l'une à l'autre en remplaçant, dans chaque transition, le mot d par $\tilde{d}$. Selon les cas, l'une ou l'autre version est plus commode.

On note $\models\!=$ la fermeture transitive de la relation $\longmapsto$ avec, comme exposant, le mot de A^* qui a permis la suite de mouvements. Formellement,

$$(p,h) \overset{s}{\models\!=} (q,h')$$

lorsqu'il existe des états $p_0,\ldots,p_n$, des symboles $s_1,\ldots,s_n \in A \cup \{\varepsilon\}$ et des mots $h_0,\ldots,h_n \in Z^*$ tels que $(p_0,h_0) = (p,h)$, $(p_n,h_n) = (q,h')$, $s = s_1 \cdots s_n$ et

$$(p_{j-1},h_{j-1}) \overset{s_j}{\longmapsto} (p_j,h_j) \qquad j = 1,\ldots,n\,.$$

Une suite de mouvements est aussi appelé un *calcul*. Une configuration c est *accessible* s'il existe un mot $w \in A^*$ tel que est $i \overset{w}{\models\!=} c$.ù

Un *automate à pile* est une machine à pile munie d'un ensemble de *configurations terminales* $T \subset Q \times Z^*$. On écrit $\mathcal{A} = (Q,Z,i,R,T)$ et la machine $\mathcal{M} = (Q,Z,i,R)$ est la machine sous-jacente. Un calcul $i \overset{w}{\models\!=} t$ est *réussi* pour $\mathcal{A}$ si $t \in T$ (rappelons que i est la configuration initiale). On dit que x est *reconnu* ou *accepté* par l'automate à pile si x est l'étiquette d'un calcul réussi. On note le langage reconnu par l'automate à pile par

$$L(\mathcal{M},T) = \{w \in A^* \mid i \overset{w}{\models\!=} t,\ t \in T\}\,.$$

Les trois façons d'acceptations les plus rencontrées sont par pile vide, par états terminaux, ou par pile vide et états terminaux.
– Acceptation par pile vide : l'ensemble des configurations terminales est $T = Q \times \{\varepsilon\}$.
– Acceptation par états terminaux : l'ensemble des configurations terminales est $T = F \times Z^*$ pour une partie F de Q.
– Acceptation par pile vide et états terminaux : l'ensemble des configurations terminales est $T = F \times \{\varepsilon\}$ pour une partie F de Q.

Les langages acceptés, dans un automate, sont notés de façon différente selon leur mode d'acceptation. On note
– $N(\mathcal{A}) = L(\mathcal{M}, Q \times \{\varepsilon\})$ le langage accepté par pile vide,
– $T(\mathcal{A}) = L(\mathcal{M}, F \times Z^*)$ le langage accepté par états terminaux,
– $L(\mathcal{A}) = L(\mathcal{M}, F \times \{\varepsilon\}$ le langage accepté par pile vide et états terminaux.

On rencontre aussi l'écriture Null($\mathcal{A}$) pour $N(\mathcal{A})$.

Exemple 6.2. Considérons la machine à pile de la figure 6.2. Il y a un seul état, et un seul symbole de pile, S, qui joue aussi le rôle de fond de pile.

La pile sert de compteur : chaque a lu incrémente le compteur, chaque b le décrémente. Initialement, le compteur vaut 1 ; à aucun moment, sauf peut-être à la fin, le compteur ne peut être nul. Les langages $N(\mathcal{M})$ et $L(\mathcal{M})$ coïncident parce qu'il n'y a qu'un seul état. Le langage reconnu est le langage de Lukasiewicz. Le langage $T(\mathcal{M})$ est l'ensemble des préfixes du langage de Lukasiewicz. ◁

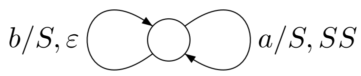

FIG. 6.2 – Un automate à pile pour le langage de Lukasiewicz.

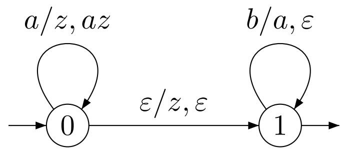

FIG. 6.3 – Un automate à pile pour le langage $\{a^n b^n \mid n \geq 0\}$.

EXEMPLE 6.3. La machine à pile de la figure 6.3 est un automate qui reconnaît le langage $\{a^n b^n \mid n \geq 0\}$.

La configuration initiale est $(0, z)$ et l'état final est 1. On reconnaît par pile vide et état final. Les calculs réussi vont de 0 à 1. Ils empilent autant de fois la lettre a dans l'état 0 qu'ils dépilent des lettres a dans l'état 1. Pour passer de 0 à 1, l'automate dépile la lettre z qui est le fond de pile au début. Il y a, dans cet automate, une ε-transition. On peut s'en passer, en modifiant l'automate. ◁

EXEMPLE 6.4. Voici deux automates à pile pour reconnaître le langage $\{wc\tilde{w} \mid w \in \{a, b\}^*\}$, qui est engendré par la grammaire

$$S \to aSa \mid bSb \mid c\,.$$

Le premier automate empile les lettres, comme il les lit, sur la pile en attendant la lettre c. Puis il dépile en vérifiant que la lettre lue est bien égale à celle empilée.

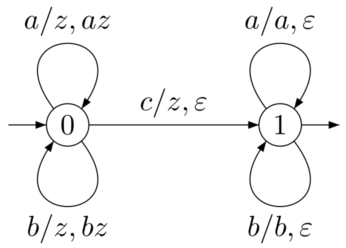

FIG. 6.4 – Un automate à pile pour le langage des palindromes sur $\{a, b\}$ avec marqueur central c.

Le deuxième automate – qui illustre une construction à venir – est non déterministe. Le fond de pile est constitué de l'axiome S. En présence d'un S en sommet de pile, on applique une règle de façon non déterministe. En présence d'une lettre en haut de pile, on vérifie que la lettre lue lui est bien égale.

◁

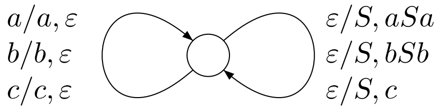

FIG. 6.5 – Un autre automate à pile pour le même langage.

3.6.2 Langages algébriques et automates à pile

Dans cette section, nous prouvons que les langages algébriques sont exactement les langages reconnus par automates à pile acceptant par pile vide avec états terminaux.

THÉORÈME 6.5. *Pour tout langage algébrique L sur A, il existe un automate à pile $\mathcal{A}$ à un seul état tel que $L = L(\mathcal{A})$.*

Preuve. Soit $G = (V, P, S)$ une grammaire pour L. L'automate sera à pile à gauche. Il a un seul état q. Son alphabet de pile est $V \cup A$, le symbole de fond de pile est l'axiome S. Les transitions sont les quintuplets $(q, a, a, q, \varepsilon)$, pour $a \in A$, et $(q, X, \varepsilon, q, \alpha)$, pour toute règle (X, α). Il n'est pas difficile de vérifier que si $S \xrightarrow[g]{*} uX\alpha$, il existe une suite de mouvement

$$(q, S) \overset{u}{\models\!\!=} (q, X\alpha)$$

et réciproquement. Le théorème en résulte. ∎

EXEMPLE. Considérons la grammaire

$$\begin{aligned} S &\to TS \mid \varepsilon \\ T &\to aSb \end{aligned}$$

Les 5 transitions sont :

$$\begin{array}{ll} (1) & (a/a, \varepsilon) \\ (2) & (b/b, \varepsilon) \\ (3) & (\varepsilon, S, TS) \\ (4) & (\varepsilon, S, \varepsilon) \\ (5) & (\varepsilon, T, aSb) \end{array}$$

A la dérivation gauche $S \to TS \to aSbS \to abS \to ab$ correspond la suite de mots de pile :

$$\begin{array}{rl} S & \\ TS & (3) \\ aSbS & (5) \\ SbS & (1) \\ bS & (4) \\ S & (2) \\ \varepsilon & (4) \end{array}$$

THÉORÈME 6.6. *Pour tout automate à pile $\mathcal{A}$, le langage $L(\mathcal{A})$ est algébrique.*

Preuve. Soit $\mathcal{A} = (Q, Z, i, R, T)$ un automate à pile, à *pile à gauche*. Posons $T = F \times \{\varepsilon\}$ avec $F \subset Q$. Pour tous $p, q \in Q$ et $h \in Z^*$, on définit le langage

$$L(p, h, q) = \{x \in A^* \mid (p, h) \stackrel{x}{\models\!\!=} (q, \varepsilon)\}$$

En particulier,

$$L(p, \varepsilon, q) = \begin{cases} \emptyset & \text{si } p \neq q \\ \{\varepsilon\} & \text{si } p = q \end{cases} \tag{6.1}$$

Par définition, on a, avec $i = (q_0, z_0)$,

$$L(\mathcal{A}) = \bigcup_{t \in F} L(q_0, z_0, t) \tag{6.2}$$

Maintenant, on a la relation (pile à gauche !)

$$L(p, hh', q) = \bigcup_{r \in Q} L(p, h, r) L(r, h', q) \tag{6.3}$$

En effet, pour $x \in L(p, hh', q)$, on a

$$(p, hh') \stackrel{x}{\models\!\!=} (q, \varepsilon)$$

On considère le plus court préfixe y de x tel que

$$(p, hh') \stackrel{y}{\models\!\!=} (r, h')$$

pour un r. Alors on a

$$(p, h) \stackrel{y}{\models\!\!=} (r, \varepsilon)$$

donc $y \in L(p, h, r)$, et bien sûr $(r, h') \stackrel{w}{\models\!\!=} (q, \varepsilon)$ pour $x = yw$.

Maintenant, pour $z \in Z$, on a

$$L(p, z, p') = \bigcup_{(p,z,s,q,\tilde{d}) \in R} sL(q, d, p') \tag{6.4}$$

Les équations (6.1), (6.2), (6.3), (6.4) contiennent les informations dont on a besoin pour construire une grammaire pour les langages $L(p, z, q)$. Elle comporte les variables $[p, z, q]$ pour $p, q \in Q$ et $z \in Z \cup \{\varepsilon\}$. Chaque variable va engendrer un des langages $L(p, z, q)$, Il est commode d'introduire aussi des variables $[p, h, q]$ pour engendrer les langages $L(p, h, q)$, où h parcourt les mots de Z^* de longueur bornée par la longueur maximale des dernières composantes (composante δ) dans les transitions (p, z, s, q, δ) de l'automate. Ceci fait un nombre très grand mais fini de variables. S'y ajoute une variable S, axiome de la grammaire. Chacune des équations (6.1), (6.2), (6.3), (6.4) se traduit en règles de la grammaire comme suit.

L'équation (6.1) donne les règles

$$[p, \varepsilon, p] \to \varepsilon\,, \quad p \in Q$$

Les variables $[p, \varepsilon, q]$ pour $p \neq q$ sont improductives.

L'équation (6.2) se traduit dans les règles

$$S \to [q_0, z_0, t]\,, \quad t \in F$$

Rappelons que S est l'axiome et F est l'ensemble des états terminaux.

Chaque équation (6.3), avec $h, h' \neq \varepsilon$, donne les règles

$$[p, hh', q] \to [p, h, r][r, h', q]\,, \quad r \in Q$$

A chaque fois, la longueur des mots h, h codés dans les variables des membres droits diminue. Ces règles sont toutes quadratiques.

Chaque équation (6.4), donne les règles

$$[p, z, p'] \to s[q, \delta, p']\,, \quad (p, z, s, q, \tilde{\delta}) \in R$$

Il s'agit donc d'un nombre fini de règles, ici aussi. Le langage engendré par cette grammaire est bien $L(\mathcal{A})$. ∎

Exemple 6.7. Considérons l'automate à pile de la figure 6.3 que nous reproduisons ici par commodité.

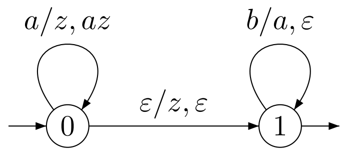

Fig. 6.6 – Un automate à pile pour le langage $\{a^n b^n \mid n \geq 0\}$.

Les états sont $0, 1$, et il y a trois transitions $(0, z, a, 0, az)$, $(0, z, \varepsilon, 1, \varepsilon)$ et $(1, a, b, 1, \varepsilon)$. La première règle est

$$S \to [0, z, 1]$$

car la configuration initiales est $(0, z)$ et le seul état final est 1. Maintenant, on utilise les transitions pour obtenir

$$\begin{aligned} [0, z, 1] &\to a[0, za, 1] \mid [1\varepsilon, 1] \\ [0, z, 0] &\to a[0, za, 0] \\ [1, a, 1] &\to b[1, \varepsilon, 1] \end{aligned}$$

Les décompositions donnent

$$\begin{aligned} [0, za, 1] &\to [0, z, 0][0, a, 1] \mid [0, z, 1][1, a, 1] \\ [0, za, 0] &\to [0, z, 0][0, a, 0] \end{aligned}$$

En fait, on voit facilement que $[0, a, 0]$ et donc $[0, za, 0]$ et $[0, z, 0]$ sont improductives. Après suppression de ces variables, et la substitution de ε aux occurrences de la variable $[1\varepsilon, 1]$ la grammaire devient

$$\begin{aligned} S &\to [0, z, 1] \\ [0, z, 1] &\to a[0, za, 1] \mid \varepsilon \\ [1, a, 1] &\to b \\ [0, za, 1] &\to [0, z, 1][1, a, 1] \end{aligned}$$

On peut identifier S et $[0, z, 1]$, et renommer $[0, za, 1]$ en T et $[1, a, 1]$ en X. L'écriture devient

$$\begin{aligned} S &\to aT \mid \varepsilon \\ X &\to b \\ T &\to SX \end{aligned}$$

Il ne reste plus qu'à faire les substitutions évidentes de X par b et de T par Sb pour obtenir la grammaire $S \to aSb \mid \varepsilon$ qui nous est plus familière.

3.6.3 Equivalences

Dans cette section, nous prouvons que les divers modes de reconnaissance que nous avons introduits sont équivalents. Auparavant, nous considérons encore un autre modèle.

Une machine à pile $\mathcal{A} = (Q, Z, i, R)$ est dite *à fond de pile testable* s'il existe une partition $Z = F \cup Y$ telle que tout mot de pile d'une configuration accessible est dans $\{\varepsilon\} \cup FY^*$.

Proposition 6.8. *Pour tout automate à pile, il existe un automate à pile à fond de pile testable qui lui est équivalent.*

Ceci signifie donc que, quel que soit le mode de reconnaissance, on peut transformer un automate à pile en un automate à pile à fond de pile testable avec le même mode de reconnaissance.

Preuve. Soit $\mathcal{A} = (Q, Z, (q_0, z_0), R, T)$ un automate à pile, soit $\bar{Z}$ une copie de Z, et soit $\gamma : Z^* \to \{\varepsilon\} \cup \bar{Z}Z^*$ l'application définie par $\varepsilon \mapsto \varepsilon$ et $zh \mapsto \bar{z}h$ pour $z \in Z$, $h \in Z^*$. Soit alors

$$\mathcal{B} = (Q, (q_0, \bar{z}_0), Z \cup \bar{Z}, R \cup R', T')$$

avec

$$R' = \{(p, \bar{z}, s, q, \gamma(d)) \mid (p, z, s, q, d) \in R\} \quad T' = \{(q, \gamma(h)) \mid (q, h) \in T\}$$

Toute configuration accessible à partir de $(q_0, \bar{z}_0)$ est dans $\{\varepsilon\} \cup \bar{Z}Z^*$, et $\mathcal{B}$ est donc à fond de pile testable. Il n'est pas difficile de vérifier que

$$L(\mathcal{A}, T) = L(\mathcal{B}, T')$$ ■

Proposition 6.9. *Soit K un langage.*

(1) *Si $K = T(\mathcal{A})$, pour un automate à pile $\mathcal{A}$, alors $K = N(\mathcal{B})$ pour un automate à pile $\mathcal{B}$.*
(2) *Si $K = N(\mathcal{A})$, pour un automate à pile $\mathcal{A}$, alors $K = L(\mathcal{B})$ pour un automate à pile $\mathcal{B}$.*
(3) *Si $K = L(\mathcal{A})$, pour un automate à pile $\mathcal{A}$, alors $K = T(\mathcal{B})$ pour un automate à pile $\mathcal{B}$.*

Preuve. (1) On introduit deux nouveaux états, t et r. Le premier sert à vider la pile lorsque l'on est dans un état final de $\mathcal{A}$, le deuxième sert à empêcher que la pile de $\mathcal{A}$ se vide si l'état d'arrivée n'est pas final. Plus précisément, on considère que $\mathcal{A}$ est à fond de pile testable, et que l'alphabet de fond de pile est E. Soit F l'ensemble des états terminaux de $\mathcal{A}$ Pour tout état final $t \in F$, on ajoute les ε-transitions $(t, z, \varepsilon, r, \varepsilon)$ et $(r, z, \varepsilon, r, \varepsilon)$ pour toute lettre $z \in Z$. Ceci permet de vider la pile. Ensuite, une production qui vide la pile dépile nécessairement une lettre de E. Une telle production $(p, y, s, q, \varepsilon)$ avec $y \in E$ est supprimée si $q \notin F$, et est remplacée par (p, y, s, r, y). Ainsi, la pile n'est plus vidée, et la machine passe dans un état où elle bloque. Il résulte de la construction que le nouvel automate reconnaît K par pile vide.

(2) Il suffit de prendre $\mathcal{B}$ égal à $\mathcal{A}$, avec tous les états terminaux.

(3) On part de $\mathcal{A}$ supposé à fond de pile testable, et on introduit un nouvel état t qui sera l'unique état terminal du nouvel automate. Une production de $\mathcal{A}$ qui aboutit à un acceptation doit vider la pile, et conduire dans un état final. Elle est donc de la forme $(p, y, s, q, \varepsilon)$, où y est un symbole de fond de pile et q et un état final de $\mathcal{A}$. Ces productions sont remplacées, dans le nouvel automate, par la production $(p, y, s, t, \varepsilon)$, alors que les productions $(p, y, s, q, \varepsilon)$, où q n'est pas final, sont supprimées.

Dans le nouvel automate, un calcul aboutit à l'état t toujours avec une pile vide. Ceci montre que les deux automates reconnaissent les mêmes langages. ∎

3.7 Lemme d'itération

Il existe un lemme d'itération pour les langages context-free semblable au lemme d'itération pour les langages réguliers.

THÉORÈME 7.1. `(Lemme d'itération)`. *Pour tout langage context-free L sur un alphabet A, il existe un entier N ne dépendant que de L tel que tout mot w de L de longueur au moins N possède une factorisation $w = xuyvz$ telle que*

1. $|uyv| \leq N$
2. $uv \neq \varepsilon$
3. $xu^nyv^nz \in L$ *pour tout* $n \geq 0$

Voici déjà un exemple d'utilisation de ce lemme.

EXEMPLE 7.2. Le langage $L = \{a^nb^nc^n \mid n \geq 0\}$ n'est pas context-free. Supposons le contraire, et considérons le mot $w = a^Nb^Nc^N$ où N est l'entier du lemme d'itération. Alors w admet une factorisation $w = xuyvz$ avec les propriétés du lemme, et en particulier $xuuyvvz$ est dans L. Il en résulte que u et v sont chacun des puissances d'une lettre, et l'un des mots u ou v n'est pas vide. Si par exemple $u \in a^+$, alors $u = a^k$ pour un entier $k > 0$, et $v = b^\ell$ $xuuyvvz = a^{N+k}b^{N+\ell}c^N$, et ce mot n'est pas dans L. La même contradiction s'obtient si v est une puissance de la lettre c, et si u est une puissance de la lettre b ou de la lettre c. ◁

Comme conséquence immédiate, nous avons la propriété suivante.

PROPRIÉTÉ 7.3. *L'intersection de deux langages algébriques n'est pas toujours un langage algébrique.*

En effet, considérons les langages $R = \{a^n b^n c^k \mid n, k \geq 0\}$ et $R' = \{a^k b^n c^n \mid n, k \geq 0\}$. Ces langages sont algébriques, et leur intersection ne l'est pas puisque c'est le langage L de l'exemple précédent. ◁

En revanche, on a le résultat que voici.

PROPOSITION 7.4. *L'intersection d'un langage algébrique et d'un langage régulier est encore un langage algébrique.*

Preuve. Soit $\mathcal{A} = (Q, (q_0, z_0), Z, R, F \times \{\varepsilon\})$ un automate à pile reconnaissant un langage algébrique (par pile vide et états terminaux) et soit $\mathcal{B} = (P, i, T)$ un automate déterministe complet reconnaissant un langage régulier K. On construit un automate à pile $\mathcal{A}'$ pour l'intersection en faisant le produit de l'automate $\mathcal{A}$ par l'automate $\mathcal{B}$. Les états de $\mathcal{A}'$ sont les éléments de $Q \times P$, la configuration initiale est $((q_0, i), z_0)$, les états terminaux sont $F \times T$, et les flèches sont

$$((p, p'), z, a, (q, p' \cdot a), \delta), \qquad \text{pour } a \in Aet(p, z, a, q, \delta) \in R$$

et

$$((p, p'), z, \varepsilon, (q, p'), \delta), \qquad \text{pour } (p, z, \varepsilon, q, \delta) \in R\,.$$

Il est clair que $\mathcal{A}'$ engendre $L \cap K$. ∎

Pour la démonstration du lemme d'itération, nous avons besoin d'un lemme sur les arbres.

LEMME 7.5. *Dans un arbre de hauteur k où chaque sommet a au plus m fils, le nombre de feuilles est au plus m^k.*

Preuve. Si l'arbre est de hauteur 0, la racine est son unique feuille. Sinon, l'abre a au plus m sous-arbres, chacun de hauteur $k-1$, donc chacun ayant au plus m^{k-1} feuilles. Le nombre total des feuilles est donc au plus m^k. ∎

Preuve du lemme d'itération. Soit L un langage context-free. On suppose L engendré par une grammaire G propre. Soit k le nombre de variables de la grammaire, et soit m la longueur maximale des membres droits des règles de G. On pose $N = m^{(k+1)}$. Nous prouvons le lemme pour cet entier.

Soit w un mot de L. Un arbre de dérivation de w dans G est un arbre où chaque sommet a au plus m fils. Chaque feuille a pour étiquette une lettre, et le nombre de feuilles de l'arbre est égale à la longueur du mot w. Appelons arbre *dépouillé* l'arbre privé de ses feuilles.

Si w est de longueur au moins N, alors l'arbre est de hauteur au moins $k+1$ et l'arbre dépouillé a une hauteur au moins k. Il existe donc un chemin de la racine à une feuille

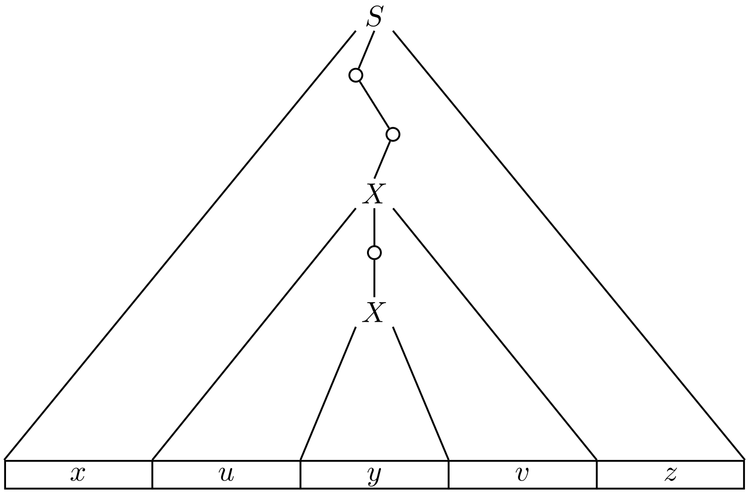

FIG. 7.1 – Lemme d'itération.

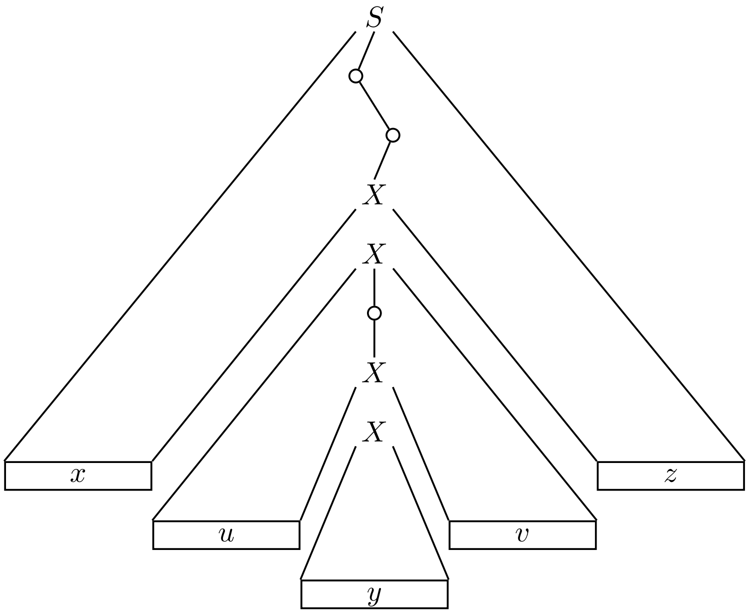

FIG. 7.2 – Lemme d'itération, vue éclatée.

(de l'arbre dépouillé) de longueur au moins k. Il existe donc deux nœuds, sur un chemin de la racine vers une feuille, étiquetés par la même variable.

Considérons un tel chemin de longueur maximale. Les sommets qui le composent, depuis la feuille vers la racine, sont notés $(t_0, t_1, \dots, t_k, \dots, r)$. Ici t_0 est une feuille de l'arbre dépouillé, t_1 son nœud parent, et r la racine. On peut fort bien avoir $t_k = r$. Les étiquettes de $t_0, \dots, t_k$ sont des variables, et comme il y a k variables, il existe deux indices i, j avec $0 \le i < j \le k$ tels que t_i et t_j ont la même variable X en étiquette.

Soient D_i l'arbre de dérivation de racine t_i, et D_j l'arbre de dérivation de racine t_j. Soit y le mot des feuilles de D_i et y' le mot des feuilles de D_j. Comme nous avons choisi un chemin de longueur maximale au départ, la hauteur de D_j est $j \le k$, la longueur de y' est au plus N. De plus, si l'on supprime le sous-arbre de racine t_i dans D_j, en ne gardant que le nœud, on obtient un arbre de dérivation dont le mot des feuilles est uXv pour des mots u, v dont l'un au moins n'est pas vide car t_j a au moins deux fils (car il n'y a pas de règle unitaire) dont l'un n'est pas sur le chemin vers t_i. De la même manière, en supprimant D_i dans l'arbre de dérivation global, on obtient un arbre dont le mot des feuilles est de la forme xXz pour des mots x, z. De plus, on obtient des dérivations

$$S \xrightarrow{*} xXz,\ X \xrightarrow{*} uXv, X \xrightarrow{*} y$$

ce qui démontre le théorème. ∎

Bibliographie

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translating and Compiling*, volume I : Parsing. Prentice Hall, 1972.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translating and Compiling*, volume II : Compiling. Prentice Hall, 1973.

[3] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[4] J. Almeida. *Finite Semigroups and Universal Algebra*. World Scientific, 1994.

[5] A. Arnold. *Finite Transition Systems*. Prentice Hall, 1997.

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

[7] M.-P. Béal. *Codage symbolique*. Masson, 1993.

[8] C. Benzaken. *Systèmes formels*. Masson, Paris, 1991.

[9] J. Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.

[10] M. Crochemore and W. Rytter. *Jewels in Stringology*. Wiley and Sons, 2002.

[11] S. Eilenberg. *Automata, Languages and Machines*, volume B. Academic Press, 1976.

[12] S. Ginsburg. *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, 1966.

[13] E. Graedel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games*, volume 2500 of *Lect. Notes Comp. Sci.* Springer-Verlag, 2002.

[14] D. Gries. *Compiler Construction for Digital Computers*. Wiley and Sons, 1971.

[15] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, 1978.

[16] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2001. second edition.

[17] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[18] J. M. Howie. *Automata and Languages*. Clarendon Press, 1991.

[19] G. Lallement. *Semigroups and Combinatorial Applications*. Wiley and Sons, 1979.

[20] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, 2001. 3è édition.

[21] M. Lothaire. *Combinatorics on Words*, volume 17 of *Encyclopedia of Mathematics*. Addison-Wesley, 1983. Reprinted in the *Cambridge Mathematical Library*, Cambridge University Press, 1997.

[22] M. Lothaire. *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of Mathematics*. Cambridge University Press, 2002.

[23] A. Meduna. *Automata and Languages. Theory and Applications.* Springer-Verlag, 2000.

[24] E. Moore, editor. *Sequential Machines – Selected Papers.* Addison-Wesley, 1964.

[25] D. Perrin and J.-E. Pin. *Automata on Infinite Words.* Academic Press, 2003.

[26] J.-E. Pin. *Varieties of Formal Languages.* Plenum Press, 1986.

[27] E. Roche and Y. Schabes. *Finite-state Language Processing.* MIT Press, 1997.

[28] J. Sakarovitch. *Élements de théorie des automates.* Vuibert, 2003.

[29] A. Salomaa. *Theory of Automata.* Pergamon Press, 1969.

[30] A. Salomaa. *Formal Languages.* Academic Press, 1973.

[31] A. Salomaa. *Computation and Automata.* Cambridge University Press, 1985.

[32] P. Séébold. *Théorie des automates – Méthodes et exercices corrigés.* Vuibert, 1999.

[33] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume 1. MIT Press, 1990.

[34] P. Wolper. *Introduction à la calculabilité.* InterEditions, 1991.