Emmanuel Jeandel
modifié par Vincent Demange

Version du 16 janvier 2023

LANGAGES FORMELS

TABLE DES MATIÈRES

PRÉLIMINAIRES

La plupart des notions abordées dans ce cours sont assez standard, et se retrouvent dans la plupart des livres de théorie des langages.

Même si ce cours ne suit aucun livre en particulier, la lecture des ouvrages suivants est conseillée :

— Olivier Carton, *Langages formels*, Éditions Vuibert (lire la première partie uniquement).

— John E. Hopcroft et Jeffrey D. Ullman, *Introduction to automata theory, Languages and Computation*, Addison Wesley (chapitres 1 à 6 et chapitre 10).

— John E. Hopcroft, Rajeev Motwani et Jeffrey D. Ulman, *Introduction to automata theory, Languages and Computation*, Addison Wesley (nouvelle version du livre précédent, avec des chapitres légèrement différents ; lire Chapitres 1 à 7).

Ces notes correspondent à un cours de 14h, accompagnés de 16h d'exercices. Le cours est composé de deux parties :

— Une première partie sur les automates finis et les langages rationnels.

  ◦ On y introduit d'abord les langages rationnels (sous la forme de regexp).

  ◦ On introduit ensuite les automates déterministes et on montre que tout langage reconnu par un automate déterministe est rationnel.

  ◦ Enfin on donne la définition des automates non déterministes, on démontre que tout langage rationnel est reconnu par un automate non déterministe et que les automates non déterministes sont équivalents aux automates déterministes, ce qui montre que les langages rationnels sont exactement les langages reconnus par les automates déterministes.

  ◦ Dans un dernier chapitre, on explique comment minimiser les automates déterministes, et quelques critères pour montrer qu'un langage n'est pas rationnel.

— Une deuxième partie sur les grammaires (algébriques) et les automates à pile.

  ◦ Le premier chapitre introduit le concept de grammaires, d'arbre de dérivation, et l'algorithme CYK (Cocke-Kasami-Younger) d'analyse de mot par une grammaire quelconque.

  ◦ Le second chapitre s'intéresse aux automates à pile, et démontre l'équivalence entre langages reconnus par automates à pile et langages correspondant à une grammaire.

  ◦ Le dernier chapitre s'intéresse aux automates à pile déterministes, et aux grammaires LR(0) et LR(1), l'accent étant principalement mis sur les grammaires LR(0) et le reste laissé à la sagacité du lecteur.

— Le cours est complété par une annexe qui montre l'utilisation des algorithmes présentés en cours sur des exemples imposants.

Certains choix audacieux et discutables ont été nécessaires :

— Les automates non déterministes à ϵ-transitions n'ont pas été présentés, pour deux raisons :

1. La définition de l'acceptation est assez différente de celles des automates sans ϵ-transitions (il faut renoncer à la définition 3.2 et adopter une définition plus proche de celle donnée ultérieurement pour les automates à pile).
2. Il n'est pas facile aisément de passer d'un automate non-déterministe avec ϵ-transitions à un automate déterministe sans passer par la case automate non déterministe sans ϵ-transitions, ce qui fait qu'une présentation des automates avec ϵ-transitions nécessite *aussi* de passer par le cas particulier des automates non déterministes sans ϵ-transitions, ce qui n'est pas possible dans le temps imparti.

Ce parti pris a deux désavantages : (a) la preuve d'équivalence avec les expressions régulières est un peu plus compliquée (b) les ϵ-transitions surgissent de toute manière dans le chapitre sur les automates à pile, car elles y sont nécessaires.

— Le lemme de l'étoile n'est pas présenté comme un lemme, mais comme une méthode pour prouver qu'un langage n'est pas rationnel à la section 4.3. Il en résulte une utilisation plus simple en TD qui évite la confusion qui pourrait naître des quantificateurs. La version donnée ici pourrait théoriquement s'écrire ainsi : si L est rationnel, il existe un entier n tel que pour tout mot $z = uvw \in L$ tel que $|v| > n$, il existe r, s, t avec $|s| > 0$ tel que $v = rst$ et $urs^\star tw \subseteq L$. L'inconvénient, si c'en est un, est qu'il devient plus difficile de trouver des langages où il ne s'applique pas.

— Les formes normales de Greibach et Chomsky ne sont pas présentées. L'algorithme CYK (Cocke-Kasami-Younger) d'analyse de mot par une grammaire est présenté dans le cas d'une grammaire en forme C2F qui est plus général que le cas des grammaires sous forme normale de Chomsky, et plus simple que le cas général (voir Lange, Leiß 2009). Il permet de traiter de manière simple toute grammaire qui ne contient pas d'ϵ-production. Le cas d'une grammaire générale est présentée brièvement.

— Aucune méthode de preuve (type lemme d'Ogden) n'est donnée pour montrer qu'un langage n'est pas algébrique, ou pas algébrique déterministe, par manque de temps.

— Les grammaires LR(0) et LR(1) sont présentées principalement dans le cadre de grammaires sans ϵ-transitions.

Bonne lecture !

Première partie

Langages rationnels

Chapitre 1

LANGAGES ET LANGAGES RÉGULIERS

1.1 Définitions - Mots

Nous commençons par quelques définitions élémentaires.

Définition 1.1 (*Alphabet*)

Un alphabet est un ensemble fini d'éléments, qu'on appelle lettres ou symboles. L'alphabet sera souvent noté A (comme alphabet), ou encore Σ.

♦ **Exemple**

$A = \{\mathtt{a}, \mathtt{b}\}$ est un alphabet composé de deux lettres.

$\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$ est un autre alphabet, composé de 10 lettres.

$A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ est un alphabet. Il est composé de 26 lettres.

$A = \{\mathtt{o}, \vdash, =\}$ est un alphabet composé de trois lettres.

Définition 1.2 (*Mots*)

Un mot w sur l'alphabet A est une suite finie $w_1 w_2 \dots w_n$ de lettres de l'alphabet A.
n est appelée longueur du mot, et sera également notée $|w|$.
Le mot avec 0 lettre sera noté ϵ.

♦ **Exemple**

$w = \mathtt{abbaba}$ est un mot sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Il est de longueur 6. Sa 5e lettre est un $\mathtt{b}$.

$w = \mathtt{1664}$ est un mot sur l'alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$, de longueur 4.

Définition 1.3 (*Opérations sur les mots*)

On dispose de deux opérations sur les mots d'un même alphabet A :

— La concaténation. Étant donné deux mots u, v, on obtient le mot uv en concaténant le mot v au mot u. C'est donc un mot de longueur $|u| + |v|$.

— Le miroir. Si u est un mot, on note u^R le mot obtenu en lisant u dans l'autre sens. C'est donc un mot de même longueur que u.

♦ **Exemple**

Sur l'alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$, la concaténation des mots $\mathtt{16}$ et $\mathtt{64}$ est le mot $\mathtt{1664}$.

Sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ la concaténation des mots $u = \mathtt{orni}$ et $v = \mathtt{thorynque}$ est le mot $\mathtt{ornithorynque}$.

Sur le même alphabet, le miroir du mot $u = \texttt{engager}$ est le mot $u^R = \texttt{regagne}$.

Sur un alphabet A quelconque, la concaténation du mot ϵ et du mot u est le mot u.

1.2 Langages

Définition 1.4

Un langage $\mathcal{L}$ sur un alphabet A est un ensemble de mots de A.

♦ **Exemple**

Sur l'alphabet $A = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}, \texttt{f}, \texttt{g}, \texttt{h}, \texttt{i}, \texttt{j}, \texttt{k}, \texttt{l}, \texttt{m}, \texttt{n}, \texttt{o}, \texttt{p}, \texttt{q}, \texttt{r}, \texttt{s}, \texttt{t}, \texttt{u}, \texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z}\}$, on peut considérer par exemple :

— Le langage $\mathcal{L}_1$ de tous les mots de longueur 4 : $\mathcal{L}_1$ contient le mot `toto` mais aussi le mot `zwpt`.
— Le langage $\mathcal{L}_2$ de tous les mots du français. Le langage $\mathcal{L}_2$ contient par exemple le mot `ornithorynque` mais pas le mot `mvtmjsun`.
— Le langage $\mathcal{L}_3$ de tous les mots de l'anglais. Le langage $\mathcal{L}_3$ contient le mot `platypus` mais pas le mot `mvemjsun`.
— Le langage $\mathcal{L}_4 = \{\texttt{con}, \texttt{per}\}$.
— Le langage $\mathcal{L}_5 = \{\texttt{fusion}, \texttt{version}, \texttt{sister}\}$.

Définition 1.5 (*Opérations ensemblistes sur les langages*)

Si $\mathcal{L}$ et $\mathcal{L}'$ sont deux langages sur l'alphabet A, on note :
— $\mathcal{L} \cup \mathcal{L}'$ l'union des deux langages : c'est l'ensemble des mots qui sont dans $\mathcal{L}$ ou dans $\mathcal{L}'$. On la note aussi $\mathcal{L} + \mathcal{L}'$.
— $\mathcal{L} \cap \mathcal{L}'$ l'intersection des deux langages : c'est l'ensemble des mots qui sont dans $\mathcal{L}$ et dans $\mathcal{L}'$.
— $\mathcal{L} \setminus \mathcal{L}'$ est l'ensemble des mots qui sont dans $\mathcal{L}$ mais pas dans $\mathcal{L}'$.
— $\mathcal{L}^c$ le complémentaire de $\mathcal{L}$. C'est l'ensemble des mots sur l'alphabet A qui ne sont pas dans $\mathcal{L}$.

Attention, pour la dernière opération, on doit savoir de quel alphabet on parle : le complémentaire du langage n'est pas le même suivant l'alphabet.

♦ **Exemple**

$\mathcal{L}_1 \cup \mathcal{L}_2$ est l'ensemble des mots qui sont de longueur 4, ou qui existent en français. Il contient donc le mot `ornithorynque`, le mot `test`, le mot `rpoi`, mais pas le mot `zaoiruowx`.

$\mathcal{L}_1 \cap \mathcal{L}_2$ est l'ensemble des mots qui sont de longueur 4 qui existent en français. Il contient donc le mot `test`, mais pas le mot `rpoi`, ni le mot `ornithorynque`.

$\mathcal{L}_2 \cap \mathcal{L}_3$ contient les mots du français qui existent aussi en anglais. Il contient donc le mot `sensible` mais pas le mot `ornithorynque` ni le mot `platypus`.

$\mathcal{L}_2 \setminus \mathcal{L}_3$ contient les mots du français qui n'existent pas en anglais. Il contient donc le mot `ornithorynque` mais pas le mot `sensible` ni le mot `platypus`.

$\mathcal{L}_1^c$ est l'ensemble des mots dont la longueur n'est pas 4. Il contient le mot `ornithorynque`, le mot `zpr` et aussi le mot ϵ (le mot de longueur 0).

Définition 1.6

Si $\mathcal{L}$ et $\mathcal{L}'$ sont deux langages sur l'alphabet A, on note :

- $\mathcal{L}\mathcal{L}'$ la concaténation des deux langages : c'est l'ensemble des mots qu'on obtient en concaténant un mot de $\mathcal{L}$ et un mot de $\mathcal{L}'$.
- On note $\mathcal{L}^n$ la concaténation de n mots de $\mathcal{L}$. Par exemple :
  - $\mathcal{L}^1 = \mathcal{L}$.
  - $\mathcal{L}^2 = \mathcal{L}\mathcal{L}$.
  - $\mathcal{L}^0 = \{\epsilon\}$ par convention.
- $\mathcal{L}^\star$ est la concaténation de zéro à plusieurs mots de $\mathcal{L}$.
  On peut donc écrire $\mathcal{L}^\star = \bigcup_{n\geq 0} \mathcal{L}^n$.

♦ **Exemple**

$\mathcal{L}_1\mathcal{L}_1$ est l'ensemble des mots qu'on obtient en concaténant deux mots de 4 lettres. C'est donc l'ensemble des mots de 8 lettres.

$\mathcal{L}_1\mathcal{L}_2$ est l'ensemble des mots qu'on obtient en concaténant un mot de 4 lettres à un mot du français. On y trouve par exemple `ultrornithorynque` mais pas le mot `traduction` (pourquoi ?).

$\mathcal{L}_2\mathcal{L}_2$ est l'ensemble des mots qu'on obtient en concaténant deux mots du français. On y trouve par exemple `soleil` (pourquoi ?) mais pas `traduction` (pourquoi ?).

$\mathcal{L}_4\mathcal{L}_5$ est constitué de six mots : `confusion`, `conversion`, `consister`, `perfusion`, `perversion`, `persister`.

$\mathcal{L}_1^\star$ est l'ensemble des mots qu'on obtient en concaténant 0 ou plusieurs fois des mots de longueur 4. C'est donc l'ensemble des mots dont la longueur est multiple de 4.

$\mathcal{L}_2^\star$ est l'ensemble des mots qu'on obtient en concaténant 0 ou plusieurs mots du français. On y trouve par exemple `soleil`, `traduction` mais pas `ultrornithorynque`.

$A^\star$ est l'ensemble des mots qu'on obtient en concaténant 0 ou plusieurs fois des mots de A. C'est donc exactement l'ensemble de tous les mots sur l'alphabet A.

Et des exemples plus utiles :

Fait 1.1

Pour tout langage L,

$$L + \emptyset = \emptyset + L = L$$
$$L\{\epsilon\} = \{\epsilon\}L = L$$
$$L\emptyset = \emptyset L = \emptyset$$

Les propriétés précédentes montrent que $\emptyset$ et $\{\epsilon\}$ jouent le rôle de 0 et 1 : $\emptyset$ est neutre pour l'addition (union) et absorbant pour la multiplication (concaténation). $\{\epsilon\}$ est le neutre pour la multiplication (concaténation).

1.3 Langages rationnels (regexp)

1.3.1 Définitions

Définition 1.7

Soit A un alphabet. Les langages rationnels sont les langages qu'on peut obtenir à partir de :
- Des ensembles $\{a\}$ pour tout lettre a dans A.
- De l'ensemble vide $\emptyset$ et de l'ensemble $\{\epsilon\}$.
- Des opérations union (+), concaténation et étoile.

♦ **Exemple**

$\{\texttt{a}\}^\star\{\texttt{b}\}+\{\texttt{c}\}+\{\texttt{d}\}$ est un langage rationnel. Par abus de notation, on écrira plutôt $\texttt{a}^\star\texttt{b}+\texttt{c}+\texttt{d}$.

$\{\texttt{b}\}^\star\{\texttt{b}\}+\{\texttt{c}\}+\{\epsilon\}$ est un langage rationnel. Par abus de notation, on écrira plutôt $\texttt{b}^\star\texttt{b}+\texttt{c}+\epsilon$.

Le langage des expressions régulières utilisé ici est celui utilisé dans la plupart des manuels de théorie des langages, mais n'est pas toujours celui utilisé en pratique. Dans la plupart des langages de programmation, le symbole | est utilisé à la place du symbole $+$.

1.3.2 Propriétés des langages rationnels

Les expressions régulières sont utilisées à l'intérieur de nombreux logiciels pour deux raisons :
- Il est rapide de tester, étant donné un mot u et une regexp E, si u appartient au langage représenté par E. Dit autrement, la recherche de motifs dans une chaîne se fait rapidement.
- Il est facile de créer des expressions régulières correspondant à un langage donné, si tant est qu'il soit rationnel.

On va comprendre pourquoi dans les prochains cours.

Pour finir celui-ci, nous allons juste donner deux ou trois méthodes pour construire des expressions régulières.

Fait 1.2

Si A, B, C sont trois langages, alors $A(B+C) = AB+AC$ et aussi $(A+B)C = AC+BC$.

La concaténation est donc distributive sur l'union, ce qui justifie qu'on les note multiplication et addition.

Lemme 1.3 (*Arden*)

Soit A, B deux langages tels que $\epsilon \notin A$.
Alors l'équation $X = AX + B$ a une seule solution qui est $X = A^\star B$.

Attention, dans le cas particulier où $B = \emptyset$ (c'est à dire d'une équation du type $X = AX$) la solution est donc $X = \emptyset$.

Preuve : Soit A et B deux langages tels que $\epsilon \notin A$. On montre que :

$$X = AX + B \Longleftrightarrow X = A^\star B$$

$\boxed{\Leftarrow}$ (existence)

Supposons que $X = A^\star B$. Alors

$$\begin{aligned}
AX + B &= AA^\star B + B \\
&= (AA^\star + \{\epsilon\})B && \text{(distributivité)} \\
&= (A \bigcup_{n\geq 0} A^n + \{\epsilon\})B && \text{(définition de } A^\star) \\
&= (\bigcup_{n\geq 0} AA^n + \{\epsilon\})B && \text{(distributivité)} \\
&= (\bigcup_{n\geq 1} A^n + \{\epsilon\})B \\
&= (\bigcup_{n\geq 1} A^n + A^0)B && \text{(définition de } A^0) \\
&= (\bigcup_{n\geq 0} A^n)B \\
&= A^\star B && \text{(définition de } A^\star) \\
&= X
\end{aligned}$$

$\boxed{\Rightarrow}$ (unicité)

Supposons $X = AX + B$. On montre $X = A^\star B$ par double inclusion :

$\boxed{\subseteq}$ Soit w un mot. On montre par récurrence (forte) sur la longueur de w que

$$\text{si } w \in X \text{ alors } w \in A^\star B$$

Supposons que $w \in X$.

— si $|w| = 0$ alors $w = \epsilon$. Comme $w \in X = AX + B$ et que $\epsilon \notin A$ alors $w \notin AX$, donc nécessairement $w \in B$, et donc $w = \epsilon w \in A^\star B$.

— si $|w| > 0$. Comme $w \in X = AX + B$ on a deux cas :

  — si $w \in B$ alors $w \in A^\star B$ comme précédemment ;

  — si $w \in AX$ alors $w = uv$ avec $u \in A$ et $v \in X$. Comme $\epsilon \notin A$ alors $|u| > 0$ et donc $|v| < |w|$, et alors par hypothèse de récurrence appliquée à v on déduit que $v \in A^\star B$, et donc $w = uv \in AA^\star B \subseteq A^\star B$.

$\boxed{\supseteq}$ On montre par récurrence sur $n \in \mathbb{N}$ que

$$A^n B \subseteq X$$

— si $n = 0$ alors $A^n B = A^0 B = \{\epsilon\}B = B \subseteq AX + B = X$;

— soit $n \in \mathbb{N}$ tel que $A^n B \subseteq X$, alors

$$\begin{array}{rcll}
A^{n+1}B & = & AA^nB & \text{(définition de } A^{n+1}) \\
& \subseteq & AX & \text{(hypothèse de récurrence)} \\
& \subseteq & AX + B = X &
\end{array}$$

D'où on déduit que

$$A^\star B = \bigcup_{n\geq 0} A^n B \subseteq X$$

■

Ce lemme permet de donner facilement des expressions régulières pour certains langages.

♦ **Exemple**

Soit L le langage des mots sur l'alphabet $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$ où tout $\mathtt{a}$ est suivi plus tard d'un $\mathtt{b}$. Par exemple $\mathtt{acbaccb}$ et $\mathtt{aacb}$ sont des mots de L mais pas $\mathtt{acbacc}$.

Pour trouver une expression pour le langage L, on définit un langage auxiliaire : $L_{\mathtt{b}}$ est l'ensemble des mots qui contiennent au moins un $\mathtt{b}$ et où tout $\mathtt{a}$ est suivi plus tard d'un $\mathtt{b}$.

Les deux langages vérifient le système d'équations suivant :

$$\left\{\begin{array}{rcl} L & = & \mathtt{a}L_{\mathtt{b}} + (\mathtt{b}+\mathtt{c})L + \epsilon \\ L_{\mathtt{b}} & = & (\mathtt{a}+\mathtt{c})L_{\mathtt{b}} + \mathtt{b}L \end{array}\right.$$

On résout ce système d'équations de la façon suivante. De la dernière équation on obtient

$$L_{\mathtt{b}} = (\mathtt{a}+\mathtt{c})^{\star}\mathtt{b}L$$

On remplace dans la première équation :

$$L = \mathtt{a}L_{\mathtt{b}} + (\mathtt{b}+\mathtt{c})L + \epsilon = \left[\mathtt{a}(\mathtt{a}+\mathtt{c})^{\star}\mathtt{b} + (\mathtt{b}+\mathtt{c})\right] L + \epsilon$$

D'où on déduit

$$L = (\mathtt{a}(\mathtt{a}+\mathtt{c})^{\star}\mathtt{b} + \mathtt{b} + \mathtt{c})^{\star}\epsilon = (\mathtt{a}(\mathtt{a}+\mathtt{c})^{\star}\mathtt{b} + \mathtt{b} + \mathtt{c})^{\star}$$

Exercices

(1 - 1) *(Opérations sur les langages)*

Prenons l'alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$.
Soit L_1 l'ensemble des mots du français, et L_2 l'ensemble des mots de longueur 4.
Remplir le tableau suivant, précisant si le mot w est ou non dans le langage indiqué :

	L_1	L_2	$L_1 \cap L_2$	$L_1 \cup L_2$	$L_1 L_1$	$L_1 L_1 \cap L_1$	$L_1 L_2 \cap L_2 L_1$
ornithorynque							
soleil							
uniforme							
popimanger							
sombrestes							
vraiment							
ratsment							
test							
ioauzroiuqs							
qjsr							

	$L_1 L_2 \cap L_1$	$L_1 \backslash L_1 L_1$	$(L_2 \cap L_1)(L_2 \cap L_1)$	$((L_2 \cap L_1)(L_2 \cap L_1)) \cap L_1$
ornithorynque				
soleil				
uniforme				
popimanger				
sombrestes				
vraiment				
ratsment				
test				
ioauzroiuqs				
qjsr				

(1 - 2) Remplir le tableau suivant, précisant si le mot w est ou non dans le langage correspondant à l'expression régulière :

	a*	a*+b*	(a+b)*	a*b*	(bb)*	(ab+ba)*	b(a+b)*	a*b(a+b)*	a*b(a+b)*+a*
ϵ									
abab									
baba									
bbbb									
aaa									
bbbbb									
baabaaba									
aaabaaa									
baaa									
aabbbb									
babaabba									

(1 - 3) Pour chacun des langages suivants, décrire le langage en français, puis donner un mot qui appartient au langage et un mot qui n'appartient pas au langage :

$$L_a = AA^\star \qquad L_e = \{uu^R | u \in A^\star, u \neq \epsilon\}$$
$$L_b = A^\star A \qquad L_f = L_e \cap L_1$$
$$L_c = \{uu | u \in A^\star\} \qquad L_g = L_e AA^\star$$
$$L_d = L_c \cap L_1 \qquad L_h = L_g \cap L_1$$

(1 - 4) On se place sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Expliquer en français ce que signifient les expressions régulières suivantes :

$$\mathtt{a}^\star\mathtt{b} \qquad (\mathtt{ab})^\star + (\mathtt{ba})^\star$$
$$((\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b}))^\star \qquad (\mathtt{ab}^\star\mathtt{a}+\mathtt{b})^\star$$

(1 - 5) Écrire une expression régulière pour l'ensemble des mots de longueur impaire sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$, et pour les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui ont un nombre impair de $\mathtt{a}$.

(1 - 6) Retrouver les expressions régulières de la question précédente en écrivant et résolvant un système d'équations. On introduira le langage auxiliaire L' des mots de longueur paire pour le premier cas, et le langage auxiliaire L' des mots qui ont un nombre pair de $\mathtt{a}$ pour le deuxième.

(1 - 7) Dans tout cet exercice, on suppose que l'alphabet est $\{a, b, c\}$. La commande

```
grep -E reg toto.txt
```

permet de donner les lignes dans lesquelles un bout de la ligne correspond à la regexp `reg`.

Q 1) À quel langage rationnel correspond la ligne suivante de grep ?

```
grep -E ab toto.txt
```

`grep` utilise d'autres symboles particuliers. Voici un extrait de `man grep` :

> Le point . correspond à n'importe quel caractère. [...]
> **Ancrage** L'accent circonflexe ^ et le symbole dollar $ sont des métacaractères correspondant respectivement à une chaîne vide au début et en fin de ligne.
> **Répétitions** Dans une expression rationnelle, un caractère peut être suivi par l'un des opérateurs de répétition suivants :
> **?** L'élément précédent est facultatif et peut être rencontré au plus une fois.
> ***** L'élément précédent peut être rencontré zéro ou plusieurs fois.
> **+** L'élément précédent peut être rencontré une ou plusieurs fois.
> [...]
> **Alternatives** Deux expressions rationnelles peuvent être reliées par l'opérateur infixe | ; l'expression résultante correspondra à toute chaîne qui comporte l'une ou l'autre des deux expressions.

Q 2) Donner les langages rationnels correspondant aux lignes de grep suivantes :

```
grep -E a|b toto.txt
grep -E ba+b toto.txt
grep -E b.b toto.txt
grep -E b(a|c)?b$ toto.txt
grep -E ^ba+b toto.txt
```

Note : `grep` contient aussi des références arrières, ce qui n'est pas possible à simuler dans le langage traditionnel des expressions régulières (et pour cause : on obtient ainsi des langages qui ne sont pas rationnels)

Chapitre 2

AUTOMATES FINIS DÉTERMINISTES

Un automate fini sur l'alphabet A est représenté par un graphe avec un nombre fini de sommets, qu'on appelle *états*. De chaque état u, et pour chaque lettre $a \in A$, part exactement un arc, qu'on appelle *transition*, vers un autre état.

Un état particulier est identifié comme état *initial*, et certains sommets sont identifiés comme sommets *finaux* (ou *finals*).

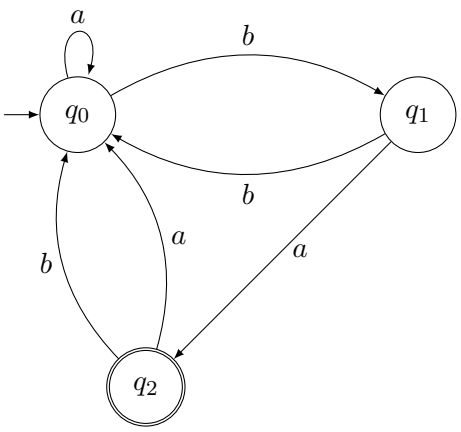


FIGURE 2.1 – Un premier exemple d'automate déterministe sur l'alphabet $A = \{a, b\}$.

Un exemple est donné à la figure 2.1. Dans les dessins, on représentera toujours l'état initial (ici q_0) avec une flèche entrante et les états finaux seront entourés (ici q_2). Un état initial peut aussi être final. À noter que, contrairement au cas des graphes vu en cours de modélisation ou d'algorithmique, on s'autorise ici à avoir des boucles sur les états (par exemple la boucle de q_0 à q_0 étiquetée par a), et même à avoir deux arcs parallèles, si tant est qu'ils portent des étiquettes différentes (comme les deux arcs de q_2 à q_0).

À tout automate fini $\mathcal{A}$ est associé un langage $L(\mathcal{A})$. Pour savoir si un mot est dans le langage $L(\mathcal{A})$ (on dit aussi que le mot est *accepté* par l'automate), on procède ainsi : on part de l'état initial, on suit les transitions correspondant aux lettres du mot, et on regarde où on arrive. Si le dernier état est acceptant, le mot est accepté (et donc dans le langage), sinon il est refusé (et donc pas dans le langage).

Toujours sur l'exemple de la figure 2.1, regardons ce qui se passe sur le mot $abaa$. Partant de q_0, on reste en q_0 après la lecture du premier a. On se déplace ensuite en q_1 après la lecture du premier b. La lecture du deuxième a nous amène en q_2 et enfin la lecture du troisième a nous amène en q_0, qui n'est pas acceptant. Donc le mot $abaa$ n'est pas dans le langage.

A contrario, regardons ce qui se passe pour le mot $bababa$. On passe successivement de q_0 à q_1 puis q_2, q_0, q_0, q_1 et enfin q_2. Comme on termine sur un état acceptant, le mot $bababa$ est accepté.

Pour en dire plus, nous allons bientôt introduire la définition précise d'un automate fini déterministe. Mais avant cela, donnons les principaux intérêts du concept.

— Il est très facile, et très rapide, de savoir si un mot est accepté par un automate déterministe. Plusieurs programmes correspondant à l'automate de la figure 2.1 sont représentés à la figure 2.2 (on rappelle qu'une chaîne de caractères, en C, est terminée par le caractère '\0'). On remarque en particulier que, si on s'autorise les goto (c'est mal), il n'est même pas nécessaire d'utiliser une variable dans le programme ! On dit quelquefois que les langages acceptés par automates finis sont sans mémoire.

— Les automates finis sont faciles à manipuler. On en verra des exemples dans ce cours et également dans le prochain cours.

— On peut représenter beaucoup de langages de cette façon. En particulier, tous les langages rationnels (vus dans le chapitre précédent) sont représentables avec des automates finis. On proposera une démonstration de ce résultat au travers des deux chapitres suivants.

2.1 Définitions

On donne maintenant une définition formelle des automates finis déterministes, avec laquelle on travaillera dans la suite.

Définition 2.1

Un automate fini $\mathcal{A}$ sur un alphabet A est la donnée :

— D'un ensemble fini Q, appelé ensemble des états [1]. Un état particulier, souvent noté q_0 est privilégié dans l'ensemble Q, on l'appelle état initial.

— D'un ensemble d'états finaux $F \subseteq Q$.

— D'une fonction de transition $\delta : Q \times A \to Q$. $\delta(q, a)$ est l'état où se trouve l'automate lorsqu'il lit, partant de l'état q, la lettre a.

Si la fonction de transition est totale (est une application) alors l'automate en plus d'être déterministe est dit complet. Les automates présentés dans ce chapitre sont tous complets.

♦ **Exemple**

On reprend l'exemple de la figure 2.1, pour lequel l'alphabet est $A = \{a, b\}$.

L'ensemble d'états est $Q = \{q_0, q_1, q_2\}$. L'état initial est l'état q_0. L'ensemble des états finaux est uniquement composé de l'état q_2 : $F = \{q_2\}$.

La fonction de transition δ est représentée par le tableau suivant :

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_0	q_0

En particulier $\delta(q_1, a) = q_2$.

1. La raison pour laquelle l'ensemble des états est appelé Q, et pourquoi les états sont notés $q_0, q_1 \dots q_n$, n'est pas très claire, et date sans doute du célèbre article de Turing de 1936. Quoiqu'il en soit, cette notation est restée. On utilise quelquefois S (comme *state*) à la place de Q. On le fera en particulier dans la définition suivante.

```
int  accept(char* w){
  int q;
  for(q = 0; *w != '\0'; w++)
  {
    if ((q == 0) && (*w == 'a'))
      q = 0;
    else if ((q == 0) && (*w == 'b'))
      q = 1;
    else if ((q == 1) && (*w == 'a'))
      q = 2;
    else if ((q == 1) && (*w == 'b'))
      q = 0;
    else /* q = 2 */
      q = 0;
  }
  return (q == 2);
}
```

```
int  accept(char* w){
  static int t[3][2] = {{0,1},{2,0},{0,0}};
  int q = 0;
  while (*w != '\0')
    q = t[q][*w++ - 'a'];
  return (q == 2);
}
```

```
int  accept(char* w){
  q0:
      if (*w == '\0')
        return 0;
      if (*w == 'a')
         { w++; goto q0; }
      if (*w == 'b')
         { w++; goto q1; }
  q1:
      if (*w == '\0')
        return 0;
      if (*w == 'a')
         { w++; goto q2; }
      if (*w == 'b')
         { w++; goto q0; }
  q2:
      if (*w == '\0')
        return 1;
      w++;
      goto q0;
}
```

FIGURE 2.2 – Trois programmes pour tester si un mot est accepté par un automate.

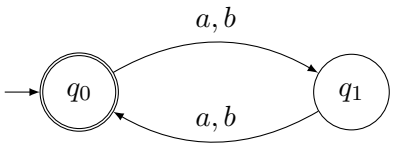


FIGURE 2.3 – Un automate qui reconnaît les mots de longueur paire.

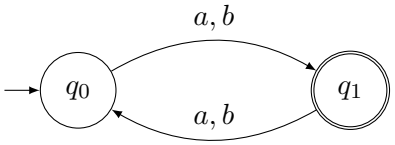


FIGURE 2.4 – Un automate qui reconnaît les mots de longueur impaire.

Définition 2.2

Soit $\mathcal{A}$ un automate sur l'alphabet A et w un mot sur l'alphabet A. On note $w = w_1 w_2 \dots w_n$.

Le mot w est accepté par l'automate $\mathcal{A}$ s'il existe des états $s_0, s_1, \dots s_n$ tels que :

— $s_0 = q_0$, l'état initial de l'automate A (on part de l'état initial) ;

— pour tout i, $s_i = \delta(s_{i-1}, w_i)$ (on suit les transitions) ;

— $s_n \in F$ (on arrive sur un état final).

On note $L(\mathcal{A})$ l'ensemble des mots acceptés par l'automate $\mathcal{A}$.

Tous les langages qu'on peut obtenir de cette façon sont appelés des langages reconnaissables.

2.2 Quelques exemples

Avant de passer aux propriétés des automates, donnons quelques exemples, sur l'alphabet $A = \{a, b\}$ ou sur l'alphabet $A = \{a, b, c\}$.

Les figures 2.3 et 2.4 représentent un automate qui reconnaît respectivement les mots de longueur paire et impaire. Intuitivement un mot arrive dans l'état q_0 du premier automate s'il est de longueur paire, et dans l'état q_1 du premier automate s'il est de longueur impaire.

La figure 2.5 représente un automate qui reconnaît les mots ayant au moins une occurrence de la lettre b. Intuitivement, on reste sur l'état q_0 tant qu'on a pas vu de b. Une fois qu'on a lu un b, on est content et on reste sur l'état q_1 quoi qu'il arrive.

La figure 2.6 représente un automate qui reconnaît les mots n'ayant pas de b. On a besoin de l'état q_1 pour dire ce qui arrive lorsqu'on lit un b, mais c'est un état particulier, qui est un état refusant et dont on ne peut pas sortir. On appelle souvent ces états des états « puits ». Il arrive qu'on décide de ne pas les représenter sur le dessin.

Enfin, la figure 2.7 représente un automate qui reconnaît les mots qui finissent par un b.

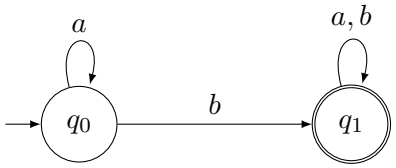


FIGURE 2.5 – Un automate qui reconnaît les mots ayant au moins un b.

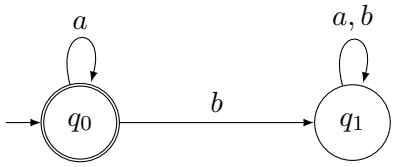


FIGURE 2.6 – Un automate qui reconnaît les mots n'ayant pas de b.

2.3 Manipulation d'automates

Comme indiqué précédemment, il est facile de manipuler des automates, pour produire, partant d'automates correspondant à des langages $L_1, L_2, \ldots$, des automates pour des langages plus compliqués s'exprimant à partir des langages $L_1, L_2, \ldots$.

La première opération a été illustrée par les exemples. Qu'obtient-on quand on échange états acceptants et non acceptants ?

Proposition 2.1

> Soit $\mathcal{A}$ un automate déterministe complet qui reconnaît un langage L. Si on échange états acceptants et refusants de l'automate $\mathcal{A}$, on obtient un automate qui reconnaît $\bar{L}$, le complémentaire de L.

On remarque bien sur les exemples que l'automate de la figure 2.4, qui reconnaît les mots de longueur impaire, est obtenu à partir de l'automate de la figure 2.3 en échangeant les états acceptants et refusants.

Attention, si on échange états acceptants et refusants sur l'automate de la figure 2.7, qui reconnaît les mots qui finissent par la lettre b, l'automate résultant n'est *pas* celui qui reconnaît les mots qui finissent par la lettre a, mais celui qui reconnaît les mots qui finissent par la lettre a, ou qui ont 0 lettre.

Les deux opérations suivantes utilisent la même idée, et consistent à combiner deux automates $\mathcal{A}_1$

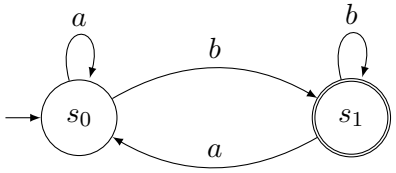


FIGURE 2.7 – Un automate qui reconnaît les mots finissant par b.

et $\mathcal{A}_2$ reconnaissant deux langages L_1 et L_2.

Supposons qu'on veuille savoir si un mot est de longueur paire et finit par b. On a en figure 2.3 un automate pour les mots de longueur paire, et en figure 2.7 celui qui reconnaît les mots qui finissent par b. Un mot vérifie les deux à la fois si, dans le premier automate, on atteint l'état final, et si, dans le deuxième automate, on atteint également l'état final. Il faut donc combiner les deux automates.

Définition 2.3

Le produit (cartésien) de deux automates $\mathcal{A}^1$ et $\mathcal{A}^2$ utilisant le même alphabet A est l'automate $\mathcal{A}$ défini de la façon suivante :

- $Q = Q^1 \times Q^2$: les états de $\mathcal{A}$ sont des couples d'états, l'un venant de $\mathcal{A}^1$, l'autre venant de $\mathcal{A}^2$.
- L'état initial est le couple (q_0^1, q_0^2) de l'état initial du premier automate et de l'état initial du second automate.
- $F = F^1 \times F^2$: les états acceptants sont les couples composés d'un état acceptant pour le premier automate et d'un état acceptant pour le deuxième.
- δ est définie par $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$: pour savoir dans quel état on se trouve si on part de (q^1, q^2) et qu'on lit la lettre a, on cherche quel état on obtient en partant de q^1 et en lisant un a, et quel état on obtient en partant de q^2 et en lisant un a.

♦ **Exemple**

Revenons sur l'exemple des mots de longueur paire qui finissent par b.

On a un automate pour les mots de longueur paire, avec deux états q_0, q_1, et un automate pour les mots qui finissent par b, également avec deux états s_0, s_1. On obtient donc $2 \times 2 = 4$ états : $(q_0, s_0), (q_0, s_1), (q_1, s_0)$ et (q_1, s_1).

L'état initial est l'état (q_0, s_0). Les états finaux sont les couples où les deux états sont finaux, donc ici uniquement (q_0, s_1).

Examinons sur un exemple comment faire une transition. Supposons être en (q_1, s_0) et qu'on lit un a. Sur le premier automate, partant de q_1, on arrive en q_0. Sur le deuxième, partant de s_0, on reste en s_0. Conclusion : de l'état (q_1, s_0), en lisant un a, on arrive sur l'état (q_0, s_0). Toutes les autres transitions s'obtiennent de la même façon. Le résultat est présenté sur la figure 2.8.

Théorème 2.2

L'automate produit défini précédemment représente l'intersection des langages L_1 et L_2 correspondant aux automates $\mathcal{A}^1$ et $\mathcal{A}^2$.

On obtient l'automate correspondant à l'union des langages L_1 et L_2 par la même construction, en prenant comme états finaux les couples (q^1, q^2) où l'un des états (au moins) est final.

La preuve de ce résultat est élémentaire et ne sera pas mentionnée. On trouvera un exemple en figure 2.9. À noter qu'on obtient par construction *un* automate qui reconnaît $L_1 \cap L_2$ (ou $L_1 \cup L_2$), mais pas forcément le plus simple. Si on reprend les automates obtenus en figure 2.7 et 2.5 et qu'on en fait le produit, on obtient l'automate de la figure 2.10 qui reconnaît donc les mots qui contiennent un b et qui finissent par b, c'est-à-dire les mots qui finissent par b. L'automate qu'on obtient est pourtant plus compliqué que celui de la figure 2.7. Il contient 4 états, dont un état qui visiblement ne sert à rien (il n'est pas possible d'arriver dans l'état (q_0, q_1)). Même en supprimant cet état inutile, on obtient un automate à 3 états, alors qu'on a vu précédemment qu'on pouvait le faire avec 2 états seulement.

Comme l'exemple l'indique, on peut produire plusieurs automates, certains plus compliqués que les autres, pour le même langage. Nous verrons dans un cours ultérieur comment simplifier ces auto-

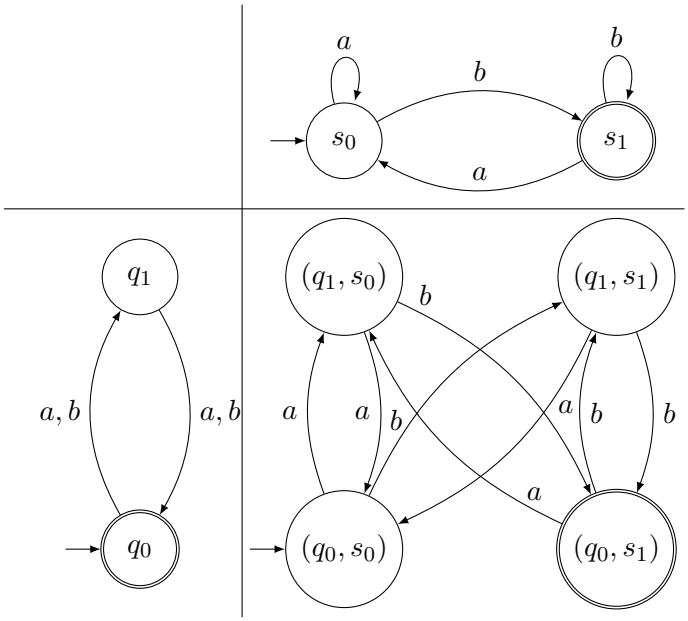


FIGURE 2.8 – Produit de l'automate qui reconnaît les mots qui finissent par b (en haut) et les mots de longueur paire (à gauche). On obtient un automate qui reconnaît les mots de longueur paire qui finissent par b.

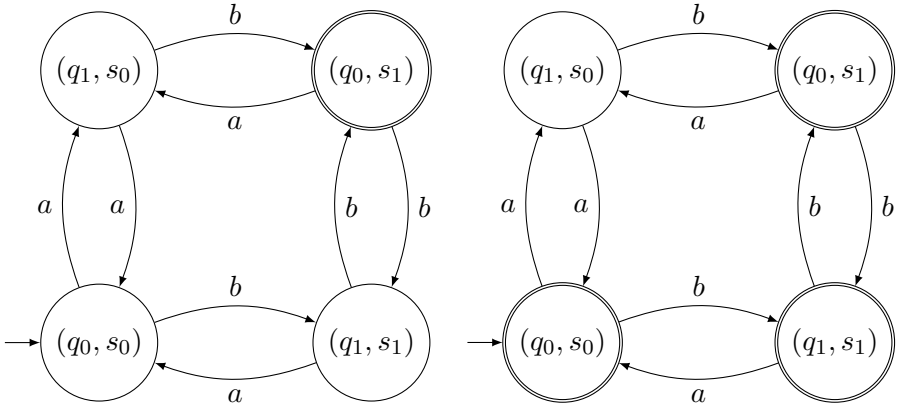


FIGURE 2.9 – Automate qui reconnaît les mots de longueur paire qui finissent par b et automate qui reconnaît les mots qui sont soit de longueur paire, soit qui finissent par b.

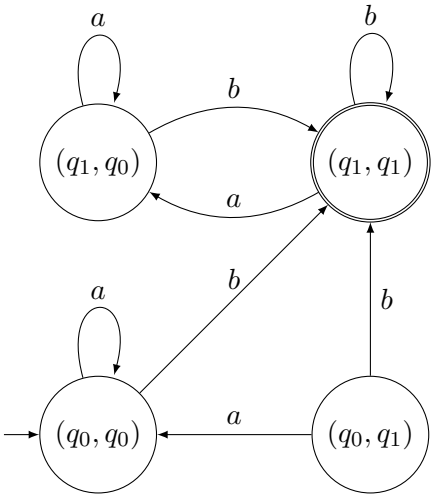


FIGURE 2.10 – Automate qui reconnaît les mots qui contiennent un b et qui finissent par b. Autrement dit, les mots qui finissent par b.

mates et obtenir, en un sens, l'automate le plus petit possible pour un langage donné.

2.4 Des automates aux regexp

Comme indiqué dans ce cours, tout ce qui est faisable avec des automates est faisable avec des expressions régulières. Nous allons le prouver dans cette partie.

Théorème 2.3

Tout langage reconnu par un automate est un langage rationnel (donné par une expression régulière).

L'idée est la suivante. D'abord nous notons L_q le langage des mots acceptés par l'automate si jamais q était l'état initial.

Supposons qu'en partant de l'état q, on peut soit lire un a et arriver dans l'état q_1, soit lire un b et arriver dans l'état q_2.

Pour un mot accepté partant de l'état q, il y a donc deux cas :

— Soit il commence par a. Dans ce cas, pour qu'il soit accepté, il faut que, partant de q_1, le mot obtenu en supprimant le premier a soit accepté.

— Soit il commence par b. Dans ce cas, pour qu'il soit accepté, il faut que, partant de q_2, le mot obtenu en supprimant le premier b soit accepté.

L'équation

$$L_q = aL_{q_1} + bL_{q_2}$$

traduit exactement cette condition.

Il y a cependant un troisième cas qui peut apparaître, si l'état q est acceptant. Dans ce cas, un mot accepté partant de l'état q peut également être le mot vide. Dans ce cas l'équation devient :

$$L_q = aL_{q_1} + bL_{q_2} + \epsilon$$

Plus généralement,

Algorithme 2.4 *Expression régulière à partir d'un automate*

Étant donné un automate $\mathcal{A}$, on obtient une expression régulière pour le langage L de la façon suivante :

- À chaque état q de l'automate, on associe un langage L_q, correspondant à l'ensemble des mots acceptés si jamais q était l'état de départ.
- Pour chaque état q de l'automate, on écrit une équation. Soit $q_1, q_2, \dots q_n$ les états obtenus en partant de q et en lisant respectivement les lettres $a_1, a_2 \dots a_n$. Alors l'équation prend la forme

$$L_q = a_1 L_{q_1} + a_2 L_{q_2} + \dots a_n L_{q_n}$$

si q n'est pas acceptant ou

$$L_q = a_1 L_{q_1} + a_2 L_{q_2} + \dots a_n L_{q_n} + \epsilon$$

sinon.
- On résout le système d'équations en utilisant le lemme d'Arden (lemme 1.3) vu dans le cours précédent.

Pour montrer que l'algorithme fonctionne, il reste à vérifier que toutes les équations formant le système sont correctes, et que le lemme d'Arden permet bien de les résoudre. Nous ne rentrerons pas ici dans une vraie démonstration.

♦ **Exemple**

Reprenons l'exemple des mots ayant au moins un b, dont l'automate est représente en figure 2.5. On définit donc deux langages L_0 et L_1 correspondant aux états q_0 et q_1 (on appelle les langages L_0 et L_1 plutôt que L_{q_0} et L_{q_1} pour ne pas alourdir les équations). Les équations obtenues sont :

$$L_0 = aL_0 + bL_1$$
$$L_1 = (a+b)L_1 + \epsilon$$

Comme on cherche L_0, on va commencer par éliminer L_1. Pour cela, on résout la deuxième équation :

$$L_1 = (a+b)^\star \epsilon = (a+b)^\star$$

et on remplace dans la première équation :

$$L_0 = aL_0 + b(a+b)^\star$$

et on résout la première équation

$$L_0 = a^\star b(a+b)^\star$$

♦ **Exemple**

Reprenons l'exemple des mots qui finissent par b, dont l'automate est représente en figure 2.7 On définit donc deux langages L_0 et L_1 correspondant aux états q_0 et q_1. Les équations obtenues sont :

$$L_0 = aL_0 + bL_1$$

$$L_1 = aL_0 + bL_1 + \epsilon$$

Comme on cherche L_0, on va commencer par éliminer L_1. Pour cela, on résout la deuxième équation :

$$L_1 = b^\star(aL_0 + \epsilon)$$

et on remplace dans la première équation :

$$L_0 = aL_0 + bb^\star(aL_0 + \epsilon)$$

$$L_0 = (a + bb^\star a)L_0 + bb^\star$$

Et on résout :

$$L_0 = (a + bb^\star a)^\star bb^\star$$

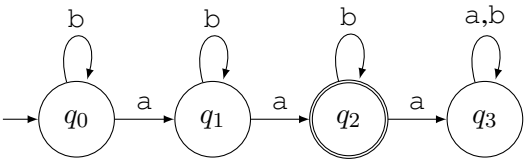
A noter que l'expression que nous obtenons n'est pas la plus simple possible. En effet, on peut aussi écrire

$$L_0 = (a + b)^\star b$$

Exercices

(2 - 1) *(Automate mystère)*

Expliquer en français ce que reconnaît cet automate.



(2 - 2) *(Constructions d'automates)*

Penser à tester vos automates !

Q 1) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui ont un nombre pair de $\mathtt{a}$.

Q 2) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui ont un nombre de $\mathtt{a}$ multiple de 3 (0 est multiple de 3).

Q 3) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui commencent et finissent par la même lettre, et qui ont au moins deux lettres.

Q 4) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ qui contiennent le mot $\mathtt{ab}$.

Q 5) Même question avec $\mathtt{aab}$.

(2 - 3) *(Produit)*

Q 1) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui ont un nombre pair de $\mathtt{a}$ et un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui ont un nombre pair de $\mathtt{b}$. Combiner les deux automates pour obtenir un automate qui reconnaît les mots qui ont un nombre pair de $\mathtt{a}$ et un nombre pair de $\mathtt{b}$.

Q 2) Construire un automate qui reconnaît les mots qui ont un nombre pair de $\mathtt{a}$ mais pas un nombre pair de $\mathtt{b}$.

(2 - 4) *(Encore un automate)*

Q 1) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui commencent par $\mathtt{aaa}$.

Q 2) On cherche maintenant à obtenir un automate qui reconnaît les mots qui terminent par $\mathtt{aaa}$. Un étudiant propose de prendre le même que le précédent, d'inverser le sens de toutes les flèches, et d'inverser état initial et état final. Pourquoi ce n'est pas possible ?

Q 3) Construire un automate qui reconnaît les mots sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$ qui finissent par $\mathtt{aaa}$.

(2 - 5) *(regexp)*

Q 1) Calculer des regexp pour les langages reconnus par les deux premiers automates de l'exercice **(2 - 2)** .

Q 2) Calculer des regexp pour le langage reconnu par l'automate de l'exercice **(2 - 1)** .

(2 - 6) *(décimal et binaire)*

On se place dans cet exercice sur l'alphabet $\{0, 1, 2, \ldots, 9\}$.

Q 1) Donner un automate qui reconnait l'ensemble des mots qui correspondent à un nombre multiple de 10.

Q 2) Donner un automate qui reconnait l'ensemble des mots qui correspondent à un nombre multiple de 2.

On va maintenant construire un automate qui reconnait l'ensemble des mots qui correspondent à un nombre multiple de 3.

On rappelle quelques règles d'arithmétique modulaire. Si on veut calculer un produit (ou une somme) modulo 3, on peut d'abord simplifier les deux termes avant de calculer le modulo. Par exemple, $32 \mod 3 = 2$, $52 \mod 3 = 1$, donc $(32 \times 52) \mod 3 = (2 \times 1) \mod 3 = 2$.

De la même façon, calculons 45527 modulo 3 :

$$\begin{array}{rcl}
4 \mod 3 & = & 1 \\
45 \mod 3 & = & (4 \times 10) + 5 \mod 3 \quad = \quad (1 \times 10) + 5 \mod 3 = 0 \\
455 \mod 3 & = & (45 \times 10) + 5 \mod 3 \quad = \quad (0 \times 10) + 5 \mod 3 = 2 \\
4552 \mod 3 & = & (455 \times 10) + 2 \mod 3 \quad = \quad (2 \times 10) + 2 \mod 3 = 1 \\
45527 \mod 3 & = & (4552 \times 10) + 7 \mod 3 \quad = \quad (1 \times 10) + 7 \mod 3 = 2
\end{array}$$

(On aurait aussi pu simplifier le $\times 10$ en $\times 1$ modulo 3).

Q 3) En utilisant l'explication précédente, construire un automate qui reconnaît l'ensemble des mots qui correspondent à un nombre multiple de 3. L'automate aura 3 états, numérotés 0, 1 et 2. Quand on lit le nombre n partant de l'état initial, on doit arriver sur l'état correspondant à $n \mod 3$. En particulier sur l'entrée 45527, on doit passer successivement par les états $0, 1, 0, 2, 1, 2$.

Q 4) En procédant de la même manière, construire l'automate qui reconnaît l'ensemble des mots qui correspondent à un nombre multiple de 5. Expliquez comment faire la même chose avec moins d'états.

On se place maintenant sur l'alphabet $\{0, 1\}$.

Q 5) Donner un automate qui reconnait l'ensemble des mots qui correspondent à un nombre écrit en binaire et qui est multiple de 2.

Q 6) Donner un automate qui reconnait l'ensemble des mots qui correspondent à un nombre écrit en binaire et qui est multiple de 4.

Q 7) Donner un automate qui reconnait l'ensemble des mots qui correspondent à un nombre écrit en binaire et qui est multiple de 3.

Q 8) Donner un automate qui reconnait l'ensemble des mots qui correspondent à un nombre écrit en binaire et qui est une puissance de 2.

Chapitre 3

AUTOMATES FINIS NON DÉTERMINISTES

À la différence des automates déterministes, un automate non déterministe peut avoir, partant d'un état donné, plusieurs transitions portant la même lettre.

Un premier exemple est donné à la figure 3.1. De l'état q_0, quand on lit un b, on peut décider soit de rester sur place, soit d'aller en q_1. De même, quand on lit un a partant de l'état q_2, on peut décider d'aller en q_0 ou en q_1.

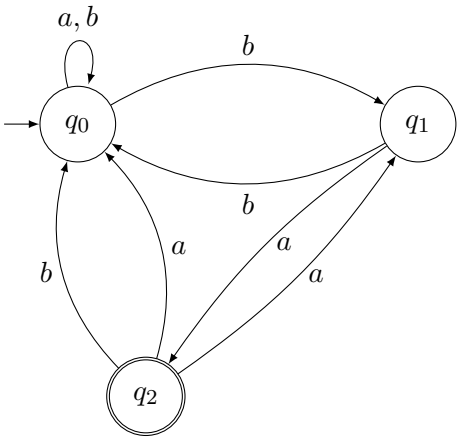


FIGURE 3.1 – Un premier exemple d'automate non déterministe sur l'alphabet $A = \{a, b\}$.

Dans un automate non déterministe, il est plus difficile de savoir si un mot est accepté : en effet, il faut regarder *tous* les chemins possibles, et voir si l'*un d'entre eux* aboutit à un état acceptant.

Prenons l'exemple du mot $abaa$. Il y a 3 chemins possibles :

— $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0$
— $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_1$
— $q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_0$

Aucun d'entre eux ne termine par un état acceptant, donc le mot $abaa$ n'est pas accepté par l'automate.

Prenons le mot $baaa$. Il y a aussi 3 chemins possibles :

— $q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0$
— $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_1 \xrightarrow{a} q_2$
— $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_0 \xrightarrow{a} q_0$

L'un d'entre eux termine sur l'état acceptant q_2, donc le mot $baaa$ est accepté.

Ce simple exemple montre donc que le traitement des automates non déterministes semble plus compliqué que celui des automates déterministes. La situation est en réalité un peu différente. D'abord, on verra rapidement qu'il est bien plus simple de concevoir et réaliser des tâches avec les automates

finis non déterministes. On pourra ainsi comparer l'automate non déterministe de la figure 3.2, qu'on comprend assez facilement, à celui de la figure 3.3 qui est bien plus difficile à concevoir.

De plus, les automates non déterministes peuvent être transformés en automates déterministes, de sorte qu'on peut tout de même vérifier facilement qu'un mot est accepté. On verra cependant que cette transformation a un prix : la taille de l'automate explose.

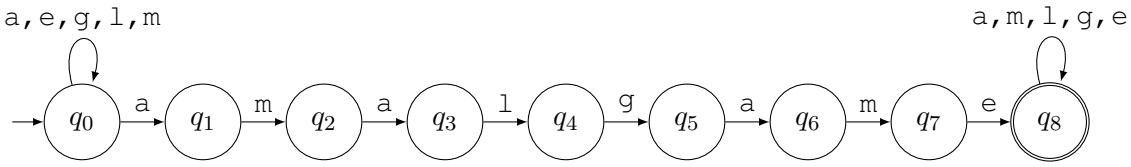


FIGURE 3.2 – Un automate non déterministe qui reconnaît l'ensemble des mots w sur l'alphabet $\{\texttt{a},\texttt{e},\texttt{l},\texttt{g},\texttt{m}\}$ qui contiennent le sous-mot `amalgame`.

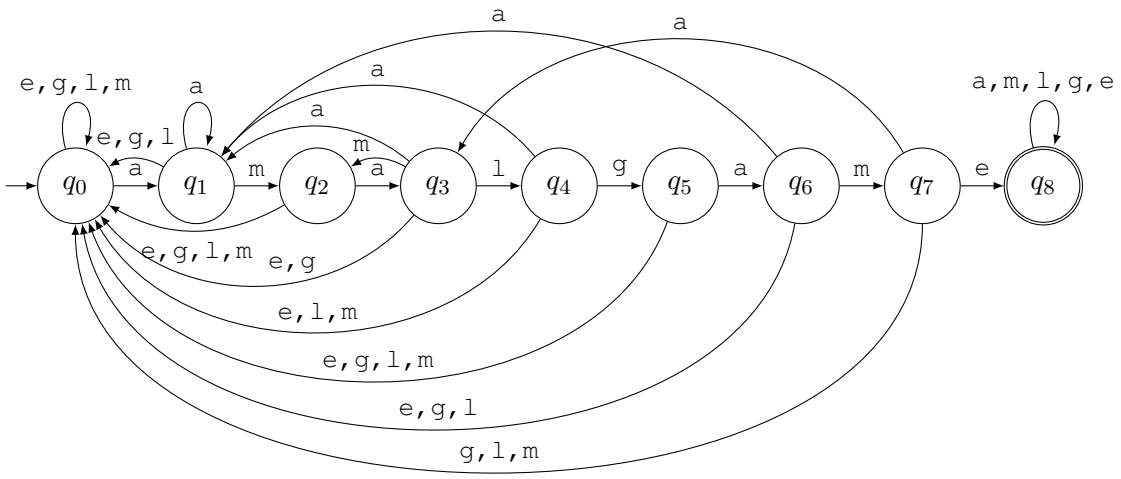


FIGURE 3.3 – Un automate déterministe qui reconnaît l'ensemble des mots w sur l'alphabet $\{\texttt{a},\texttt{e},\texttt{l},\texttt{g},\texttt{m}\}$ qui contiennent le sous-mot `amalgame`.

3.1 Définitions

Définition 3.1

Un automate fini non déterministe $\mathcal{A}$ sur un alphabet A est la donnée :

- D'un ensemble fini Q, appelé ensemble des états.
- D'un ensemble d'états initiaux $I \subseteq Q$.
- D'un ensemble d'états finaux $F \subseteq Q$.
- D'une fonction de transition $\delta \in Q \times A \to \mathcal{P}(Q)$. $\delta(q, a)$ est l'ensemble des états où peut se trouver l'automate lorsqu'il lit, partant de l'état q, la lettre a.

On observe donc deux modifications essentielles par rapport aux automates déterministes :

- on peut avoir plus d'un état initial ;
- on a désormais une fonction de transition dans $\mathcal{P}(Q)$, donc plusieurs transitions possibles partant du même état et étiquetées avec la même lettre.

♦ **Exemple**

On reprend l'exemple de la figure 3.1, pour lequel l'alphabet est $A = \{a, b\}$.

L'ensemble d'états est $Q = \{q_0, q_1, q_2\}$. L'ensemble des états initiaux est $I = \{q_0\}$. L'ensemble des états finaux est $F = \{q_2\}$.

La fonction de transition est représentée par le tableau suivant :

	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_0\}$
q_2	$\{q_0, q_1\}$	$\{q_0\}$

Si le contexte est clair, on pourra ne pas représenter les symboles « { » et « } » définissant des ensembles.

La définition de l'acceptation est quasi-identique à celle pour les automates déterministes :

Définition 3.2

Soit $\mathcal{A}$ un automate sur l'alphabet A et w un mot sur l'alphabet A. On note $w = w_1 w_2 \dots w_n$.

Le mot w est accepté par l'automate $\mathcal{A}$ s'il existe des états $s_0, s_1, \dots s_n$ tels que :

- $s_0 \in I$ (on part d'un des états initiaux) ;
- pour tout i, $s_i \in \delta(s_{i-1}, w_i)$ (on suit l'une des transitions à chaque étape) ;
- $s_n \in F$ (on arrive sur un état final).

On note $L(\mathcal{A})$ l'ensemble des mots acceptés par l'automate $\mathcal{A}$.

Tous les langages qu'on peut obtenir de cette façon sont appelés des langages reconnaissables par automates non déterministes.

3.2 Constructions d'automates

Nous allons maintenant montrer qu'il est facile, et en tout cas plus facile que sur les automates déterministes, de réaliser certaines opérations sur les automates non déterministes.

L'opération la plus simple est l'union :

Proposition 3.1

Soit $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates non déterministes, correspondant aux langages L_1 et L_2. On obtient un automate non déterministe $\mathcal{A}_3$ qui reconnaît $L_1 \cup L_2$ en mettant les deux automates côte à côte :

- $Q_3 = Q_1 \cup Q_2$: les états de $\mathcal{A}_3$ sont les états de $\mathcal{A}_1$ et de $\mathcal{A}_2$ (qu'on peut supposer distincts).
- $I_3 = I_1 \cup I_2$: les états initiaux de $\mathcal{A}_3$ sont les états initiaux de $\mathcal{A}_1$ et les états initiaux de $\mathcal{A}_2$.
- $F_3 = F_1 \cup F_2$: les états finaux de $\mathcal{A}_3$ sont les états finaux de $\mathcal{A}_1$ et les états finaux de $\mathcal{A}_2$.
- $\delta_3 = \delta_1 \cup \delta_2$: toutes les transitions sont conservées.

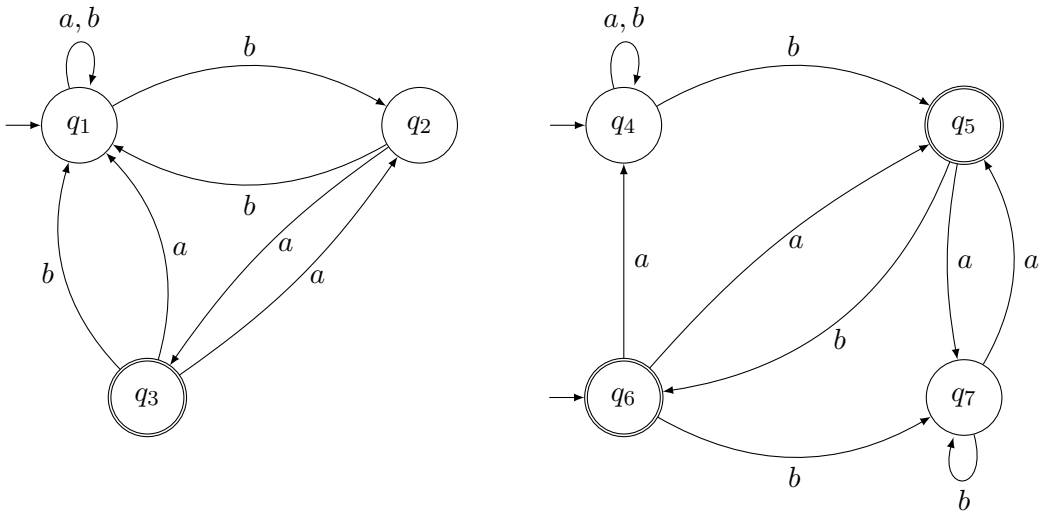


FIGURE 3.4 – Deux automates $\mathcal{A}_1$ (à gauche) et $\mathcal{A}_2$ (à droite), qui reconnaissent deux langages L_1 et L_2.

Attention, pour pouvoir faire cette transformation, il faut commencer par vérifier que les noms des états sont différents dans les deux automates. Si ce n'est pas le cas, il faut renommer les noms dans l'un des deux automates.

♦ **Exemple**

La figure 3.4 représente deux automates. Pour obtenir l'union des deux automates, il suffit de regarder le dessin, non pas comme le dessin de deux automates, mais comme le dessin d'un seul automate, ayant 3 états initiaux et 3 états finaux.

À partir d'un automate à 3 états et d'un automate à 4 états, on obtient donc un automate à 7 états.

Nous allons maintenant passer à la construction qui permet de trouver un automate non déterministe qui fait la concaténation de deux langages, en connaissant pour chacun un automate non déterministe. Intuitivement, la transformation consiste à fusionner les états finaux du premier automate avec les états initiaux du deuxième automate. Cette méthode ne fonctionne pas toujours, et il faut définir les choses de manière un peu plus précise pour obtenir une méthode qui marche dans tous les cas.

Proposition 3.2

Soit $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates non déterministes, correspondant aux langages L_1 et L_2. On obtient un automate non déterministe $\mathcal{A}_3$ qui reconnaît L_1L_2 de la façon suivante :

- $Q_3 = Q_1 \cup Q_2$: les états de $\mathcal{A}_3$ sont les états de $\mathcal{A}_1$ et de $\mathcal{A}_2$.
- $I_3 = I_1$: les états initiaux de $\mathcal{A}_3$ sont les états initiaux de $\mathcal{A}_1$.
- Il y a deux cas pour les états finaux :
  - Soit un des états initiaux de $\mathcal{A}_2$ est aussi un état final (donc $\epsilon \in L_2$). Dans ce cas, les états finaux sont les états finaux des deux automates : $F_3 = F_2 \cup F_1$.
  - Sinon, les états finaux de $\mathcal{A}_3$ sont les états finaux de $\mathcal{A}_2$: $F_3 = F_2$.
- On conserve les transitions des deux automates, et on ajoute de plus les transitions suivantes : pour chaque état q final dans $\mathcal{A}_1$, et chaque état q' initial dans $\mathcal{A}_2$, on ajoute à l'état q toutes les transitions possibles en partant de q' (tous les états finaux de $\mathcal{A}_1$ se comportent comme des états initiaux de $\mathcal{A}_2$).

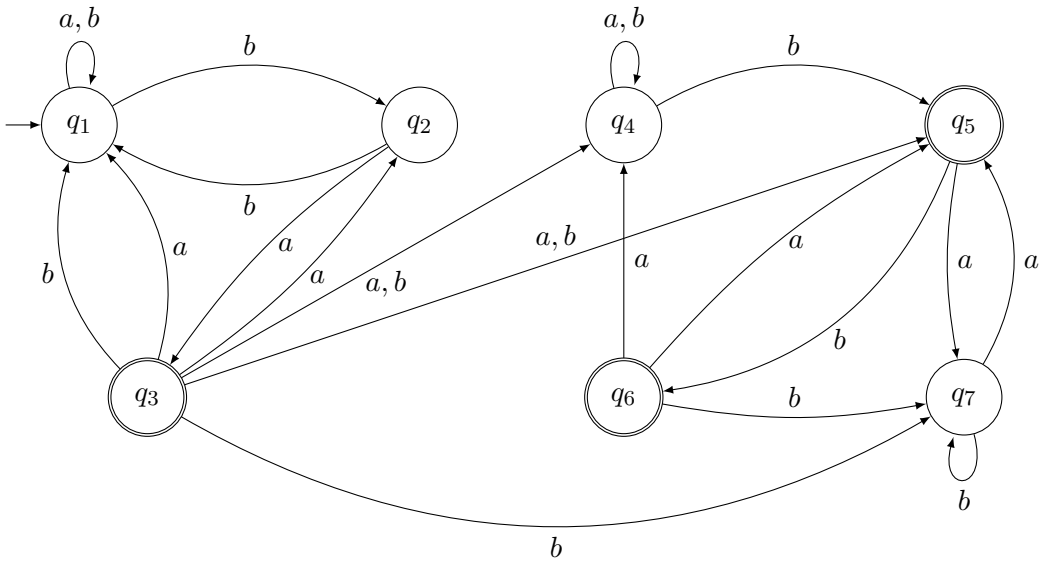


FIGURE 3.5 – Un automate qui reconnaît L_1L_2.

♦ **Exemple**

L'automate correspondant à la concaténation des langages L_1L_2 reconnus par les automates de la figure 3.4 est indiqué à la figure 3.5. Comme indiqué dans l'énoncé, les états q_4 et q_6 du deuxième automate ne sont plus des états initiaux. L'état q_3 reste un état final car un des états initiaux du deuxième automate, l'état q_6, est également un état final.

Les seules transitions ajoutées concernent l'état final de l'automate $\mathcal{A}_1$, c'est à dire q_3. On ajoute à q_3 toutes les transitions qui partent des états initiaux du deuxième automate, soit q_4 et

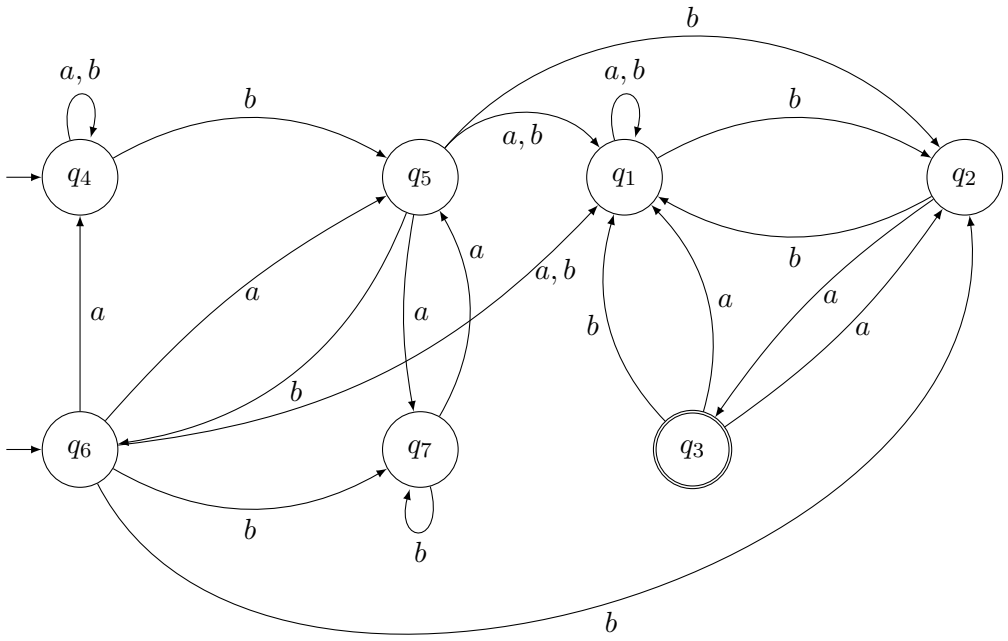


FIGURE 3.6 – Un automate qui reconnaît L_2L_1.

q_6. On ajoute donc :
— Des transitions étiquetées a, b vers l'état q_4 (correspondant aux transitions de q_4 vers q_4).
— Une transition étiquetée b vers l'état q_5 (correspondant à la transition de q_4 vers q_5).
— Une transition étiquetée a vers l'état q_4 (correspondant à la transition de q_6 vers q_4).
— Une transition étiquetée b vers l'état q_7 (correspondant à la transition de q_6 vers q_7).
— Une transition étiquetée a vers l'état q_5 (correspondant à la transition de q_6 vers q_7).

On obtient un deuxième exemple en regardant cette fois-ci l'automate correspondant à la concaténation L_2L_1, indiqué en figure 3.6. Les états initiaux sont cette fois q_4 et q_6 (et plus q_1). Comme aucun état initial de l'automate $\mathcal{A}_1$ (qui est cette fois-ci le deuxième automate) n'est aussi final, les états q_5 et q_6 ne sont plus finaux.

Toutes les transitions partant de q_1 sont répercutées vers q_5 et q_6.

Pour comprendre comment marche l'automate, regardons ce qui se passe sur le mot $baaba$.

Le mot baa est accepté par l'automate $\mathcal{A}_2$ par le chemin suivant :

$$q_4 \xrightarrow{\texttt{b}} q_5 \xrightarrow{\texttt{a}} q_7 \xrightarrow{\texttt{a}} q_5$$

et le mot ba est accepté par l'automate $\mathcal{A}_1$ par le chemin :

$$q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

Le mot $baaba$ est donc accepté par le nouvel automate. En effet, il est accepté en suivant le chemin :

$$q_4 \xrightarrow{\texttt{b}} q_5 \xrightarrow{\texttt{a}} q_7 \xrightarrow{\texttt{a}} q_5 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

Autrement dit, on concatène les deux chemins en court-circuitant l'état initial q_1 de l'automate de droite.

La dernière opération facile à réaliser est l'étoile [1] :

Proposition 3.3

Soit $\mathcal{A}_1$ un automate non déterministe, correspondant au langage L_1. On obtient un automate non déterministe $\mathcal{A}_3$ qui reconnaît $L_1^\star$ de la façon suivante :

- Les états de $\mathcal{A}_3$ sont les états de $\mathcal{A}_1$, auquel on ajoute un état initial q_0.
- L'état initial de $\mathcal{A}_3$ est q_0.
- Les états finaux de $\mathcal{A}_3$ sont les états finaux de $\mathcal{A}_1$, plus q_0.
- On conserve les transitions de l'automate $\mathcal{A}_1$. On ajoute toutes les transitions partant des états I à l'état q_0 et à tous les états finaux (les états finaux et q_0 se comportent comme des états initiaux).

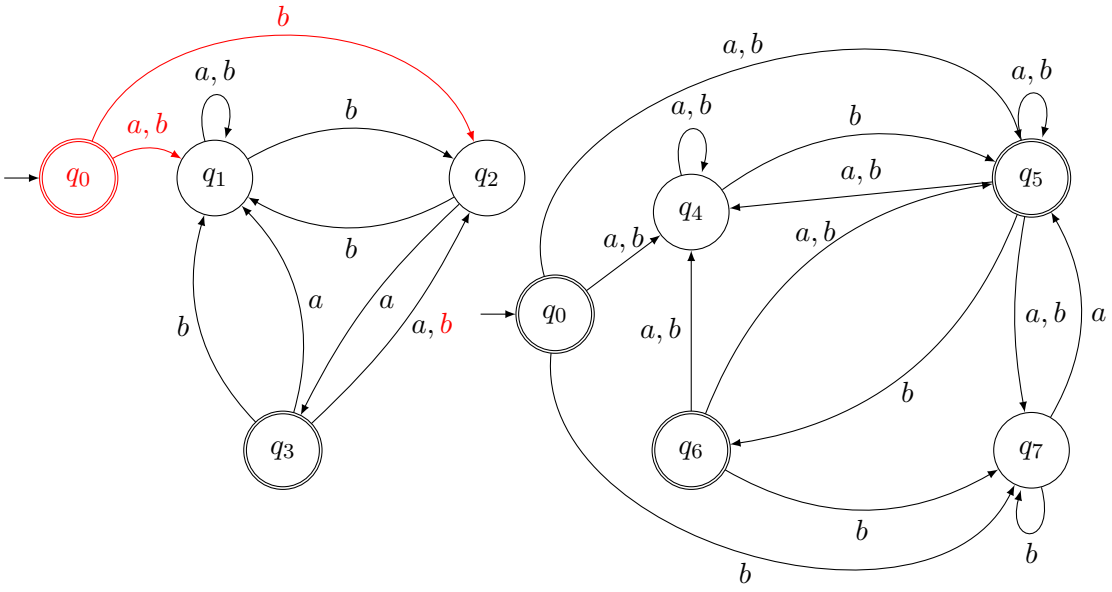


FIGURE 3.7 – Deux automates reconnaissant les langages $L_1^\star$ et $L_2^\star$.

♦ **Exemple**

La figure 3.7 représente deux automates, reconnaissant respectivement les langages L_1 et L_2.

Concentrons nous sur le premier automate. On a ajouté un état q_0, et on a ajouté, sur l'état q_0 et sur l'état q_3, toutes les transitions qui partent de l'état initial, soit de l'état q_1 : on a donc des transitions a, b qui partent de q_0 vers q_1 (correspondant aux transitions de q_1 vers q_1) et une transition b de q_0 vers q_2 (correspondant à la transition de q_1 vers q_2).

Pour plus de lisibilité, les transitions ajoutées sont indiquées en couleur.

1. Une construction alternative sans introduire l'état q_0 est de rendre final un des états initial (pour reconnaître ϵ) ne possédant pas de transition entrante (sinon trop de mots sont reconnus).

Regardons le premier automate. Dans l'automate initial, les mots aba, ba et $baaba$ sont acceptés, respectivement par les chemins

$$q_1 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

$$q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

$$q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

Dans le nouvel automate, le mot $ababaaba$ est accepté, comme le prouve le chemin :

$$q_0 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3 \xrightarrow{\texttt{a}} q_1 \xrightarrow{\texttt{b}} q_2 \xrightarrow{\texttt{a}} q_3$$

Là encore, on a court-circuité l'état initial des trois chemins.

Corollaire 3.4

Tout langage rationnel (donné par une expression régulière) est reconnu par un automate non déterministe.

Il suffit d'appliquer les 3 constructions précédentes, et en remarquant qu'on peut facilement produire des automates non déterministes pour les langages élémentaires de base.

♦ **Exemple**

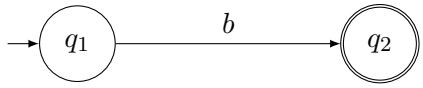
Regardons le langage

$$L = c(a + bc)^\star$$

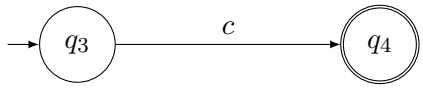
et trouvons l'automate pour ce langage. Pour cela, il faut commencer par $(a + bc)^\star$ et donc commencer par $(a + bc)$. C'est l'union de deux langages, on va commencer par bc.

C'est la concaténation de l'automate pour b et l'automate pour c.

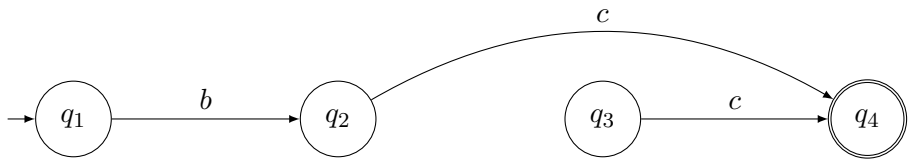
Voici l'automate pour b :



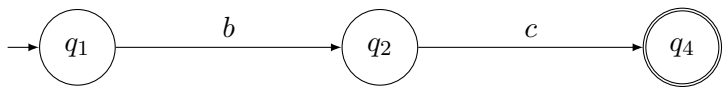
et l'automate pour c :



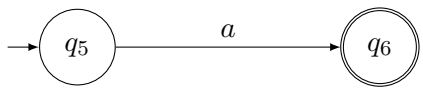
On applique l'algorithme présenté pour trouver l'automate pour bc :



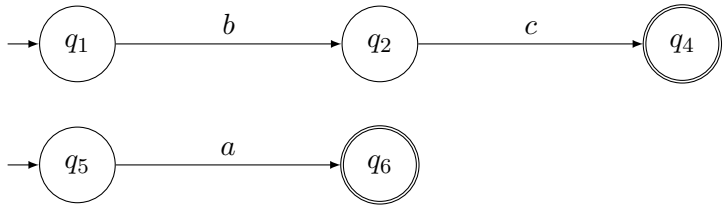
Avant d'aller plus loin, on s'aperçoit qu'on peut le simplifier : l'état q_3 ne sert à rien : en effet, il est impossible d'arriver sur cet état. On peut donc l'éliminer pour obtenir :



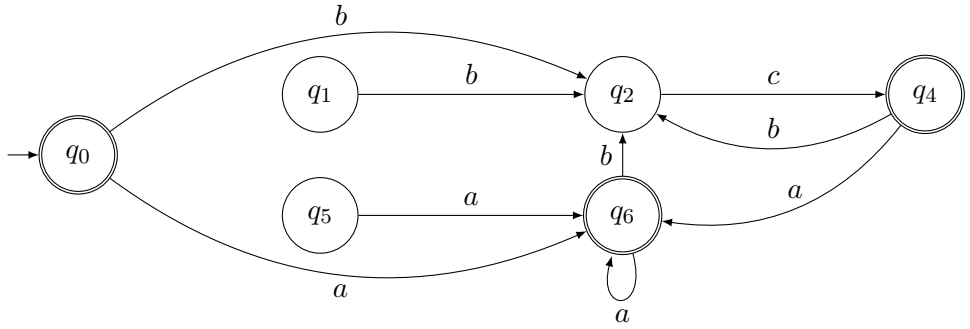
Passons à $a + bc$. L'automate pour a est :



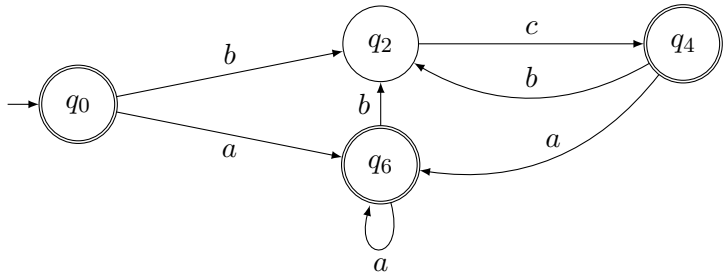
Donc l'automate pour $a + bc$ est :



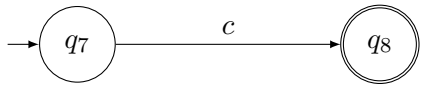
Pour obtenir $(a + bc)^\star$, on applique l'algorithme présenté précédemment :



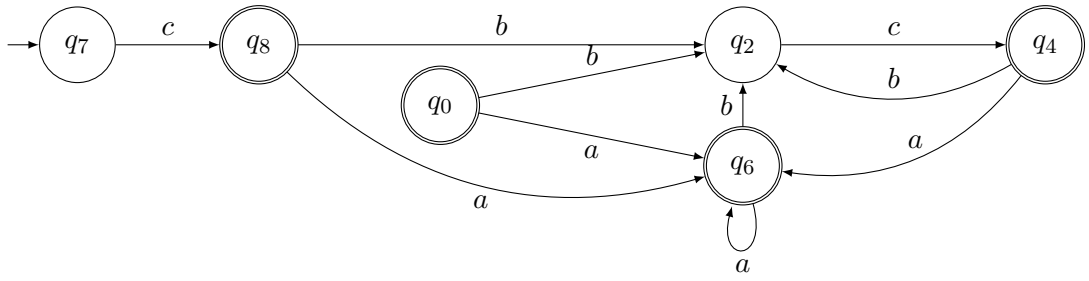
Là encore, on peut simplifier, les états q_1 et q_5 ne servent pas.



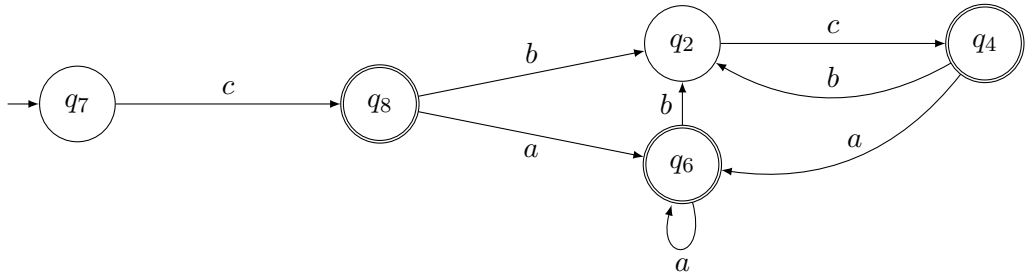
Pour finir, on veut $c(a + bc)^\star$, donc il faut prendre l'automate qui reconnaît c et celui qui reconnaît $(a + bc)^\star$. Rappelons l'automate qui reconnaît c :



On obtient finalement pour $c(a + bc)^\star$:



Qu'on peut simplifier :



Il faut noter que, même si les automates non déterministes permettent de réaliser plus facilement certaines opérations, d'autres opérations sont plus compliquées que pour les automates déterministes. Par exemple, il est plus difficile de réaliser le langage complémentaire sur un automate non déterministe. Revenons ainsi à l'automate de la figure 3.1. Si on veut obtenir le langage complémentaire, il ne suffit pas d'échanger les états finaux comme c'était le cas pour les automates déterministes. En effet, si on échange les états finaux, le mot $baaa$ sera accepté (puisqu'on peut arriver dans l'état q_0 en lisant $baaa$) alors qu'il ne devrait pas, puisqu'il est aussi accepté dans l'automate initial (puisqu'on peut arriver dans l'état q_2 en lisant $baaa$).

Les deux autres opérations (union et intersection) vues sur les automates déterministes peuvent encore fonctionner pour les automates non déterministes *mutatis mutandis*. Cependant, on préférera dans le cas de l'union la construction présentée ici qui est bien plus simple et plus « efficace » (utilise moins d'états).

3.3 Du non-déterminisme au déterminisme

Nous avons vu dans le chapitre précédent que les langages reconnus par automates finis déterministes pouvaient être représentés par des langages rationnels (donnés par expressions régulières). Nous avons vu dans ce chapitre que les expressions régulières pouvaient être représentées par des automates finis non déterministes.

Pour boucler la boucle, il reste à expliquer pourquoi les automates non déterministes peuvent être simulés par des automates déterministes.

Avant de faire cela, prenons un automate non déterministe, représenté en figure 3.8, et cherchons à savoir si le mot $abaaabba$ est reconnu. On va raisonner lettre par lettre :

— après la lecture du a, on est en q_0 ;
— après la lecture du b, on est en q_0 ou q_1 ;
— après la lecture du a, on est en q_0 ou q_2 ;
— après la lecture du a, on est en q_0 ou q_1 ;
— après la lecture du a, on est en q_0 ou q_2 ;
— après la lecture du b, on est en q_0 ou q_1 ;
— après la lecture du b, on est en q_0 ou q_1 ;
— après la lecture du a, on est en q_0 ou q_2.

Comme on le voit intuitivement, pour savoir si le mot $abaaabba$ est accepté, il n'est pas nécessaire de retenir tout le chemin, mais de savoir uniquement, à chaque étape, où on peut potentiellement se trouver.

Le procédé de déterminisation utilise cette idée.

Pour déterminiser un automate, il faut se demander quelle information on doit retenir à chaque

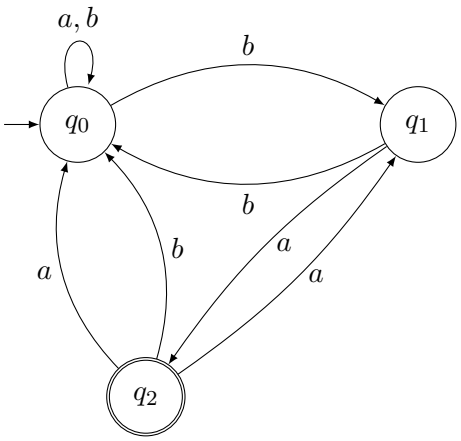


FIGURE 3.8 – Un autre exemple d'automate non déterministe sur l'alphabet $A = \{a, b\}$.

étape. La réponse est simple : il faut retenir l'*ensemble* des états où on peut éventuellement se trouver :

Proposition 3.5

Soit $\mathcal{A}_1$ un automate non déterministe.

On obtient un automate $\mathcal{A}_2$ reconnaissant le même langage de la façon suivante :

— $Q_2 = \mathcal{P}(Q_1)$: les états du nouvel automate sont les *ensembles d'états* de l'automate initial

— $q_0 = I_1$: l'état initial du nouvel automate est l'ensemble des états initiaux de l'automate précédent.

— $F_2 = \{S | S \cap F_2 \neq \emptyset\}$: les états finaux sont les ensembles dont au moins un des états est final.

— $\delta(S, a) = \cup_{s \in S} \delta(s, a)$: la fonction de transition envoie l'ensemble d'états S en lisant a vers l'ensemble des états qu'on peut atteindre partant de l'un des états de S et en lisant a.

En pratique, on peut aller plus vite, et c'est effectivement ce qu'on fait : on ne représente pas les états qu'on ne peut pas atteindre à partir de l'état initial.

Algorithme 3.3

Pour déterminiser un automate non déterministe, on calcule progressivement tous les ensembles d'états qu'on peut obtenir :

— On part avec l'ensemble S_0 des états initiaux de l'automate.

— À chaque étape, partant d'un ensemble S donné, on calcule, pour chaque lettre a, l'ensemble des états qu'on peut atteindre partant d'un état dans S et en lisant a. Si l'ensemble qu'on obtient n'a pas encore été traité, on l'ajoute à la liste des ensembles à traiter.

— Les états finaux sont les ensembles dont l'un des états est final.

♦ **Exemple**

Essayons de déterminiser l'automate de la figure 3.8.

On part avec $S_0 = \{q_0\}$.

Si on lit un a, on reste en $\{q_0\}$. Si on lit un b, on arrive en q_0 ou q_1.

On résume tout cela dans le tableau :

	a	b
$S_0 = \{q_0\}$	$\{q_0\} = S_0$	$\{q_0, q_1\} = S_1$

On passe ensuite à S_1. Si on lit un a, on arrive en q_0 ou q_2. Si on lit un b, on arrive en q_0 ou q_1 :

	a	b
$S_0 = \{q_0\}$	$\{q_0\} = S_0$	$\{q_0, q_1\} = S_1$
$S_1 = \{q_0, q_1\}$	$\{q_0, q_2\} = S_2$	$\{q_0, q_1\} = S_1$

On traite maintenant $S_2 = \{q_0, q_2\}$. Si on lit un a, on arrive en q_0 ou q_1. Si on lit un b, on arrive en q_0 ou q_1 :

	a	b
$S_0 = \{q_0\}$	$\{q_0\} = S_0$	$\{q_0, q_1\} = S_1$
$S_1 = \{q_0, q_1\}$	$\{q_0, q_2\} = S_2$	$\{q_0, q_1\} = S_1$
$S_2 = \{q_0, q_2\}$	$\{q_0, q_1\} = S_1$	$\{q_0, q_1\} = S_1$

On a tout traité, on peut s'arrêter.

Il reste à noter les états finaux. C'est ceux qui contiennent q_2, donc uniquement S_2.

L'automate résultant est décrit à la figure

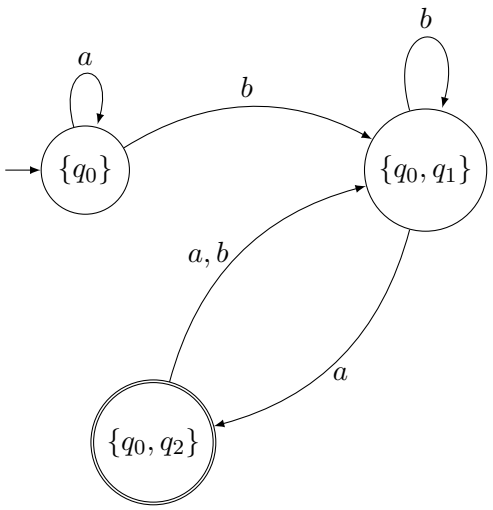


FIGURE 3.9 – L'automate déterminisé correspondant à l'automate non déterministe de la figure 3.8.

Un deuxième exemple est présenté en figure 3.10. L'automate non déterministe correspondant est en figure 3.11. L'analyse complète est décrite ci-dessous :

	a	b
$S_0 = \{q_0\}$	$\{q_0, q_2\} = S_1$	$\{q_1\} = S_2$
$S_1 = \{q_0, q_2\}$	$\{q_0, q_1, q_2\} = S_3$	$\{q_1\} = S_2$
$S_2 = \{q_1\}$	$\{q_0\} = S_0$	$\{q_2\} = S_4$
$S_3 = \{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\} = S_3$	$\{q_1, q_2\} = S_5$
$S_4 = \{q_2\}$	$\{q_0, q_1\} = S_6$	$\emptyset = S_7$
$S_5 = \{q_1, q_2\}$	$\{q_0, q_1\} = S_6$	$\{q_2\} = S_4$
$S_6 = \{q_0, q_1\}$	$\{q_0, q_2\} = S_1$	$\{q_1, q_2\} = S_5$
$S_7 = \emptyset$	$\emptyset = S_7$	$\emptyset = S_7$

Comme on peut le voir, l'automate déterministe peut être plus gros que l'automate de départ. À quel point ? Si l'automate non déterministe de départ contient n états, l'automate déterministe peut avoir jusqu'à 2^n états. C'est exactement ce qui se passe sur cet exemple : à partir d'un automate non déterministe à 3 états, on obtient un automate déterministe à $2^3 = 8$ états. Cette situation n'est pas exceptionnelle : on peut trouver un grand nombre d'exemples où la déterminisation augmente considérablement la taille de l'automate.

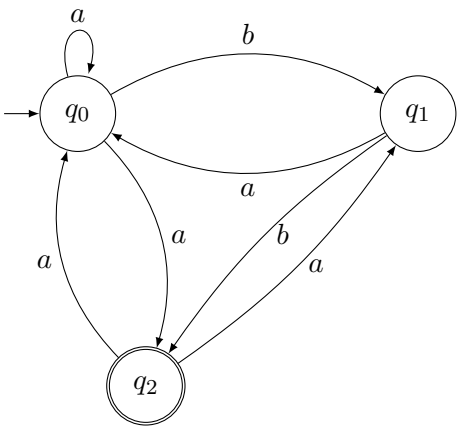


FIGURE 3.10 – Encore un exemple d'automate non déterministe sur l'alphabet $A = \{a, b\}$.

Conclusion

Les automates déterministes, non déterministes et les langages rationnels donnés par expressions régulières correspondent donc aux mêmes langages.

— à tout automate déterministe on peut associer un langage rationnel ;
— à tout langage rationnel on peut associer un automate non déterministe ;
— on peut déterminiser un automate non déterministe.

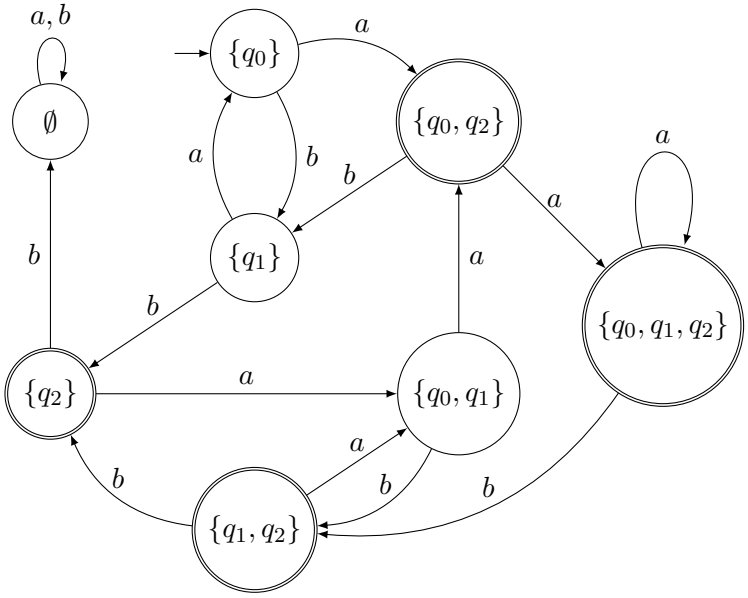


FIGURE 3.11 – L'automate déterministe correspondant au non déterministe de la figure 3.10.

Exercices

(3 - 1) *(Déterminisation)*

Q 1) Dessiner l'automate suivant puis le déterminiser (les états finaux sont représentés par le symbole $\star$) :

			a	b
$\rightarrow$	0		3	0
	1	$\star$	0	0, 3
	2		2	0
	3	$\star$	2	0, 1

Q 2) En utilisant l'automate déterministe, montrer que le mot `abbabbb` est accepté. Donner un chemin acceptant dans l'automate non déterministe.

(3 - 2) *(Constructions)*

Q 1) Construire un automate (déterministe ou non déterministe) qui reconnaît le langage L_1 des mots sur l'alphabet $\{\texttt{a}, \texttt{b}\}$ qui ont exactement deux occurrences de la lettre `b`.

Q 2) Construire un automate (déterministe ou non déterministe) qui reconnaît le langage L_2 des mots sur l'alphabet $\{\texttt{a}, \texttt{b}\}$ qui contiennent le mot `ab`.

Q 3) En utilisant les constructions vues en cours, obtenir un automate pour $L_1 \cup L_2$.

Q 4) En utilisant les constructions vues en cours, obtenir un automate pour $L_1 L_2$.

Q 5) En utilisant les constructions vues en cours, obtenir un automate pour $L_1^\star$.

Q 6) En utilisant les constructions vues en cours, obtenir un automate pour $L_1^\star L_2$.

(3 - 3) *(Déterminisation)*

Q 1) Déterminiser les automates obtenus à l'exercice précédent.

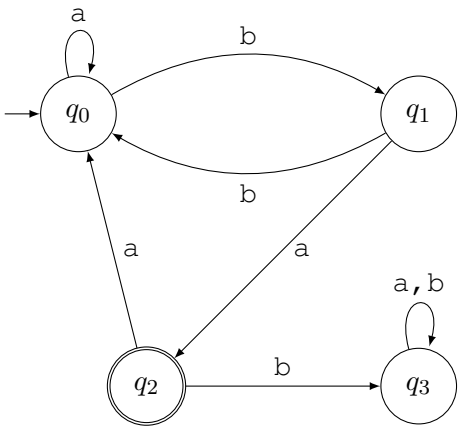
Q 2) Construire un automate pour le langage $L_1^\star \setminus L_2$.

(3 - 4) *(Opérations sur les langages)*

Soit $\mathcal{A}$ un automate déterministe reconnaissant un langage L sur l'alphabet $\{\texttt{a}, \texttt{b}\}$. Dans les questions suivantes, on demande :

— De prouver que le langage en question est rationnel, en expliquant, partant d'un automate $\mathcal{A}$ déterministe quelconque, comment le transformer pour obtenir un automate (déterministe ou non) pour le nouveau langage ;

— Puis de faire la transformation explicitement sur l'automate $\mathcal{A}_1$ ci-dessous :



- $\mathrm{Pref}(L) = \{u | \exists v, uv \in L\}$. C'est l'ensemble des préfixes des mots de L.
- $\mathrm{Suff}(L) = \{u | \exists v, vu \in L\}$. C'est l'ensemble des suffixes des mots de L.
- $b^{-1}L = \{u | bu \in L\}$. C'est l'ensemble des mots de L qui commencent par b auxquels on a enlevé la première lettre (donc le b).
- $\mathrm{half}(L) = \{x_1x_3x_5x_7 \dots x_{2n-1} | x_1x_2x_3 \dots x_{2n} \in L\}$. C'est l'ensemble des mots de L de longueur paire auxquels on a enlevé une lettre sur deux.
- $\mathrm{double}(L) = \{x_1\mathtt{a}x_2\mathtt{a}x_3\mathtt{a}x_4\mathtt{a} \dots x_n\mathtt{a} | x_1x_2x_3 \dots x_n \in L\}$. C'est l'ensemble des mots de L auxquels on a ajouté un symbole `a` entre toutes les lettres.

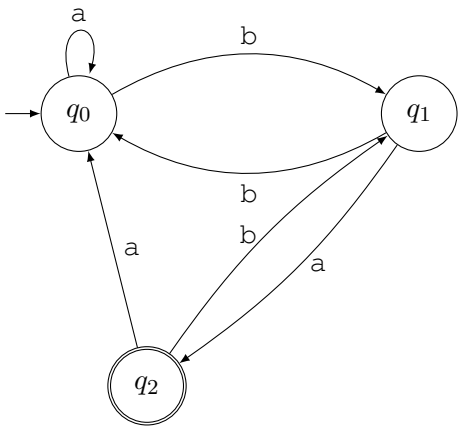
(3 - 5) *(Automate mystère)*

Soit $\mathcal{A}$ un automate déterministe sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$, constitué de n états numérotés de 0 à $n-1$ (q_0 étant l'état initial).

On construit un automate non déterministe $\mathcal{B}$ sur l'alphabet $\{\mathtt{a}, \mathtt{b}, \#\}$ de la façon suivante : $\mathcal{B}$ est constitué de $2n$ copies de l'automate $\mathcal{A}$, c'est à dire deux copies par état de l'automate initial.

Pour chaque état q_i de l'automate initial, on a donc deux copies de $\mathcal{A}$:
— La première copie est identique à $\mathcal{A}$, mais sans état final et avec comme état initial q_i
— La deuxième copie est identique à $\mathcal{A}$, mais sans état initial et avec q_i comme état final
— Pour chaque état (précédemment) final de la première copie, il y a une transition étiquetée # vers l'état q_0 de la deuxième copie.

Q 1) Construire l'automate $\mathcal{B}$ dans le cas de l'automate $\mathcal{A}$ suivant :



Q 2) Que reconnaît l'automate $\mathcal{B}$? On pourra commencer par tester les mots `ba#a`, `ba#ba` et `a#ba`.

(3 - 6) *(Tennis)*

Assia et Bertrand jouent au tennis. C'est Assia qui a le service. A chaque fois que Assia gagne un point, on note la lettre `a`. Si c'est Bertrand qui gagne, on note la lettre `b`.

Soit L l'ensemble des mots qui correspondent aux jeux gagnés par Assia. Par exemple `aaaa` est dans L, `abababaa` est dans L, mais pas `ab` (le jeu n'est pas fini), ni `bbbb` (c'est Bertrand qui a gagné le jeu), ni `aaaaa` (le jeu serait déjà fini après les 4 points d'Assia, donc on est en train de commencer un nouveau jeu).

Construire un automate déterministe pour le langage L. On utilisera comme états $0-0$, $15-0$, etc.

Chapitre 4

COMPLÉMENTS SUR LES AUTOMATES

4.1 Minimisation

On remarque assez facilement qu'il existe plus qu'un seul automate déterministe correspondant à un langage donné. La figure 4.1 montre par exemple quatre automates pour le seul et même langage des mots qui ont un nombre impair d'occurrences de la lettre b.

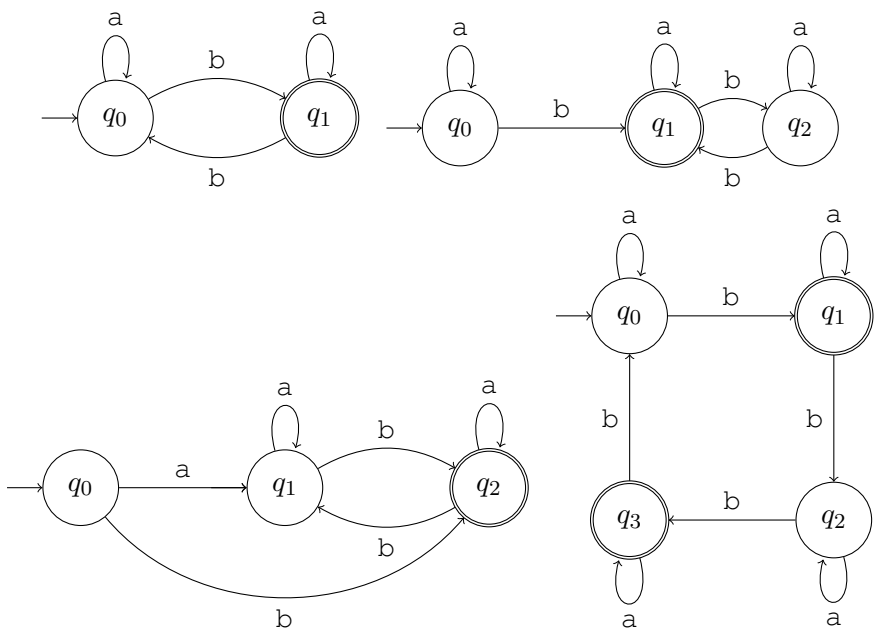


FIGURE 4.1 – Quatre automates déterministes sur l'alphabet $A = \{a, b\}$ qui reconnaissent l'ensemble des mots qui ont un nombre impair de b.

Cependant, on peut montrer qu'en un sens, il existe toujours un unique automate déterministe *minimal* pour un langage L, au sens où :

— (minimal) on ne peut pas faire d'automate déterministe plus petit en nombre d'états pour le langage L ;

— (unique) tout automate déterministe pour le langage L qui utilise le nombre minimum d'états est égal à cet automate.

Sans trop rentrer dans les détails, l'existence de cet automate est assuré par les quelques propositions suivantes :

Définition 4.1 (*Rappel*)

Si A est un automate déterministe et q l'un de ses états, on note L_q le langage des mots acceptés par l'automate si jamais q était l'état initial.

Proposition 4.1

Si A est un automate déterministe et qu'on peut trouver deux états q, q' tels que $L_q = L_{q'}$, alors on peut fusionner les deux états q et q' pour obtenir un automate plus petit qui reconnaît le même langage.

Pour fusionner les états, on élimine l'état q', et on fait pointer toutes les transitions qui allaient vers q' vers q (si q' est initial, il faut au contraire supprimer q).

Preuve : La preuve est relativement compliquée, et peut être passée en première lecture.

Soit A^1 le premier automate, et A^2 le deuxième automate, où q' a été éliminé.

Montrons par récurrence sur la longueur du mot w que :

$$\text{Si } s \neq q' \text{ alors } w \in L^1_s \text{ si et seulement si } w \in L^2_s \qquad \text{(HR)}$$

(Il suffira ensuite d'utiliser le résultat avec $s = q_0$.)

Commençons par $w = \epsilon$. Supposons que $w \in L^1_s$. Donc s est terminal dans A_1. Il est aussi terminal dans A_2, donc le résultat est prouvé. La réciproque s'établit de la même façon.

Supposons maintenant (HR) vérifiée pour tous les mots u de longueur n, et montrons que (HR) est alors vérifiée pour les mots de longueur $n+1$. Soit donc w un mot de longueur $n+1$ et écrivons $w = au$, où a désigne la première lettre de w.

Supposons que $w \in L^1_s$. Notons s' l'état obtenu après lecture de la lettre a. Par définition de l'automate, $u \in L^1_{s'}$. Il y a deux cas :

- $s' \neq q'$: on obtient par (HR) que $u \in L^2_{s'}$. Comme la transition de s à s' en lisant la lettre a est présente également dans A_2 , on en déduit $au \in L^2_s$.
- $s' = q'$: comme $L^1_q = L^1_{q'}$, on a donc $u \in L^1_q$. On déduit par (HR) que $u \in L^2_q$. La transition de s à q' étiquetée par la lettre a a disparu dans l'automate A_2 mais a été remplacée par une transition de s à q. On en déduit $au \in L^2_s$.

On a donc montré que si $w \in L^1_s$ alors $w \in L^2_s$.

Supposons maintenant que $w \in L^2_s$. Notons s' l'état obtenu après lecture de la lettre a. Par définition de l'automate, $u \in L^2_{s'}$. On en déduit par (HR) que $u \in L^1_{s'}$. Il y a deux cas :

- La transition de s à s' était déjà présente dans l'automate A_1. Dans ce cas, on déduit directement $au \in L^1_s$.
- La transition de s à s' n'est pas dans l'automate A_1. Cette situation peut arriver uniquement si $s' = q$ et que la transition de s à s' correspond à une transition qui était auparavant de s à q'. Dans ce cas, comme $L^1_q = L^1_{q'}$, on déduit que $u \in L^1_{q'}$. Et il y a une transition de s à q' portée par la lettre a, donc on déduit que $au \in L^1_s$.

On a donc montré que si $w \in L^2_s$ alors $w \in L^1_s$.

On a donc montré $w \in L^2_s \iff w \in L^1_s$ ce qui termine la preuve. ■

Proposition 4.2

Soit un automate déterministe complet A où :
— pour tout état $q \neq q'$, on a $L_q \neq L_{q'}$;
— tous les états sont atteignables à partir q_0.
Alors A est minimal en nombre d'états.

Preuve : La preuve est relativement compliquée, et peut être passée en première lecture.

Soit B un automate déterministe complet qui reconnait le même langage que A. On va montrer que B a au moins autant d'états que A. Pour ça on va montrer qu'on peut associer, à chaque état de A un état de B de sorte qu'à deux états différents de A on associe deux états différents de B (existence d'une application injective de l'ensemble des états de A dans l'ensemble des états de B).

Pour chaque état q de A, notons w_q un mot qui, partant de l'état q_0, arrive à l'état q. Comme tous les états sont atteignables à partir de l'origine, un tel mot existe nécessairement. On associe alors à l'état q de A l'état s de B obtenu en lisant le mot w_q depuis l'état initial de B.

Soit maintenant q et q' deux états différents de A. Comme ce sont des états différents, par hypothèse, ils ne reconnaissent pas le même langage. Donc on peut trouver un mot u qui est dans L_q et pas dans $L_{q'}$ ou l'inverse (et au plus un seul des deux langages est vide). Pour la suite on considère le cas où u est dans L_q.

Soit s et s' les états de B associés respectivement à q et q'.
Comme u est dans L_q alors $w_q u$ est accepté par A. Et comme A et B reconnaissent le même langage, il est aussi accepté par B. Or w_q amène à l'état s dans B, donc u est dans L_s.
De même comme u n'est pas dans $L_{q'}$ alors le mot $w_{q'} u$ n'est pas accepté par A. Donc $w_{q'}$ amenant sur s' dans B, u n'est pas non plus dans $L_{s'}$.

On en déduit donc que $L_s \neq L_{s'}$ et donc $s \neq s'$.

Finalement on a bien montré que l'automate B a au moins autant d'états que l'automate A.

■

On peut démontrer par la même technique que l'automate A est unique, au sens où deux automates pour un même langage ayant le nombre minimal d'états possibles sont égaux.

Algorithme 4.2 *Minimisation (1/2)*

Pour minimiser un automate :
- On commence par supprimer tous les états non atteignables à partir de l'origine.
- On calcule, pour chaque état q, le langage L_q des mots acceptés si on prend q comme état initial (voir algo suivant).
- On fusionne ensemble tous les états qui correspondent au même langage. Il y a donc autant d'états dans le nouvel automate que de L_q différents. Plus exactement :
  - Le nouvel automate garde, pour chaque L_q différent, un seul état.
  - Il y a une transition de l'état q à l'état q' en lisant un a dans le nouvel automate si, dans l'automate initial, la transition partant de q lisant un a arrive vers un état qui a le même langage que $L_{q'}$.

La difficulté dans cet algorithme est d'identifier les états qui reconnaissent le même langage.

Pour cela, on va utiliser un algorithme dû à Moore qui utilise un principe fort de la justice américaine : la présomption d'innocence. Au départ, on suppose tous les états égaux, jusqu'à preuve du contraire. Si on s'aperçoit que ce n'est pas le cas (par exemple parce qu'ils mènent à des états prouvés différents) alors on les sépare.

Algorithme 4.3 *Minimisation (2/2)*

Pour trouver quels états reconnaissent le même langage :
- On commence initialement par mettre d'un côté tous les états acceptants et de l'autre tous les états non acceptants.
- À chaque étape, les états sont donc divisés en plusieurs parties. On vérifie si cette partition est correcte. Si ce n'est pas le cas, on la subdivise. Donc pour chaque ensemble S de la partition et pour chaque lettre a :
  - on cherche pour chaque état de S dans quel ensemble on arrive si on lit un a ;
  - on subdivise S en séparant les états qui arrivent dans des ensembles différents.
- On recommence jusqu'à ce qu'il n'y ait plus de changement.

L'algorithme est plus clair sur un exemple :

♦ **Exemple**

On considère l'exemple de la figure 4.2.

Au départ, on sépare les états acceptants (ici q_1 et q_3) des autres.

Étape 1 (Début)

A		B				
1	3	0	2	4	5	6

On regarde ensuite les lettres de l'alphabet une par une. On commence par la lettre `a`. On écrit en dessous de chaque état dans quel partie (A ou B) on arrive :

Étape 1 - Lettre `a`

	A		B				
	1	3	0	2	4	5	6
a	A	A	A	A	A	B	B

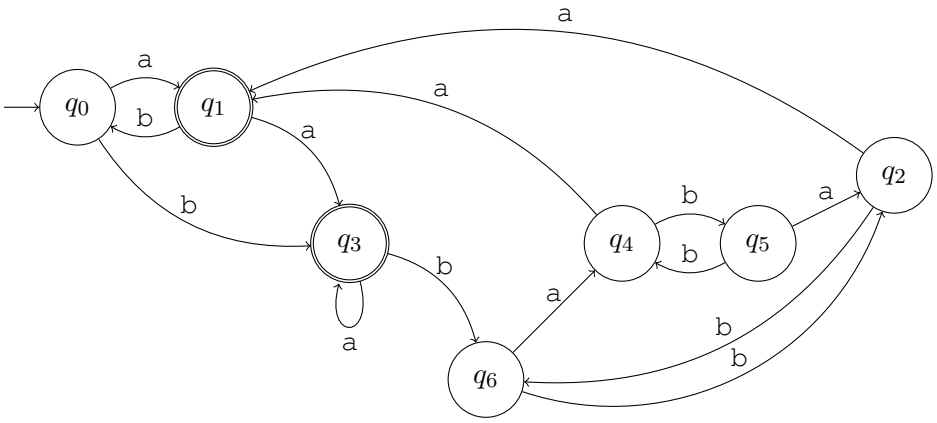


FIGURE 4.2 – Un automate pas très minimal.

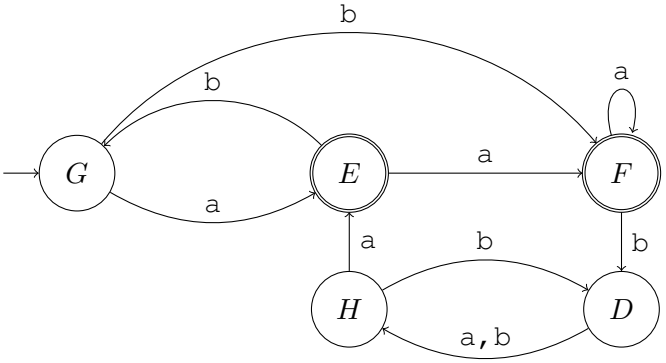


FIGURE 4.3 – Automate minimal qui correspond à l'automate de la figure 4.2.

On voit donc qu'il faut séparer 0,2,4 de 5,6, puisqu'ils n'arrivent pas dans le même ensemble. On obtient donc un nouveau découpage :

A	C	D
1 3	0 2 4	5 6

On passe à la lettre `b` :

Étape 1 - Lettre `b`

	A	C	D
	1 3	0 2 4	5 6
b	C D	A D D	C C

On déduit qu'il faut séparer 1 et 3, et qu'il faut aussi séparer 0 et $2, 4$:

Étape 1 (Fin)

E	F	G	H	D
1	3	0	2 4	5 6

On passe à la deuxième étape. On commence par la lettre `a` :

Étape 2 - Lettre `a`

	E	F	G	H	D
	1	3	0	2 4	5 6
a	F	F	E	E E	H H

Tout va bien, pas besoin de redécouper. On passe à la lettre suivante :

Étape 2 - Lettre `b`

	E	F	G	H	D
	1	3	0	2 4	5 6
b	G	D	F	D D	H H

Tout va bien, pas besoin de redécouper.

Comme aucune redécoupe n'a eu lieu à la deuxième étape, on s'arrête.

On obtient donc un nouvel automate dont les états sont E, F, G, H, D. L'état initial est celui qui contient q_0, donc G. Les états finaux sont ceux qui contiennent des états finaux donc E et F. L'automate minimal résultant est représenté à la figure 4.3.

Un autre exemple un peu plus imposant est présenté dans l'annexe A.

4.2 Examen du fonctionnement de grep et de Lex/jflex

Nous avons tous les outils théoriques pour pouvoir maintenant expliquer le fonctionnement de l'utilitaire Unix `grep` ou du logiciel Lex.

Ces logiciels, partant d'une expression régulière :

— construisent l'automate non déterministe correspondant [1] ;
— déterminisent l'automate ;
— minimisent l'automate ;
— utilisent le nouvel automate pour savoir si la ligne correspond à l'expression régulière.

Sans compter toutes les optimisations possibles, il y a quelques différences essentielles :

— L'algorithme présenté au-dessus dit si le texte intégral vérifie une expression régulière, mais ne dit pas si un texte *contient* un sous-texte qui vérifie une expression régulière (ce qui est souvent ce qu'on cherche avec `grep`).

1. La plupart de ces utilitaires utilisent des automates non déterministes différents de ceux vus en cours, qu'on appelle « automates à ϵ-transition », mais le principe reste le même.

— Dans le cas de `flex`, on ne cherche pas à savoir si la ligne (ou le début de la ligne) vérifie une expression régulière, mais plutôt quelle expression régulière elle vérifie, parmi un ensemble donné. Pour être capable de faire cela, il faut changer la définition d'automate pour avoir des états acceptants de plusieurs types. La modification est assez aisée et ne sera pas discutée plus en détails.

4.3 Langages non rationnels

L'essentiel du cours jusqu'à présent s'est intéressé à la construction d'automates ou d'expressions régulières correspondant à des langages rationnels. Bien entendu, tous les langages ne sont pas rationnels.

Nous allons ici donner quelques exemples de langages non rationnels et des critères (nécessaires ou suffisants) pour montrer qu'un langage n'est pas rationnel.

L'un des critères les plus utiles (même s'il n'est pas suffisant) est le suivant :

Proposition 4.3

Soit L un langage rationnel et supposons qu'il existe un automate A avec n états qui le reconnaît. Alors, si w est un mot de longueur strictement supérieure à n, on passe deux fois par le même état lors de la lecture de w par l'automate A.

Critère 4.4

Pour montrer qu'un langage L n'est pas rationnel :
— On suppose que L est rationnel. Il est donc reconnu par un automate avec n états pour une certaine valeur de n inconnue.
— On trouve un mot w très grand qui est dans L.
— Si on prend n lettres consécutives de w, on passe forcément deux fois par le même état en lisant ces lettres.
— On peut donc répéter le sous-mot entre ces deux états, 0 ou plusieurs fois. On dit qu'on *pompe* sur le sous-mot.
— En regardant les mots obtenus, on peut aboutir à une contradiction en engendrant des mots par *pompage* qui ne sont pas dans L.

Il faut un peu d'expérience pour trouver les bons mots w. On va donner un ou deux exemples.

Proposition 4.4

Soit $L = \{a^k b^k | k \in \mathbb{N}\}$. L est donc le langage des mots qui commencent par des a, finissent par des b, et qui ont autant de a que de b.
Alors L n'est pas rationnel.

Preuve : Supposons que L est rationnel. Il est donc reconnu par un automate déterministe avec n états. Regardons le mot $w = a^{n+1}b^{n+1}$. Le mot w est dans L. Lorsqu'on examine le parcours de l'automate sur le mot w, on voit qu'il passe forcément deux fois par le même état lors de la lecture des $n+1$ premiers a, vu qu'il n'a que n états. Soit q cet état. Découpons le mot w en 3 : $w = rst$:
— r représente les lettres lues avant d'arriver en q, r est donc de la forme a^l pour un certain l ;

— s représente les lettres lues lorsqu'on va de q à q, s est donc de la forme a^k pour un certain $k > 0$;
— et t représente les lettres lues après, t est donc de la forme $a^m b^{n+1}$ avec $l+k+m = n+1$.

On peut résumer schématiquement par

$$q_0 \xrightarrow{a^l} q \xrightarrow{a^k} q \xrightarrow{a^m b^{n+1}} q_f$$

où q_0 est l'état initial et q_f un état final.

D'après la définition de l'automate, si on lit $w = rsst$, le parcours restera le même : on ira de q_0 à q, puis de q à q, puis encore de q à q, puis de q à un état final. Donc $rsst$ est aussi un mot du langage. Mais $rsst = a^l a^k a^k a^m b^{n+1} = a^{n+1+k} b^{n+1}$ donc ne peut pas être dans L, puisqu'il a plus de a que de b. C'est une contradiction, le langage ne peut donc pas être rationnel. ■

Proposition 4.5

Soit $L = \{a^k b^l | k > l\}$. L est donc le langage des mots qui commencent par des a, finissent par des b, et qui ont plus de a que de b.
Alors L n'est pas rationnel.

Preuve : On va donner deux preuves différentes. Supposons que L est rationnel. Il est donc reconnu par un automate déterministe avec n états.

Première preuve : Regardons le mot $w = a^{n+1} b^n$. Lorsqu'on examine le parcours de l'automate sur le mot w, on voit qu'il passe forcément deux fois par le même état lors de la lecture des $n+1$ premiers a, vu qu'il n'a que n états. On peut donc le découper comme précédemment en 3 bouts, rst, où le bout s est de la forme a^k. Mais dans ce cas le mot rt est aussi accepté par l'automate. Ce qui n'est pas possible, car il a moins, ou autant, de a que de b.

Deuxième preuve : Regardons le mot $w = a^{n+2} b^{n+1}$. Lorsqu'on examine le parcours de l'automate sur le mot w, on voit qu'il passe forcément deux fois par le même état lors de la lecture des $n+1$ symboles b, vu qu'il n'a que n états. On peut donc le découper comme précédemment en 3 bouts, $w = rst$, où le bout s est de la forme b^k. Mais dans ce cas le mot $rsst$ est aussi accepté par l'automate. Ce qui n'est pas possible, car il a plus de b que de a. ■

Cette méthode ne marche pas à tous les coups. Par exemple, on ne peut pas la faire fonctionner sur le langage L des mots qui ont un nombre de a et de b différents.

En général, combiner le critère précédent à la méthode suivante fonctionne pour tous les langages

qu'on verra en TD et en examen :

Critère 4.5

Pour montrer qu'un langage L n'est pas rationnel :
- On suppose que L est rationnel.
- On utilise le fait que l'union, l'intersection, la concaténation, etc. de langages rationnels est un langage rationnel pour fabriquer à partir de L un langage L' dont on sait qu'il n'est pas rationnel
- On aboutit à une contradiction.

Proposition 4.6

Soit $L = \{a^k b^l | k \neq l\}$. L est donc le langage des mots qui commencent par des a, finissent par des b, et dont le nombre de a est différent du nombre de b

Alors L n'est pas rationnel.

Preuve : Supposons que L est rationnel. Alors $a^\star b^\star \setminus L = a^\star b^\star \cap \bar{L}$ est aussi rationnel, puisque c'est l'intersection d'un langage rationnel et du complémentaire d'un langage rationnel.

Mais $a^\star b^\star \setminus L$ est l'ensemble des mots qui commencent par des a, finissent par des b et qui ont autant de a que de b. On a déjà prouvé qu'il n'était pas rationnel. ■

Proposition 4.7

Soit $L = \{a^n b^m | n = m$ ou alors $n \times m$ est pair$\}$. Autrement dit : soit n et m sont tous les deux impairs, et dans ce cas il faut que $n = m$, soit l'un des deux (au moins) est pair et dans ce cas $n \neq m$, et on est dans le langage.

Alors L n'est pas rationnel.

On ne peut pas utiliser le critère précédent : si on prend $w = a^{n+1}b^{n+1}$ avec n impair, ajouter plusieurs a peut rendre le nombre de a pair, auquel cas le mot obtenu par pompage ne sortira pas du langage.

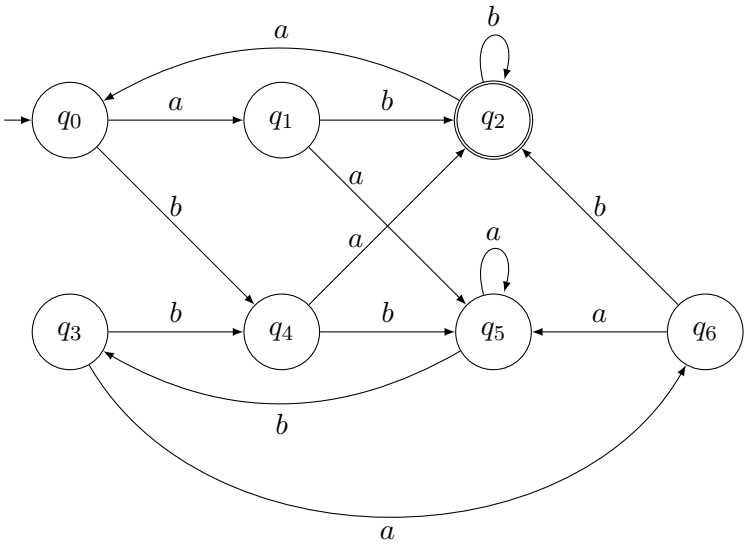
Preuve : Si L est rationnel alors $L' = L \setminus ((aa)^\star b^\star + a^\star (bb)^\star)$ est aussi rationnel.

Mais $L' = \{a^n b^n | n$ impair$\}$ dont on montre facilement qu'il n'est pas rationnel par le précédent critère. ■

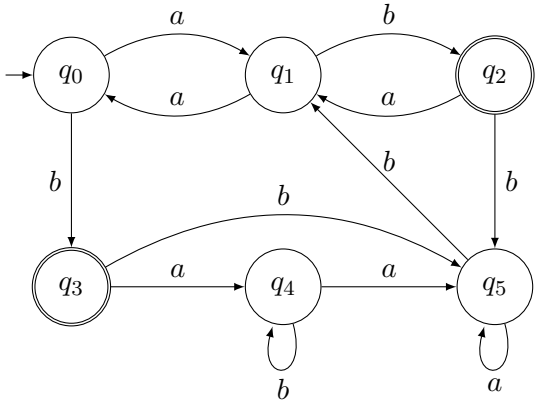
Exercices

(4 - 1) *(Minimisation)*

Q 1) Minimiser l'automate suivant :



Q 2) Même question avec l'automate suivant :



(4 - 2) *(Rationnel ou pas rationnel)* Parmi les 8 langages suivants sur l'alphabet $\{a, b\}$, trouver ceux qui sont rationnels, et ceux qui ne le sont pas. Pour les langages qui sont rationnels, donner un automate ou une expression régulière. Pour les autres, prouver qu'ils ne sont pas rationnels.

a) Le langage des mots qui ont un nombre de a divisible par 4.

b) Le langage des mots qui ont autant de a que de b.

c) Le langage des mots qui n'ont pas trois a consécutifs.

d) $\{xx^R | x \in (a+b)^\star\}$, l'ensemble des palindromes de longueur paire.

e) $\{xwx | x, w \in (a+b)^\star, x \neq \epsilon\}$, c'est à dire l'ensemble des mots qui commencent comme ils finissent (ex : $abaabab = ab \cdot aab \cdot ab$).

f) $\{xwx^R | x, w \in (a+b)^\star, x \neq \epsilon\}$, c'est à dire dont la fin est un palindrome du début (ex : $abaabba = ab \cdot aab \cdot ba$).

g) $\{a^{2n} | n \in \mathbb{N}\}$ l'ensemble des mots qui ne contiennent que des a, et qui en contiennent un nombre pair.

h) $\{a^{n^2} | n \in \mathbb{N}\}$ l'ensemble des mots qui ne contiennent que des a, et qui en contiennent un nombre qui est un carré parfait.

(4 - 3) *(Complexités)*

Q 1) Soit $\mathcal{A}$ un automate, déterministe ou non déterministe, avec n états. Montrer que si $\mathcal{A}$ accepte un mot, alors il accepte un mot de longueur strictement inférieure à n. Peut-on faire mieux ?

Q 2) Montrer que n'importe quel automate pour le langage des mots qui ont au moins 50 lettres a au moins 50 états.

Q 3) Montrer que n'importe quel automate pour le langage des mots dont le nombre de lettres est multiple de 20790 a au moins 20790 états.

Q 4) Donner un automate non déterministe pour le langage des mots dont le nombre de lettres n'est PAS multiple de 20790 avec au maximum 40 états.
Aide : $20790 = 2 \times 3 \times 5 \times 7 \times 9 \times 11$.

Q 5) Soit deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ déterministes avec n_1 et n_2 états reconnaissant respectivement des langages L_1 et L_2. Expliquer comment construire un automate pour le langage $L_1 \Delta L_2$ des mots qui sont dans L_1 ou bien (ou exclusif) dans L_2.

Q 6) En déduire que si L_1 et L_2 sont non vides et $L_1 \neq L_2$ alors on peut trouver un mot de longueur $\leq n_1 \times n_2$ qui est dans L_1 mais pas dans L_2 (ou l'inverse).

Deuxième partie

Langages algébriques

Chapitre 5

GRAMMAIRES (HORS CONTEXTE)

5.1 Définitions et premières propriétés

Définition 5.1

Une grammaire G (hors-contexte/algébrique) est la donnée :
- D'un alphabet T appelé alphabet des terminaux.
- D'un alphabet V appelé alphabet des non-terminaux, ou alphabet des variables (souvent représentés par des lettres majuscules). On distingue dans V un symbole particulier, appelé symbole de départ, et souvent noté S.
- D'un ensemble (fini) de règles :
  - Une règle (on dit aussi *production*) est de la forme $X \to w$ où :
    - X est un non-terminal ;
    - w est un mot sur l'alphabet $V \cup T$.

$$\begin{array}{lcl} S & \to & ST \\ S & \to & \mathtt{a} \\ T & \to & \mathtt{b}S \end{array}$$

TABLE 5.1 – Une grammaire algébrique G_1.

Un premier exemple de grammaire algébrique, composée de 3 règles, est représenté à la table 5.1. Il est d'usage d'écrire sur une même ligne les différentes règles correspondant au même non-terminal. Ainsi on compactera souvent les deux premières règles en

$$S \to ST \mid a$$

Un deuxième exemple, utilisant cette notation compacte est présenté à la table 5.2 :

$$\begin{array}{lcl} S & \to & SS \mid \mathtt{a} \mid T \\ T & \to & TT \mid \mathtt{b}S \end{array}$$

TABLE 5.2 – Une grammaire algébrique G_2.

Définition 5.2

Soit G une grammaire.

Soit u et v deux mots sur l'alphabet $V \cup T$ (donc pouvant contenir des variables ou des terminaux).

On dit que $u \Rightarrow v$ s'il existe une règle $R = X \to w$ et deux mots r, t tels que

— $u = rXt$,

— $v = rwt$.

$\Rightarrow^\star$ consiste à appliquer 0 ou plusieurs fois l'opération $\Rightarrow$: on écrit que $u \Rightarrow^\star v$, et on dit que u *se dérive en* v, s'il existe des mots $u_0, \dots u_n$ tels que

— $u = u_0$,

— $v = u_n$,

— et pour tout $i < n$, $u_i \Rightarrow u_{i+1}$.

Si on veut être plus précis, on peut écrire $u \Rightarrow^n v$, où n est le nombre de fois où $\Rightarrow$ a été appliquée.

Si jamais on travaille sur plusieurs grammaires à la fois, on écrira $\Rightarrow_G$ pour savoir de quelle grammaire on parle. En général ce n'est pas nécessaire.

♦ **Exemple**

Pour la grammaire G_1 de la table 5.1, on a :

— $\texttt{aS} \Rightarrow \texttt{aa}$ (en appliquant la deuxième règle) ;

— $TT \Rightarrow T\texttt{b}S$;

— $S \Rightarrow^\star \texttt{ababa}$, en effet

$$S \Rightarrow ST \Rightarrow STT \Rightarrow \texttt{a}TT \Rightarrow \texttt{a}T\texttt{b}S \Rightarrow \texttt{a}T\texttt{ba} \Rightarrow \texttt{ab}S\texttt{ba} \Rightarrow \texttt{ababa}$$

Définition 5.3

Soit G une grammaire. On note $L(G) = \{u \in T^\star | S \Rightarrow^\star u\}$. Tous les langages obtenus de cette façon s'appellent des langages *algébriques*.

Attention, $L(G)$ ne contient que les mots constitués uniquement de symboles terminaux.

♦ **Exemple**

On a donc $\texttt{ababa} \in L(G_1)$, comme vu précédemment.

On va montrer que $\texttt{b} \notin L(G_1)$.

Pour cela, établissons par récurrence sur k que si $S \Rightarrow^k u$ alors u commence par un a ou un S.

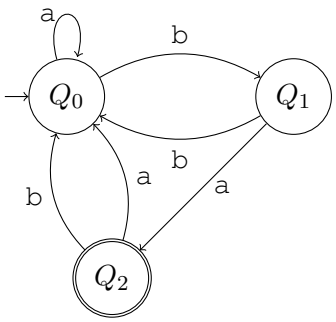
— C'est vrai pour $k = 0$: si $S \Rightarrow^k u$ c'est que $u = S$.

— Supposons que $S \Rightarrow^{k+1} u$. Notons $S \Rightarrow^k v \Rightarrow u$.

Par hypothèse de récurrence, v commence par un a ou un S. Il y a trois possibilités :

— Si la dernière règle appliquée ne touche pas à la première lettre de v, alors v commence toujours par un a ou un S.

— Sinon c'est que v commence par un S. Mais dans ce cas, il est transformé, soit en ST, soit en $\texttt{a}$, donc la première lettre de $\texttt{u}$ sera un $\texttt{a}$ ou un S.



$$\begin{array}{lcl} Q_0 & \to & \mathtt{a}Q_0 \mid \mathtt{b}Q_1 \\ Q_1 & \to & \mathtt{a}Q_2 \mid \mathtt{b}Q_0 \\ Q_2 & \to & \mathtt{a}Q_0 \mid \mathtt{b}Q_0 \mid \epsilon \end{array}$$

FIGURE 5.1 – Un automate déterministe sur l'alphabet $A = \{a, b\}$, et la grammaire qui lui correspond.

♦ **Exemple**

On considère la grammaire G_3 définie par

$$S \to aSb \mid \epsilon$$

Alors $L(G_3) = \{a^n b^n | n \in \mathbb{N}\}$.

En effet, on montre assez facilement par récurrence sur k que $S \Rightarrow^k u$ si et seulement si $u = a^k S b^k$ ou $u = a^{k-1} b^{k-1}$ (le dernier cas ne pouvant pas arriver pour $k = 0$)

Le résultat est clair pour $k = 0, 1$. Supposons le résultat vrai pour k. Soit $S \Rightarrow^{k+1} u$, donc $S \Rightarrow^k v \Rightarrow u$. Par hypothèse de récurrence, $v = a^{k-1} b^{k-1}$ ou $v = a^k S b^k$. On ne peut pas appliquer de règle sur le premier cas, donc $v = a^k S b^k$. Il n'y a que deux règles qui peuvent s'appliquer sur v : la première donne $u = a^k (aSb) b^k = a^{k+1} S b^{k+1}$, la deuxième donne $u = a^k \epsilon b^k = a^k b^k$. La réciproque s'établit de même, ce qui prouve la récurrence.

Proposition 5.1

Soit L un langage rationnel. Alors il existe une grammaire G telle que $L = L(G)$. Plus exactement, soit A un automate déterministe pour le langage L.

On obtient une grammaire pour L de la façon suivante :

— L'ensemble des non-terminaux est l'ensemble des états

— Le non-terminal initial est Q_0, l'état initial de l'automate.

— Pour chaque état Q et chaque lettre a, on a une règle $Q \to a\delta(Q, a)$

— Pour chaque état final Q, on a une règle $Q \to \epsilon$.

On trouvera un exemple à la figure 5.1. Le lecteur attentif remarquera les similarités avec la preuve du théorème 2.3.

Corollaire 5.2

Tous les langages rationnels sont algébriques, mais tous les langages algébriques ne sont pas rationnels.

Preuve de la Proposition: On montre assez facilement par récurrence sur k que $Q_0 \Rightarrow^k u$ si et seulement si :

— u est de longueur $k - 1$ et $u \in L$;

— ou $u = vQ$ avec v de longueur k et Q est l'état obtenu après lecture de v dans l'automate. ■

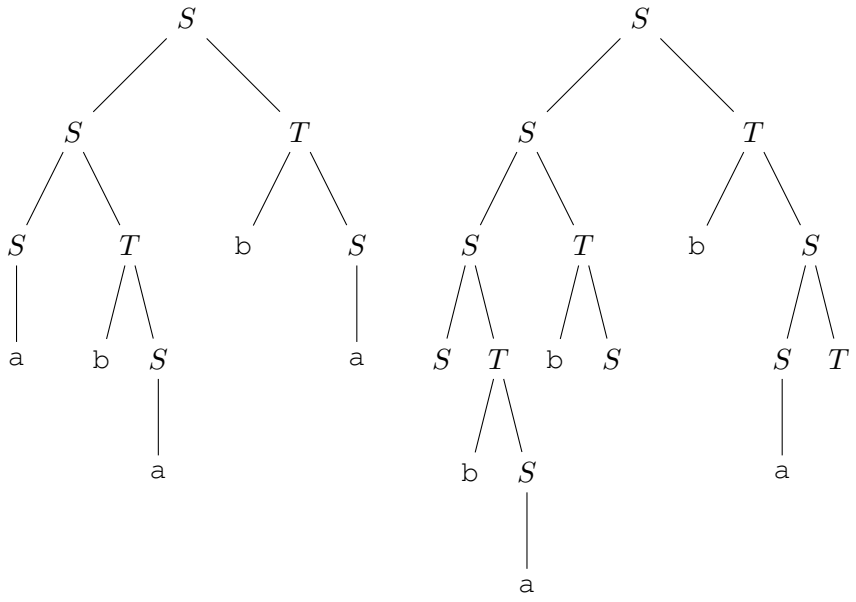


FIGURE 5.2 – Deux arbres de dérivation pour la grammaire G_1 de la table 5.1.

5.2 Arbres de dérivation

On peut représenter graphiquement le fait qu'un mot w appartient au langage $L(G)$ par un arbre :

Définition 5.4 (*Arbre de dérivation*)

Soit G une grammaire. Un arbre de dérivation est un arbre dont les nœuds sont étiquetés par $T \cup V \cup \{\epsilon\}$ et qui vérifie la propriété suivante :

- La racine est S.
- Les nœuds internes sont étiquetés par des non-terminaux.
- Si X est un nœud interne et que ses fils sont $w_1 \dots w_n$, alors $X \to w_1 w_2 \dots w_n$ est une règle de G.

On trouvera deux exemples d'arbres pour la grammaire G_1 à la figure 5.2.

Définition 5.5

À tout arbre de dérivation A, on associe le mot w constitué de la concaténation de toutes les feuilles dans l'ordre d'un parcours en profondeur de gauche à droite. On dit alors que A est un arbre de dérivation pour w.

Un arbre de dérivation est dit partiel si l'une des feuilles est un non-terminal. Sinon, il est dit total.

♦ Exemple

Le premier arbre de dérivation de la figure 5.2 est total. Le mot associé à cet arbre est le mot `ababa`. Le deuxième arbre est partiel, puisqu'il a deux variables S et une variable T en feuille. Le mot associé est $S\texttt{bab}S\texttt{ba}T$.

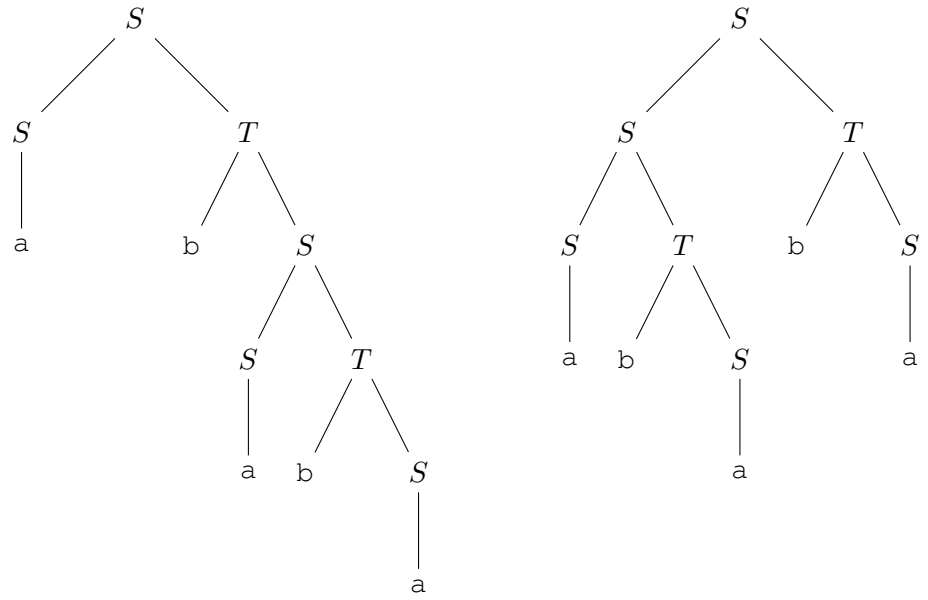


FIGURE 5.3 – Deux arbres de dérivation différents pour le mot ababa pour la grammaire G_1 présentée en table 5.1.

Théorème 5.3

Soit G une grammaire. Alors l'ensemble des mots correspondant à des arbres de dérivation totaux pour G est exactement $L(G)$.

Preuve : On montre par récurrence sur $k > 0$ que $S \Rightarrow^k w$ si et seulement si il existe un arbre A avec $k-1$ nœuds internes dont w est le mot des feuilles. ■

Lorsqu'on est intéressé par des applications concrètes des grammaires, l'objet essentiel est l'arbre de dérivation : on ne s'intéresse pas uniquement à savoir si w appartient à $L(G)$, mais aussi à l'arbre de dérivation qui le prouve.

En particulier, une question essentielle est de savoir si un mot admet un seul arbre de dérivation.

Définition 5.6

Une grammaire est ambiguë [1] s'il existe un mot w qui a plusieurs arbres de dérivation.

La grammaire de la table 5.1 est ambiguë. En effet, on trouvera en figure 5.3 deux arbres de dérivation pour le mot ababa.

Sur quelques exemples, il est souvent facile de montrer qu'une grammaire est ambiguë, en exhibant deux arbres de dérivation pour le même mot. Il est beaucoup plus difficile de prouver qu'une grammaire est non-ambiguë. En fait c'est même **impossible** dans le cas général : il n'est pas possible d'écrire un algorithme qui est capable de prouver qu'une grammaire est ambiguë ou non.

Attention, ce n'est pas parce qu'une grammaire est ambiguë qu'il n'est pas possible de trouver une grammaire non-ambiguë pour le même langage. Par exemple, il est facile de prouver que la grammaire constituée des seules règles $S \to abS \mid a$ est non ambiguë et engendre les mêmes mots que G_1.

1. Attention, on écrit ambigu, ambiguë, et ambiguïté. Le tréma sur le e est pour éviter qu'on prononce « ambigue » comme « digue ».

5.3 Algorithme CYK pour analyser un mot - cas sans ϵ

Nous présentons maintenant un premier algorithme pour analyser un mot et déterminer s'il est dans le langage $L(G)$ et, le cas échéant, donner un arbre de dérivation qui le prouve.

Dans les faits, cet algorithme n'est pas toujours utilisable, car il est un peu lent, comme on l'expliquera à la fin de l'exposition. C'est toutefois le meilleur algorithme qui fonctionne sur tout type de grammaire ; les autres algorithmes que nous présenterons ne peuvent être appliqués qu'à des grammaires spécialisées.

Pour simplifier l'exposition, on va se contenter du cas où aucune règle ne contient ϵ. Les modifications à apporter si ce n'est pas le cas seront décrites rapidement à la fin, mais ne sont pas à connaître.

L'algorithme utilisé s'appelle l'algorithme CYK, pour Cocke-Younger-Kasami. L'algorithme dans sa forme originelle suppose que les grammaires sont dans une certaine forme, appelée forme normale de Chomsky. On va utiliser une variante plus générale de cet algorithme, plus simple à présenter et qui ne nécessite pas de construire cette forme normale.

Algorithme 5.7 *Algorithme CYK, phase 1*

Partant d'une grammaire où aucune règle n'est de la forme $X \to \epsilon$, transformer la grammaire en une nouvelle grammaire en forme C2F, c'est-à-dire où les règles sont de la forme

- $X \to YZ$
- $X \to Y$
- $X \to a$

Cette forme est assez simple à obtenir, et nous ne donnerons pas d'explication détaillée. Le but est principalement :

- de s'arranger pour qu'aucune règle n'ait plus de 3 lettres, ce qui peut se faire en ajoutant des variables : si on a une règle $X \to ABCDE$, on peut la transformer en $X \to YZ$, $Y \to AB$, $Z \to CW$ et $W \to DE$, où Y, Z, W sont des nouvelles variables ;
- de s'arranger pour que les paires soient constituées de deux variables et non pas de variables et de non-terminaux, en remplaçant les non-terminaux par des variables : on remplace toute règle $X \to aY$ par des règles $X \to AY$ et $A \to a$.

Bien entendu, il faut essayer d'économiser au maximum l'ajout de nouvelles variables.

Algorithme 5.8 *Algorithme CYK, phase 2*

Pour chaque non-terminal X, calculer la clôture $Cl(X)$, qui est l'ensemble des non-terminaux Y tel que $Y \Rightarrow^\star X$.

On peut voir ça comme un problème de graphe : les nœuds sont les non-terminaux, il y a un arc de A à B s'il y a une règle $A \to B$. La clôture de X est alors l'ensemble des variables Y tel qu'il y a un chemin de Y à X.

Algorithme 5.9 *Algorithme CYK, phase 3*

Étant donné un mot w de n lettres numérotées de 1 à n, on construit un tableau à deux dimensions $T[i][j]$, de sorte que $T[i][j]$ est l'ensemble des variables X telles que $X \Rightarrow^\star w_i w_{i+1} w_{i+2} \dots w_{i+j}$.

On remplit le tableau T de la façon suivante :

— Pour $j = 0$ et tout i (de 1 à n) :
  — Si la lettre w_i est un a, on met dans $T[i][0]$ l'ensemble des variables X telles qu'il y a une règle $X \to a$.
  — On applique l'opérateur de clôture au résultat : on ajoute toutes les variables Y qui sont dans la clôture des variables déjà présentes.
— Pour j quelconque (de 1 à $n-1$), pour chaque i (de 1 à $n-j$) et pour chaque $k < j$:
  — S'il y a une règle $X \to YZ$ où
    — Y est dans $T[i][k]$, i.e. $Y \Rightarrow^\star w_i \dots w_{i+k}$, et que
    — Z est dans $T[i+k+1][j-k-1]$, i.e. $Z \Rightarrow^\star w_{i+k+1} \dots w_{i+j}$
    — alors on ajoute X à $T[i][j]$ car $X \to YZ \Rightarrow^\star w_i \dots w_{i+j}$.
  — On applique l'opérateur de clôture au résultat : on ajoute toutes les variables Y qui sont dans la clôture des variables déjà présentes.

Une fois l'algorithme terminé, le mot est accepté si $T[1][n-1]$ contient le non-terminal S.

La troisième est la plus compliquée et mérite d'être faite sur un exemple.

♦ **Exemple**

Partons de la grammaire

$$\begin{array}{lcl} S & \to & TS \mid Ta \\ T & \to & aU \mid b \\ U & \to & S \mid UTU \mid cT \end{array}$$

Première phase Il faut d'abord la transformer pour qu'elle soit de la bonne forme. On va commencer par régler le problème de la règle $U \to UTU$ qui est trop longue. Pour cela, on ajoute une variable R :

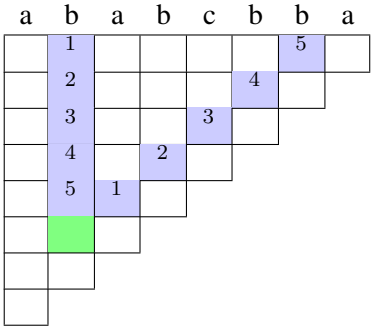
$$\begin{array}{lcl} S & \to & TS \mid Ta \\ T & \to & aU \mid b \\ U & \to & S \mid RU \mid cT \\ R & \to & UT \end{array}$$

Ensuite il faut régler le cas des règles $S \to Ta$, $U \to cT$ et $T \to aU$, qui contiennent une variable et un terminal. Pour cela on ajoute deux variables A et C.

$$\begin{array}{lcl} S & \to & TS \mid TA \\ T & \to & AU \mid b \\ U & \to & S \mid RU \mid CT \\ R & \to & UT \\ A & \to & a \\ C & \to & c \end{array}$$

Deuxième phase On calcule ensuite la relation clôture. On a uniquement une production $U \to S$, donc $cl(R) = \{R\}, cl(U) = \{U\}, cl(T) = \{T\}$ et $cl(S) = \{U, S\}$. La relation clôture signifie qu'une variable S peut être vue comme une variable U (puisque $U \to S$).

Troisième phase La troisième phase est la plus longue. Essayons de l'exécuter sur le mot `ababcbba`. On va remplir le tableau qui est en forme de triangle (j en ligne, i en colonne) :



Mis à part la première ligne qui est un cas particulier, le principe général est le suivant. Le but est de mettre dans la case coloriée en vert tous les non-terminaux X qui peuvent engendrer le mot délimité par les deux bandes bleues, c'est-à-dire dans l'exemple le mot `babcbb`. Pour cela, on va regarder toutes les façons de couper ce mot en deux, et on va voir s'il existe une façon de couper ce mot de sorte que la première moitié corresponde à une variable Y, la deuxième à une variable Z, et qu'il y ait de plus une règle $X \to YZ$ dans le langage. Les variables correspondant aux différentes façons de couper le mot ont déjà été calculées et représentées dans les cellules en bleue. Il reste ensuite à les apparier, et à voir si ça correspond à une règle du langage. On les apparie comme indiqué sur le schéma : la cellule 1 avec la cellule 1, la cellule 2 avec la 2, etc.

Une fois cette opération effectuée, il reste à clôturer la cellule, en ajoutant toutes les variables W qui peuvent se transformer en une des variables X qu'on a mis dans la cellule.

Première ligne Pour remplir la première ligne, on cherche, pour chaque lettre, quelle variable peut la produire en une étape. Pour a c'est A, pour b c'est T, et pour c c'est C.

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A

On applique ensuite l'opérateur de clôture, qui ne change rien.

Deuxième ligne (La deuxième ligne est un cas particulier, le cas intéressant arrive pour la troisième ligne.)

Pour remplir la deuxième ligne, on regarde, dans chaque cellule, si la conjonction de la cellule juste en haut, et en haut à droite correspond à une règle connue.

Par exemple, pour savoir ce qu'il y a sur la cellule verte, on regarde les cellules bleues :

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A

Ici, on s'aperçoit que TA (ce qu'on lit en bleu) est une règle $S \to TA$, donc on écrit S dans la cellule. On fait pareil partout :

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A
	S			U		S	

On passe à la clôture. L'opérateur de clôture transforme tous les S en U, S :

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A
	S,U			U		S,U	

Troisième ligne L'exécution sur la troisième ligne donne le résultat suivant :

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A
	S,U			U		S,U	
T	R			R	S		

Expliquons uniquement le coefficient en vert. On apparie les cellules en bleu comme indiqué précédemment. Comme il n'y a rien dans la cellule en bas à droite, les deux seules possibilités sont d'apparier S avec T ou U avec T. La première correspond à la règle de grammaire $R \to ST$, donc on écrit R dans la cellule.

Pour finir la troisième ligne, il reste à clôturer, ce qui change les S en S, U :

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A
	S,U			U		S,U	
T	R			R	S,U		

Tableau final On remplit les autres lignes de la même façon et on aboutit au résultat suivant :

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A
	S,U			U		S,U	
T	R			R	S,U		
	U						
T	R						
S,U							

Comme il y a un S dans la dernière case, le mot est accepté.

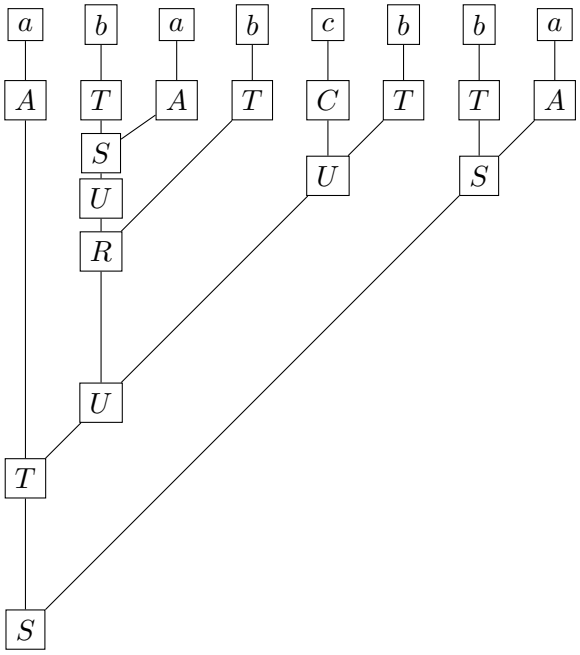
Si jamais on veut connaître l'arbre de dérivation qui le prouve, il faut faire un parcours en arrière du tableau, et se demander *pourquoi* on a mis un S dans cette case [2].

a	b	a	b	c	b	b	a
A	T	A	T	C	T	T	A
	S^7,U^6			U^8		S^9,U	
T	R^4			R	S,U		
	U^3						
T^2	R						
S^1,U							

2. Ce qui peut rappeler certains exercices du cours de modélisation.

— S^1 vient de $S^1 \to T^2 S^9$.
— T^2 vient de $T^2 \to AU^3$.
— U^3 vient de $U^3 \to R^4 U^8$.
— R^4 vient de $R^4 \to U^6 T$.
— U^6 vient de $U^6 \to S^7$ (c'est ici qu'on voit l'importance de la règle de clôture).
— S^7 vient de $S^7 \to TA$.
— U^8 vient de $U^8 \to CT$.
— S^9 vient de $S^9 \to TA$.
— Et enfin les autres C, T, A viennent de c, b et a.

On obtient ainsi un arbre de dérivation, que l'on va représenter différemment d'habitude :



Un exemple un peu plus imposant est donné dans l'annexe B.

Théorème 5.4

L'algorithme précédent produit un arbre de dérivation pour le mot w de taille n en un temps $O(n^3)$.

La complexité vient évidemment du temps de remplissage du tableau. C'est un tableau à deux dimensions, constitués de $O(n^2)$ cases et il faut $O(n)$ pour remplir chaque case.

C'est beaucoup, mais on ne peut pas faire mieux dans le cas général d'une grammaire quelconque. Pour les besoins pratiques, c'est beaucoup trop lent. On verra dans la suite du cours deux types d'algorithmes qui permettent d'aller beaucoup plus vite, mais qui ne fonctionnent que sur des grammaires particulières.

Traitement des ϵ (Hors programme)

Le seul endroit de l'algorithme qui est impacté s'il y a des règles du type $X \to \epsilon$ est la construction de l'ensemble clôture.

S'il n'y a pas d'ϵ, la seule possibilité pour avoir $R \Rightarrow^\star T$ est d'avoir une suite de dérivations du type $R \to X \to Y \to Z \to .. \to T$, ce qui se détecte bien, par exemple avec un parcours de graphe.

Si on a des règles avec ϵ, d'autres comportements peuvent apparaître. Ainsi, on peut avoir $R \Rightarrow^\star T$ si par exemple $R \to XY$, $X \to T$ et $Y \to \epsilon$.

L'opérateur clôture est donc plus difficile à calculer. Sans trop de détails, la méthode générale est la suivante :

— Phase 2.1 : commencer par trouver les variables X telles que $X \Rightarrow^\star \epsilon$ (l'algorithme est facile mais mérite qu'on y réfléchisse).

— Phase 2.2 : si $X \to YZ$ et que $Y \Rightarrow^\star \epsilon$, ajouter dans le graphe une flèche de X à Z. De même, on met une flèche de X à Y si $Z \Rightarrow^\star \epsilon$.

Exercices

(5 - 1) *(Une grammaire)*

Dans cet exercice, on considère la grammaire suivante

$$\begin{array}{lcl} S & \to & \texttt{a}S \mid TT \\ T & \to & \texttt{b}T \mid \texttt{c}ST \mid \texttt{d} \end{array}$$

Q 1) Montrer que $S \Rightarrow^\star \texttt{aaadd}$. Donner un arbre de dérivation.

Q 2) Même question avec `aadbcaaddd`.

Q 3) Montrer que $S \not\Rightarrow^\star \texttt{bcad}$.

Q 4) Expliquer pourquoi il est relativement facile de savoir, pour un mot w sur l'alphabet $\{\texttt{a},\texttt{b},\texttt{c},\texttt{d}\}$, si $S \Rightarrow^\star w$ pour cette grammaire.

(5 - 2) *(Une autre grammaire)*

$$\begin{array}{lcl} S & \to & \texttt{a}S \mid TT \\ T & \to & \texttt{b}T \mid ST \mid \texttt{a} \end{array}$$

Q 1) Montrer que $S \Rightarrow^\star \texttt{bbabaaa}$. Donner un arbre de dérivation.

Q 2) Montrer que $S \Rightarrow^\star \texttt{aaaa}$. Donner deux arbres différents.

(5 - 3) *(Encore une grammaire !)*

$$\begin{array}{lcl} S & \to & \texttt{a} \mid ESE \\ E & \to & \texttt{a} \mid \texttt{b} \end{array}$$

Q 1) Expliquer quels sont les mots engendrés par cette grammaire.

(5 - 4) *(Ambiguïté)* On considère la grammaire suivante, dont les terminaux sont $\{\texttt{1},\texttt{+}\}$

$$S \to S\texttt{+}S \mid \texttt{1}$$

Q 1) Montrer que la grammaire est ambiguë. Donner deux arbres de dérivation pour `1+1+1`.

Q 2) Donner une grammaire non ambiguë pour le même langage.

On considère la grammaire suivante, dont les non terminaux sont E, S et les terminaux sont $\{\texttt{nop},\texttt{if},\texttt{then},\texttt{else},\texttt{true},\texttt{false},\texttt{␣}\}$

$$\begin{array}{lcl} S & \to & \texttt{nop} \mid \texttt{if␣}E\texttt{␣then␣}S \mid \texttt{if␣}E\texttt{␣then␣}S\texttt{␣else␣}S \\ E & \to & \texttt{true} \mid \texttt{false} \end{array}$$

Q 3) Montrer que la grammaire est ambiguë. Donner un mot qui a deux arbres de dérivation.

Une solution classique pour résoudre le problème est d'adopter la grammaire suivante.

$$\begin{array}{lcl} S & \to & G \mid B \\ G & \to & \texttt{nop} \mid \texttt{if␣}E\texttt{␣then␣}G\texttt{␣else␣}G \\ B & \to & \texttt{if␣}E\texttt{␣then␣}S \mid \texttt{if␣}E\texttt{␣then␣}G\texttt{␣else␣}B \\ E & \to & \texttt{true} \mid \texttt{false} \end{array}$$

Q 4) Essayez la grammaire sur des exemples, et expliquer pourquoi le problème est résolu.

(5 - 5) *(Profils)* On se place dans cet exercice sur l'alphabet $\{\texttt{a},\texttt{b},\texttt{c}\}$. Soit $w = w_1 w_2 \dots w_n$ un mot. Le *profil* de w est une représentation visuelle dans le plan du mot w comme une ligne brisée reliant les points $(x_0, y_0), (x_1, y_1) \dots (x_n, y_n)$ de sorte que
— $(x_0, y_0) = (0, 0)$
— $x_i = x_{i-1} + 1$
— Si $w_i = \texttt{a}$, alors $y_i = y_{i-1} + 1$
— Si $w_i = \texttt{b}$, alors $y_i = y_{i-1} - 1$
— Si $w_i = \texttt{c}$, alors $y_i = y_{i-1}$

Q 1) Dessiner le profil des mots `abaab`, `aabaacbbbaabb`, `bcbaccaab`.

Q 2) À quoi ressemble le profil d'un mot w qui a autant de lettres `a` que de lettres `b` ?

Q 3) Soit w un mot qui a autant de lettres `a` que de lettres `b`. On suppose que w commence par la lettre `a`. Montrer qu'on peut découper w en 4 parties, $w = \texttt{a}u\texttt{b}v$ de sorte que les mots u et v aient autant de a que de b (il peut être pertinent d'utiliser les questions précédentes).

On considère la grammaire

$$S \to \texttt{a}S\texttt{b}S \mid \texttt{b}S\texttt{a}S \mid \texttt{c}S \mid \epsilon$$

On va démontrer que cette grammaire engendre exactement les mots qui ont autant de `a` que de `b`.

Q 4) Montrer par récurrence sur k que, si u est un mot sur l'alphabet $\{\texttt{a},\texttt{b},\texttt{c},S\}$ tel que $S \Rightarrow^k u$, alors u a autant de `a` que de `b`.

Q 5) Montrer par récurrence forte sur la taille de w que, si w est un mot sur l'alphabet $\{\texttt{a},\texttt{b},\texttt{c}\}$ qui a autant de `a` que de `b`, alors $S \Rightarrow^\star w$. On raisonnera en fonction de la première lettre de w (si w en a une).

On considère maintenant que la lettre `a` représente une parenthèse ouvrante, et la lettre `b` une parenthèse fermante. On s'intéresse au langage L des mots bien parenthésés, c'est-à-dire où toute parenthèse ouvrante est refermée par exactement une parenthèse fermante. Par exemple, le mot `aacbb` est bien parenthésé : $((c))$, le mot `acabcbccab` est bien parenthésé : $(c()c)cc()$, mais le mot `baabba` n'est pas bien parenthésé : $)(())$, ni le mot `acbb` : $(c))$.

Q 6) À quoi ressemble le profil d'un mot w bien parenthésé ? On ne demande pas de preuve.

Q 7) Donner une grammaire pour les mots bien parenthésés. On ne demande pas de preuve.

(5 - 6) *(CYK)* On considère la grammaire :

$$\begin{array}{lcl} S & \to & AS \mid TT \\ T & \to & BT \mid SA \mid \texttt{c} \\ A & \to & B \mid T \mid \texttt{a} \\ B & \to & BB \mid \texttt{b} \end{array}$$

Q 1) Exécuter l'algorithme CYK sur les mots `bacbc`, `bcca`, `bbacbc`. Donner un arbre de dérivation pour les mots acceptés par la grammaire.

Chapitre 6

AUTOMATES À PILE

Les automates finis, vus précédemment, permettent de reconnaître les langages rationnels. Il y a un équivalent pour les grammaires (algébriques), les automates à pile.

Comme leur nom l'indique, les automates à pile disposent, en plus d'un ensemble d'états, d'une pile, et les transitions peuvent manipuler cette pile : empiler des symboles sur la pile, ou les dépiler.

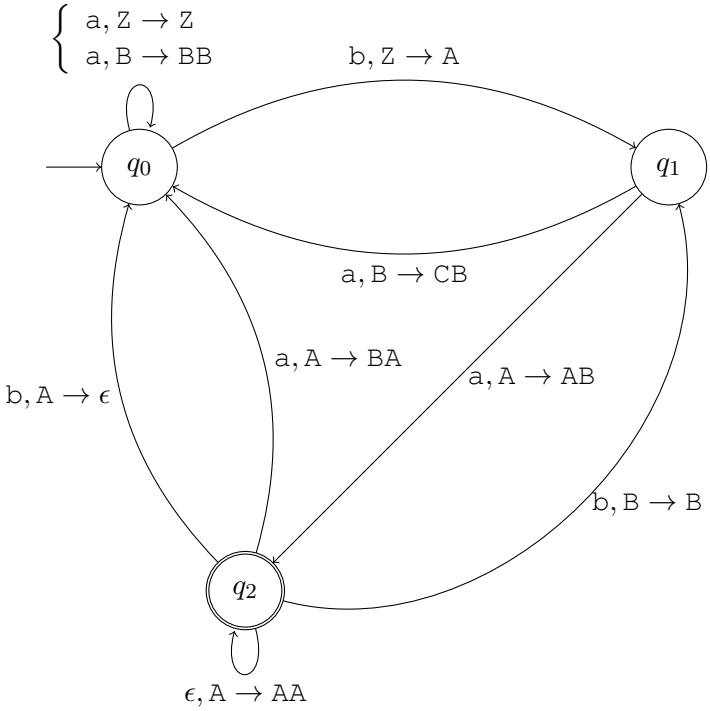


FIGURE 6.1 – Un premier exemple d'automate à pile sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$.

La relation de transition dans un automate à pile est plus compliquée que dans un automate fini. En effet la transition ne dépend pas uniquement de la lettre lue, mais également du symbole en haut de la pile. Un premier exemple est présenté dans la figure 6.1.

Une transition du type $x, Y \to ZW$ signifie si je lis un x, et que le symbole en haut de la pile est un Y, alors je peux suivre la transition. Pour cela, je remplace le Y en haut de la pile par ZW.

Attention — *Attention*

Attention, dans toute la suite, la tête de la pile est à gauche.

Ainsi :
— $\texttt{a}, \texttt{Z} \to \texttt{Z}$ signifie : si je lis un `a` et que le symbole en haut de la pile est un `Z`, alors je peux suivre la transition, sans changer la pile.
— $\texttt{b}, \texttt{Z} \to \texttt{A}$ signifie : si je lis un `b` et que le symbole en haut de la pile est un `Z`, alors je peux suivre la transition, en changeant le `Z` en `A`.
— $\texttt{a}, \texttt{A} \to \texttt{BA}$ signifie : si je lis un `a` et que le symbole en haut de la pile est un `A`, alors je peux suivre la transition, en empilant un `B`.
— $\texttt{a}, \texttt{A} \to \epsilon$ signifie : si je lis un `a` et que le symbole en haut de la pile est un `A`, alors je peux suivre la transition, en dépilant le `A`.
— $\epsilon, \texttt{A} \to \texttt{BA}$ signifie : si le symbole en haut de la pile est un `A`, alors je peux suivre la transition, en empilant un `B`.

La toute dernière transition est appelée une transition *instantanée*, au sens où elle se produit sans qu'une seule lettre du mot d'entrée ne soit lue. Elles permettent de manipuler les symboles en haut de la pile sans lire de lettre.

Examinons par exemple ce qui se passe sur le mot `abab`. On partira toujours avec une pile contenant uniquement au départ le symbole `Z`.
— Au départ, la seule règle qui peut s'appliquer est la règle $\texttt{a}, \texttt{Z} \to \texttt{Z}$ qui nous emmène en q_0. On lit donc le premier `a`, sans changer l'état de la pile, qui est toujours `Z`.
— On lit ensuite le `b`. La seule règle qui peut s'appliquer est la règle $\texttt{b}, \texttt{Z} \to \texttt{A}$ qui nous emmène en q_1. La pile contient maintenant un `A`.
— On lit le `a` suivant. Comme la tête de la pile est un `A`, la seule règle possible est la règle $\texttt{a}, \texttt{A} \to \texttt{AB}$ qui nous emmène en q_2. La pile contient maintenant `AB`.
— On lit le `b` suivant. Comme la tête de la pile est un `A`, il y a deux règles possibles : la règle $\texttt{b}, \texttt{A} \to \epsilon$ et la règle $\epsilon, \texttt{A} \to \texttt{AA}$. La première lit (*consomme*) le `b`, mais pas la deuxième. On décide dans cet exemple (mais les deux choix sont possibles) d'appliquer la deuxième règle. On reste donc en q_2, et on ajoute un `A`. La pile contient maintenant `AAB`.
— On est toujours à la lecture du `b`. Cette fois-ci, on décide de changer de règle (mais les deux choix sont possibles). On utilise donc la règle $\texttt{b}, \texttt{A} \to \epsilon$, qui efface le `A` et nous emmène en q_0. La pile contient maintenant `AB`.

6.1 Définitions

Définition 6.1

Un automate à pile non déterministe (NPDA) $\mathcal{A}$ sur un alphabet A est la donnée :
— D'un ensemble fini Q, appelé ensemble des états. Un état particulier, souvent noté q_0, est privilégié dans l'ensemble Q, qu'on appelle état initial.
— D'un ensemble d'états finaux $F \subseteq Q$.
— D'un alphabet Γ appelé l'alphabet de pile. Un symbole est privilégié, souvent noté `Z`, et on l'appelle symbole de pile initial.
— D'une fonction $\delta : Q \times (A \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^\star)$.
$\delta(q, a, X) \ni (q', W)$ signifie que, partant de q, lisant un a et avec le symbole X en tête de pile, on peut arriver dans l'état q' en remplaçant X par W.

♦ **Exemple**

L'automate à pile de la figure 6.1 est donné par :
— $Q = \{q_0, q_1, q_2\}$. L'état initial est q_0. L'ensemble des états finaux est $F = \{q_2\}$.
— $\Gamma = \{\mathtt{Z}, \mathtt{A}, \mathtt{B}\}$. Le symbole de pile initial est $\mathtt{Z}$.
— A priori, δ se représente comme un tableau à 3 dimensions.
Pour plus de simplicité, on la représente ici comme 3 tableaux (puisque 3 états) à 2 dimensions, un par état.

q_0	a	b	ϵ
Z	$q_0, \mathtt{Z}$	$q_1, \mathtt{A}$	
A			
B	$q_0, \mathtt{BB}$		

q_1	a	b	ϵ
Z			
A	$q_2, \mathtt{AB}$		
B	$q_0, \mathtt{CB}$		

q_2	a	b	ϵ
Z			
A	$q_0, \mathtt{BA}$	q_0, ϵ	$q_2, \mathtt{AA}$
B		$q_1, \mathtt{B}$	

On peut aussi utiliser un seul tableau en spécifiant tous les couples (état,symbole de pile) possibles :

δ	$q_0, \mathtt{Z}$	$q_0, \mathtt{A}$	$q_0, \mathtt{B}$	$q_1, \mathtt{Z}$	$q_1, \mathtt{A}$	$q_1, \mathtt{B}$	$q_2, \mathtt{Z}$	$q_2, \mathtt{A}$	$q_2, \mathtt{B}$
a	$q_0, \mathtt{Z}$		$q_0, \mathtt{BB}$		$q_2, \mathtt{AB}$	$q_0, \mathtt{CB}$		$q_0, \mathtt{BA}$	
b	$q_1, \mathtt{A}$							q_0, ϵ	$q_1, \mathtt{B}$
ϵ								$q_2, \mathtt{AA}$	

Définition 6.2

Soit $\mathcal{A}$ un NPDA. Une *configuration* est la donnée d'un état q et du mot $P \in \Gamma^\star$ représentant le contenu de la pile. Dans le mot P, on suppose que le haut de la pile est à gauche et le fond de la pile à droite. Donc
— $P = \mathtt{BA}$ représente une pile composée de deux symboles, le symbole en haut de la pile étant un $\mathtt{B}$.
— $P = \epsilon$ représente une pile vide.

Définition 6.3

Soit $\mathcal{A}$ un NPDA sur l'alphabet A. Soit $x \in A \cup \{\epsilon\}$, $P, Q \in \Gamma^*$, et q et q' des états.

On dit que $(q, P) \overset{x}{\mapsto} (q', Q)$ si
— $P = XY$ avec $X \in \Gamma$ et $Y \in \Gamma^*$. X représente le haut de la pile, et Y le reste de la pile.
— $Q = WY$ avec $W \in \Gamma^*$. On a donc remplacé X par W.
— $(q', W) \in \delta(q, x, X)$.

Si w est un mot, on dit que $(q, P) \overset{w}{\mapsto} (q', Q)$ si (q', Q) s'obtient à partir de (q, P) en lisant successivement toutes les lettres de w (ou des ϵ) : plus précisément, il existe $w_1 \dots w_n, s_0, s_1, \dots s_n$ et $P_0, P_1, \dots P_n$ tel que $s_0 = q$, $P_0 = P$, $s_n = q', P_n = Q$, $w = w_1 \dots w_n$ et pour tout i, $(q_{i-1}, P_{i-1}) \overset{w_i}{\longmapsto} (q_i, P_i)$.

♦ **Exemple**

Le déroulement présenté précédemment peut donc s'écrire :

$$q_0, \mathtt{Z} \overset{\mathtt{a}}{\mapsto} q_0, \mathtt{Z} \overset{\mathtt{b}}{\mapsto} q_1, \mathtt{A} \overset{\mathtt{a}}{\mapsto} q_2, \mathtt{AB} \overset{\epsilon}{\mapsto} q_2, \mathtt{AAB} \overset{\mathtt{b}}{\mapsto} q_0, \mathtt{AB}$$

Ou de façon plus compacte

$$q_0, \mathtt{Z} \overset{abab}{\longmapsto} q_0, \mathtt{AB}$$

Contrairement à un automate fini, un automate à pile peut avoir plusieurs types de comportements bien différents sur un mot w :

— Cas idéal : on avance progressivement dans l'automate, en empilant ou dépilant au passage.
— Cas problématique 1 : on se retrouve bloqué à un état donné, car aucune transition ne nous permet d'avancer, même si le mot n'est pas entièrement lu pour autant.
— Cas problématique 2 (sous cas du premier) : on se retrouve bloqué à un état donné, car la pile est maintenant vide. Or, toutes les opérations nécessitent de regarder le symbole en début de pile.
— Cas problématique 3 : on lit une infinité de ϵ, en changeant la pile, et on obtient un calcul qui ne s'arrête jamais. C'est par exemple ce qui peut arriver dans l'exemple de la figure 6.1 : on peut dans l'état q_2 empiler une infinité de $\mathtt{A}$ sans jamais progresser.

Cette diversité des comportements se retrouve dans les différentes façons d'accepter un mot. Dans la suite, on se contentera de la définition suivante, mais il existe beaucoup de variantes :

Définition 6.4

On définit trois modes d'acceptation :
— Un mot w est accepté par un automate à pile $\mathcal{A}$ par état final s'il existe un état final $q_f \in F$ et un contenu de pile P tel que $q_0, \mathtt{Z} \overset{w}{\longmapsto} q_f, P$.
— Un mot w est accepté par un automate à pile $\mathcal{A}$ par pile vide s'il existe un état q tel que $q_0, \mathtt{Z} \overset{w}{\longmapsto} q, \epsilon$.
— Un mot w est accepté par un automate à pile $\mathcal{A}$ par état final et pile vide s'il existe un état $q_F \in F$ tel que $q_0, \mathtt{Z} \overset{w}{\longmapsto} q_F, \epsilon$.

Proposition 6.1

Les trois modes d'acceptation sont équivalents : pour tout automate qui accepte par état final, on peut construire un automate qui accepte par pile vide le même langage, et de même pour les 5 autres cas.

Preuve : États finaux$\rightarrow$ pile vide. On commence par ajouter un nouvel état q_{-1} à l'automate, et une transition de q_{-1} vers q_0 transformant le fond de pile en $\mathtt{Z}\$$. Grâce à cette transformation on peut supposer qu'aucun mot n'aboutit à la pile vide. Soit F l'ensemble des états finaux. On crée un nouvel automate avec un nouvel état q. De chaque état final, on crée une transition lisant ϵ et ne modifiant pas l'état de la pile vers ce nouvel état q. Dans q, on ajoute des transitions lisant ϵ qui vident la pile.

Pile vide$\rightarrow$ états finaux. On crée un nouvel état initial q_{-1}, et une transition de q_{-1} vers q_0 en lisant ϵ qui ajoute un $\$$ en fond de pile (de sorte que la pile contient maintenant $\mathtt{Z}\$$). On ajoute enfin un état q_f pour unique état final et une transition de tous les états (sauf q_{-1}) vers q_f en lisant ϵ avec $\$$ en symbole de pile et qui élimine le $\$$. La seule façon de suivre cette transition est donc que la pile dans l'automate d'origine soit vide (pour que $\$$ se retrouve maintenant en tête de pile).

Les autres cas sont similaires. ■

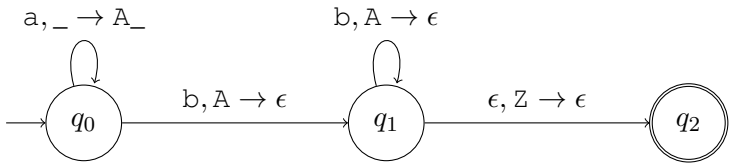


FIGURE 6.2 – Un exemple d'automate à pile sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Le symbole _ représente n'importe quel symbole de pile.

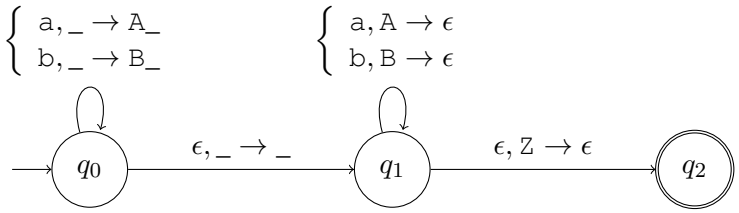


FIGURE 6.3 – Un autre exemple d'automate à pile sur l'alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Le symbole _ représente n'importe quel symbole de pile.

Dans la suite, on supposera pour les preuves qu'on accepte simultanément par état final et pile vide. Cependant, pour les exemples, on se contentera souvent d'acceptation par état final.

6.2 Exemples

♦ **Exemple**

Examinons la figure 6.2. On s'intéresse à la reconnaissance par état final. Le comportement intuitif de l'automate est le suivant : au départ, à chaque fois qu'on lit une lettre a, on empile un A. Ensuite, tant qu'on lit une lettre b, on dépile un A. Pour qu'on puisse prendre la transition de q_1 à q_2 , il faut donc qu'il ne reste plus rien sur la pile sauf le symbole Z initial. C'est donc qu'on a lu autant de a que de b. Le langage reconnu est donc $\{\mathtt{a}^n\mathtt{b}^n, n > 0\}$. Notons que :

— Sur le mot ϵ et plus généralement sur les mots de la forme $\mathtt{a}^n$, l'automate s'arrête dans l'état q_0, en ayant empilé un certain nombre de A.

— Sur le mot $\mathtt{a}^n\mathtt{b}^k$ avec $0 < k < n$, l'automate s'arrête dans l'état q_1, en ayant un certain nombre de A sur la pile.

— Sur le mot $\mathtt{a}^n\mathtt{b}^m$ avec $m > n$, l'automate plante : à un moment donné, on est en q_1 avec Z sur la pile et on a reconnu a^nb^n, donc on peut aller en q_2 avec une pile vide et alors plus aucune règle ne s'applique, or il reste des b à reconnaître.

♦ **Exemple**

Examinons la figure 6.3. On s'intéresse à la reconnaissance par état final. Le comportement intuitif de l'automate est le suivant : au départ, à chaque fois qu'on lit une lettre a, on empile un A et on empile un B quand on lit un b. À un moment donné, on décide de changer d'état pour aller à q_1. Ensuite, on dépile des A quand on lit des a, et des B quand on lit des b. Il est clair que les mots reconnus sont exactement les palindromes de longueur paire. Il s'agit d'un

automate non-déterministe puisqu'il faut savoir à quel moment passer de q_0 à q_1, donc deviner le milieu du mot.

Si le mot n'est pas un palindrome, ou si on devine mal le milieu du mot, on va bloquer, soit en q_1, soit en q_2.

6.3 Équivalence automates à pile et grammaires

Nous allons maintenant indiquer comment transformer une grammaire en automate à pile, et un automate à pile en grammaire, ce qui prouve qu'ils reconnaissent les mêmes langages :

Théorème 6.2

Tout langage algébrique (donné par une grammaire algébrique) est reconnaissable par un automate à pile.

Preuve : On commence par changer la grammaire, de symbole de départ S, pour que toutes les règles soient d'une des formes suivantes :

$$A \to \epsilon \qquad A \to B \qquad A \to BC \qquad A \to a$$

L'automate aura 3 états, q_0, q_1 et q_F, le troisième étant l'état final. Le fonctionnement de l'automate se déroule essentiellement dans la pile sur l'état q_1.

— Une transition $q_0 \xrightarrow{\epsilon, Z \to SZ} q_1$ pour amorcer le processus.
— Pour chaque transition $A \to \epsilon$, on a une transition $q_1 \xrightarrow{\epsilon, A \to \epsilon} q_1$.
— Pour chaque transition $A \to B$, on a une transition $q_1 \xrightarrow{\epsilon, A \to B} q_1$.
— Pour chaque transition $A \to BC$, on a une transition $q_1 \xrightarrow{\epsilon, A \to BC} q_1$.
— Pour chaque transition $A \to a$, on a une transition $q_1 \xrightarrow{a, A \to \epsilon} q_1$.
— Une transition $q_1 \xrightarrow{\epsilon, Z \to \epsilon} q_F$ pour terminer.

La preuve que la transformation est correcte est assez évidente, mais longue. Une illustration de la transformation est présentée à la figure 6.4. ■

Théorème 6.3

Tout langage reconnaissable par un automate à pile est algébrique (donné par une grammaire algébrique).

Preuve : On part d'un automate à pile qui reconnaît simultanément par pile vide et état final.

Pour chaque état q, q' et chaque symbole de pile X, on crée une variable $X_q^{q'}$. Intuitivement, $X_q^{q'}$ est l'ensemble des mots qu'on accepte si on accepte par pile vide et état final, que q est l'état initial, q' l'état final, et X le symbole de pile initial.

Les règles de la grammaire sont les suivantes :

— Pour la transition $q \xrightarrow{a, A \to \epsilon} \tilde{q}$, une règle $A_q^{\tilde{q}} \to a$.
— Pour la transition $q \xrightarrow{a, A \to B} \tilde{q}$, des règles (une par q') $A_q^{q'} \to a B_{\tilde{q}}^{q'}$.
— Pour la transition $q \xrightarrow{a, A \to BC} \tilde{q}$, des règles (une par q' et p) $A_q^{q'} \to a B_{\tilde{q}}^{p} C_p^{q'}$.
— Des règles $S \to Z_{q_0}^{q_F}$ pour chaque état final q_F.

La preuve est assez technique, mais l'idée peut s'expliquer. Soit w un mot accepté avec q comme état initial, q' comme état final et X comme symbole de pile, et supposons que la première étape est d'appliquer la transition $q \xrightarrow{a, X \to YZ} \tilde{q}$. On se retrouve

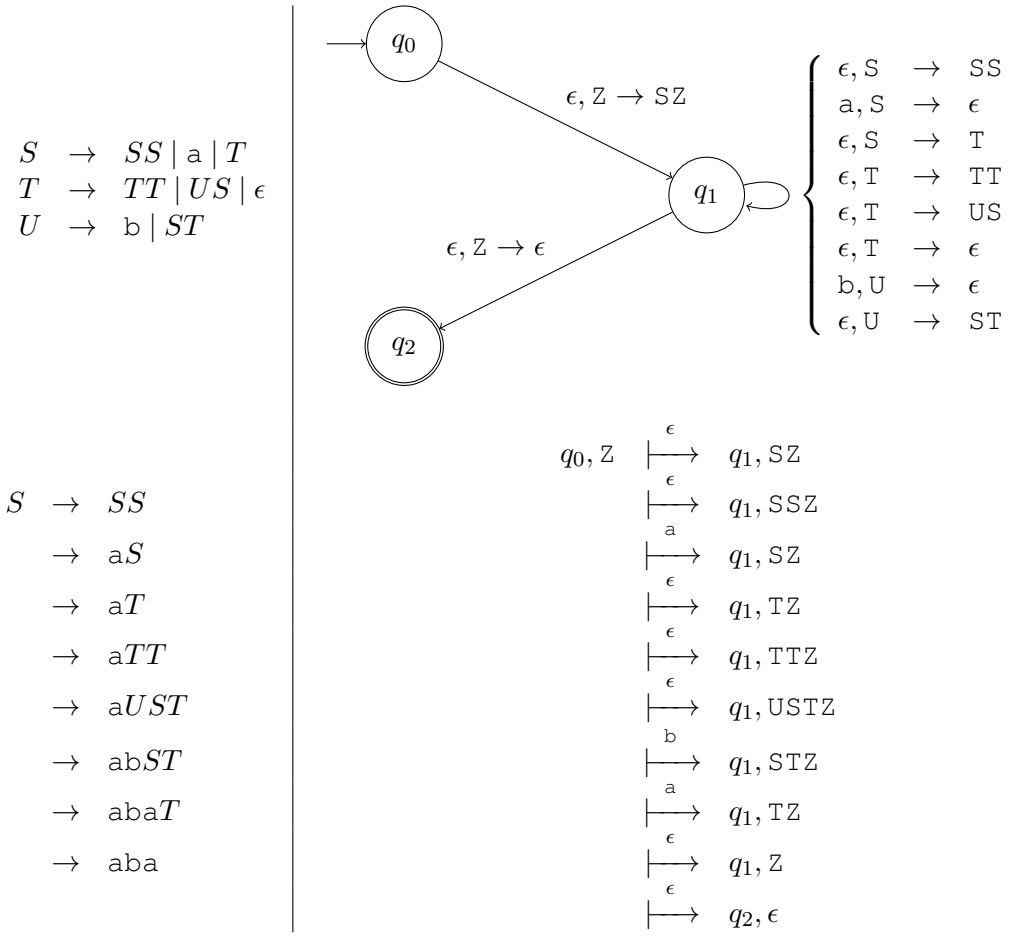
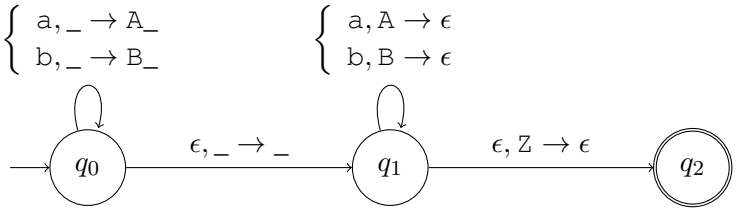


FIGURE 6.4 – Conversion d'une grammaire en automate à pile. En dessous un exemple de dérivation dans la grammaire et de la dérivation associée dans l'automate à pile.

donc dans l'état $\tilde{q}$, avec YZ en fond de pile et on peut écrire $w = au$. Pour accepter le mot w partant de q, il faut donc accepter le mot u partant de $\tilde{q}$. Il faut donc atteindre l'état final q' en partant de $\tilde{q}$, et dépiler YZ. Pour dépiler YZ, il faut d'abord réussir à dépiler Y (ce qui nous amène dans un état p) puis réussir à dépiler Z. ■



$$
\begin{array}{lcl|lcl|lcl|l}
q_0 & \xrightarrow{\texttt{a,Z}\to\texttt{AZ}} & q_0 & Z_0^0 \to \texttt{a}A_0^0 Z_0^0 & q_0 & \xrightarrow{\texttt{b,Z}\to\texttt{BZ}} & q_0 & Z_0^0 \to \texttt{b}B_0^0 Z_0^0 & & & & \\
 & & & Z_0^0 \to \texttt{a}A_0^1 Z_1^0 & & & & Z_0^0 \to \texttt{b}B_0^1 Z_1^0 & & & & \\
 & & & Z_0^0 \to \texttt{a}A_0^2 Z_2^0 & & & & Z_0^0 \to \texttt{b}B_0^2 Z_2^0 & & & & \\
 & & & Z_0^1 \to \texttt{a}A_0^0 Z_0^1 & & & & Z_0^1 \to \texttt{b}B_0^0 Z_0^1 & & & & \\
 & & & Z_0^1 \to \texttt{a}A_0^1 Z_1^1 & & & & Z_0^1 \to \texttt{b}B_0^1 Z_1^1 & & & & \\
 & & & Z_0^1 \to \texttt{a}A_0^2 Z_2^1 & & & & Z_0^1 \to \texttt{b}B_0^2 Z_2^1 & & & & \\
 & & & Z_0^2 \to \texttt{a}A_0^0 Z_0^2 & & & & Z_0^2 \to \texttt{b}B_0^0 Z_0^2 & & & & \\
 & & & Z_0^2 \to \texttt{a}A_0^1 Z_1^2 & & & & Z_0^2 \to \texttt{b}B_0^1 Z_1^2 & q_0 & \xrightarrow{\epsilon,\texttt{Z}\to\texttt{Z}} & q_1 & Z_0^0 \to Z_1^0 \\
 & & & Z_0^2 \to \texttt{a}A_0^2 Z_2^2 & & & & Z_0^2 \to \texttt{b}B_0^2 Z_2^2 & & & & Z_0^1 \to Z_1^1 \\
 & & & & & & & & & & & Z_0^2 \to Z_1^2 \\
q_0 & \xrightarrow{\texttt{a,A}\to\texttt{AA}} & q_0 & A_0^0 \to \texttt{a}A_0^0 A_0^0 & q_0 & \xrightarrow{\texttt{b,A}\to\texttt{BA}} & q_0 & A_0^0 \to \texttt{b}B_0^0 A_0^0 & & & & \\
 & & & A_0^0 \to \texttt{a}A_0^1 A_1^0 & & & & A_0^0 \to \texttt{b}B_0^1 A_1^0 & q_0 & \xrightarrow{\epsilon,\texttt{A}\to\texttt{A}} & q_1 & A_0^0 \to A_1^0 \\
 & & & A_0^0 \to \texttt{a}A_0^2 A_2^0 & & & & A_0^0 \to \texttt{b}B_0^2 A_2^0 & & & & A_0^1 \to A_1^1 \\
 & & & A_0^1 \to \texttt{a}A_0^0 A_0^1 & & & & A_0^1 \to \texttt{b}B_0^0 A_0^1 & & & & A_0^2 \to A_1^2 \\
 & & & A_0^1 \to \texttt{a}A_0^1 A_1^1 & & & & A_0^1 \to \texttt{b}B_0^1 A_1^1 & q_0 & \xrightarrow{\epsilon,\texttt{B}\to\texttt{B}} & q_1 & B_0^0 \to B_1^0 \\
 & & & A_0^1 \to \texttt{a}A_0^2 A_2^1 & & & & A_0^1 \to \texttt{b}B_0^2 A_2^1 & & & & B_0^1 \to B_1^1 \\
 & & & A_0^2 \to \texttt{a}A_0^0 A_0^2 & & & & A_0^2 \to \texttt{b}B_0^0 A_0^2 & & & & B_0^2 \to B_1^2 \\
 & & & A_0^2 \to \texttt{a}A_0^1 A_1^2 & & & & A_0^2 \to \texttt{b}B_0^1 A_1^2 & q_1 & \xrightarrow{\texttt{a,A}\to\epsilon} & q_1 & A_1^1 \to a \\
 & & & A_0^2 \to \texttt{a}A_0^2 A_2^2 & & & & A_0^2 \to \texttt{b}B_0^2 A_2^2 & q_1 & \xrightarrow{\texttt{b,B}\to\epsilon} & q_1 & B_1^1 \to b \\
q_0 & \xrightarrow{\texttt{a,B}\to\texttt{AB}} & q_0 & B_0^0 \to \texttt{a}A_0^0 B_0^0 & q_0 & \xrightarrow{\texttt{b,B}\to\texttt{BB}} & q_0 & B_0^0 \to \texttt{b}B_0^0 B_0^0 & q_1 & \xrightarrow{\epsilon,\texttt{Z}\to\epsilon} & q_2 & Z_1^2 \to \epsilon \\
 & & & B_0^0 \to \texttt{a}A_0^1 B_1^0 & & & & B_0^0 \to \texttt{b}B_0^1 B_1^0 & & \text{Début} & & S \to Z_0^2 \\
 & & & B_0^0 \to \texttt{a}A_0^2 B_2^0 & & & & B_0^0 \to \texttt{b}B_0^2 B_2^0 & & & & \\
 & & & B_0^1 \to \texttt{a}A_0^0 B_0^1 & & & & B_0^1 \to \texttt{b}B_0^0 B_0^1 & & & & \\
 & & & B_0^1 \to \texttt{a}A_0^1 B_1^1 & & & & B_0^1 \to \texttt{b}B_0^1 B_1^1 & & & & \\
 & & & B_0^1 \to \texttt{a}A_0^2 B_2^1 & & & & B_0^1 \to \texttt{b}B_0^2 B_2^1 & & & & \\
 & & & B_0^2 \to \texttt{a}A_0^0 B_0^2 & & & & B_0^2 \to \texttt{b}B_0^0 B_0^2 & & & & \\
 & & & B_0^2 \to \texttt{a}A_0^1 B_1^2 & & & & B_0^2 \to \texttt{b}B_0^1 B_1^2 & & & & \\
 & & & B_0^2 \to \texttt{a}A_0^2 B_2^2 & & & & B_0^2 \to \texttt{b}B_0^2 B_2^2 & & & &
\end{array}
$$

FIGURE 6.5 – Un automate à pile déjà vu et la grammaire équivalente (!).

$$
\begin{array}{rl|lcl}
 & q_0, \mathtt{Z} & S & \to & Z_0^2 \\
\overset{\mathtt{a}}{\longmapsto} & q_0, \mathtt{AZ} & & \to & \mathtt{a}A_0^1 Z_1^2 \\
\overset{\mathtt{b}}{\longmapsto} & q_0, \mathtt{BAZ} & & \to & \mathtt{ab}B_0^1 A_1^1 Z_1^2 \\
\overset{\epsilon}{\longmapsto} & q_1, \mathtt{BAZ} & & \to & \mathtt{ab}B_1^1 A_1^1 Z_1^2 \\
\overset{\mathtt{b}}{\longmapsto} & q_1, \mathtt{AZ} & & \to & \mathtt{abb}A_1^1 Z_1^2 \\
\overset{\mathtt{a}}{\longmapsto} & q_1, \mathtt{Z} & & \to & \mathtt{abba}Z_1^2 \\
\overset{\epsilon}{\longmapsto} & q_2, \epsilon & & \to & \mathtt{abba}
\end{array}
$$

FIGURE 6.6 – Un exemple de dérivation dans l'automate à pile et la dérivation associée dans la grammaire.

♦ **Exemple**

La transformation est présentée en figure 6.5 et le lien entre les calculs de l'automate à pile et les dérivations de la grammaire correspondante est présentée dans la figure 6.6. Bien que le lien apparaisse clairement sur l'exemple, la dérivation de la grammaire ne s'obtient pas de façon si évidente. Ainsi, pour savoir que la deuxième dérivation de la grammaire est $Z_0^2 \to \mathtt{a}A_0^{\mathbf{1}} Z_{\mathbf{1}}^2$ plutôt que $Z_0^2 \to \mathtt{a}A_0^{\mathbf{0}} Z_{\mathbf{0}}^2$, il faut savoir « à l'avance » que le $\mathtt{A}$ que nous venons de mettre sur la pile disparaîtra quand nous serons dans l'état q_1, c'est-à-dire qu'il faut connaître la transition $q_1, \mathtt{AZ} \overset{\mathtt{a}}{\longmapsto} q_1, \mathtt{Z}$ de l'automate, transition qui arrive bien plus tard.

Dans cet exemple, le résultat est assez imposant : on passe d'un automate à 3 états et 6 transitions à une grammaire de 28 non-terminaux et 66 productions. On peut démontrer qu'il y a des exemples où on ne peut pas faire plus simple que cette méthode.

Dans notre cas, on peut simplifier. Il y a en effet plein de variables qu'on ne peut pas transformer, donc qu'on peut éliminer, comme la variable A_1^2. On obtient :

$$
\begin{array}{llll}
Z_0^2 \to \mathtt{a}A_0^1 Z_1^2 & A_0^1 \to \mathtt{a}A_0^1 A_1^1 & B_0^1 \to \mathtt{a}A_0^1 B_1^1 & Z_0^2 \to \mathtt{b}B_0^1 Z_1^2 \\
A_0^1 \to \mathtt{b}B_0^1 A_1^1 & B_0^1 \to \mathtt{b}B_0^1 B_1^1 & Z_0^2 \to Z_1^2 & A_0^1 \to A_1^1 \\
B_0^1 \to B_1^1 & A_1^1 \to \mathtt{a} & B_1^1 \to \mathtt{b} & Z_1^2 \to \epsilon \\
S \to Z_0^2 & & &
\end{array}
$$

Qu'on peut simplifier (en remplaçant A_1^1, B_1^1, Z_1^2 par leur expression) en :

$$
\begin{array}{l}
S \to \mathtt{a}A \mid \mathtt{b}B \mid \epsilon \\
A \to \mathtt{a}A\mathtt{a} \mid \mathtt{b}B\mathtt{a} \mid \mathtt{a} \\
B \to \mathtt{a}A\mathtt{b} \mid \mathtt{b}B\mathtt{b} \mid \mathtt{b}
\end{array}
$$

On peut comprendre cette grammaire : S est l'ensemble des palindromes pairs, A est l'ensemble des palindromes pairs suivis de la lettre $\mathtt{a}$, B est l'ensemble des palindromes pairs suivis de la lettre $\mathtt{b}$.

6.4 Automates déterministes

Dans une perspective de compilation, les automates non déterministes ne sont pas utilisables, puisqu'on ne sait pas quelles transitions suivre. On s'intéresse donc aux automates à pile déterministes. La définition n'est pas si évidente que ça, car on s'autorise, même dans un automate déterministe, à avoir des transitions étiquetées par ϵ, mais il ne faut pas qu'elles puissent rentrer en conflit avec d'autres transitions :

Définition 6.5

Un automate à pile est déterministe si pour chaque état q et symbole de pile X :

— S'il y a une transition sortant de q étiquetée par ϵ, X, alors elle est unique, et il n'y a pas de transition étiquetée par a, X.

— S'il y a une transition sortant de q étiquetée par a, X alors elle est unique.

L'automate de la figure 6.2 qui reconnaît $\{\mathtt{a}^n\mathtt{b}^n, n > 0\}$ est déterministe.

Contrairement au cas des automates finis qui est bien compris, il n'existe pas de procédure pour déterminiser un automate à pile non déterministe :

Théorème 6.4 (*Non prouvé*)

Il existe des langages acceptés par automate à pile, mais pas par automate à pile déterministe. Le langage des palindromes en fait partie.

On comprend intuitivement pourquoi on a besoin du non déterminisme pour les palindromes : il faut deviner où se trouve la moitié du mot.

On en déduit donc qu'il est impossible d'utiliser des automates déterministes pour reconnaître n'importe quelle grammaire. En fait l'algorithme CYK présenté au chapitre suivant est actuellement l'algorithme le plus rapide (asymptotiquement) pour savoir si un mot est reconnu par une grammaire, et sa complexité en $O(n^3)$ le rend complètement inadapté à une utilisation en compilation.

On se restreint donc aux langages qu'on peut obtenir avec des automates déterministes :

Définition 6.6

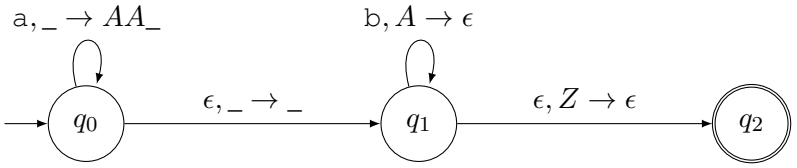
Un langage est algébrique déterministe s'il est accepté par un automate à pile déterministe.

À quoi ressemblent les langages algébriques déterministes, et comment construire des automates déterministes pour ces langages sera la principale question étudiée lors du prochain cours.

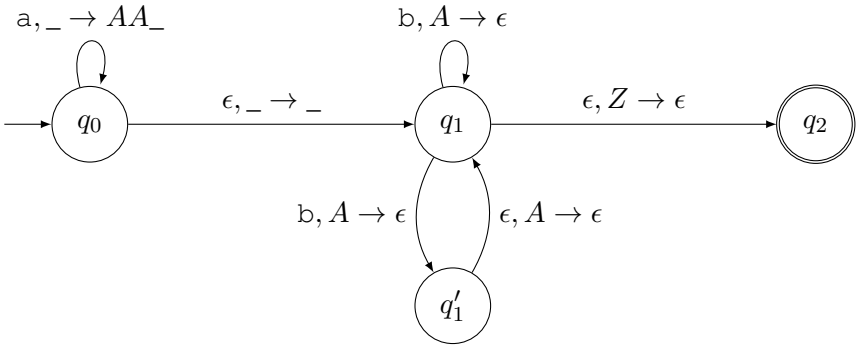
Exercices

(6 - 1) *(Automates à pile)*

Q 1) Trouver le langage reconnu par état final par l'automate à pile suivant (_ représente n'importe quel symbole de pile) :



Q 2) Même question avec l'automate suivant :



(6 - 2) *(Constructions)* Dans tout l'exercice, on considère l'alphabet $\{\mathtt{a}, \mathtt{b}\}$.

Q 1) Construire un automate qui reconnait les mots qui ont autant de `a` que de `b`.

Q 2) Transformer le premier automate de **(6 - 1)** pour le rendre déterministe.

On rappelle que $|u|$ est la longueur du mot u.

Q 3) Construire un automate qui reconnaît l'ensemble des mots de la forme $u_1\mathtt{a}v\mathtt{b}u_2$ tels que $|u_1| + |u_2| = |v|$.

C'est donc l'ensemble des mots qu'on peut découper en trois parties (séparées par un `a` et un `b`) de sorte que la deuxième partie est aussi grande que les deux autres réunies.

Par exemple `abababababb` est dans le langage, puisque

$$\mathtt{abababababb} = \underbrace{\mathtt{ab}}_{u_1} \cdot \mathtt{a} \cdot \underbrace{\mathtt{baaba}}_{v} \cdot \mathtt{b} \cdot \underbrace{\mathtt{abb}}_{u_2}$$

(6 - 3) *(Opérations sur les langages)*

Q 1) Soit L_1 et L_2 deux langages algébriques. Expliquer comment construire une grammaire pour $L_1 \cup L_2$ à partir d'une grammaire pour L_1 et d'une grammaire pour L_2.

Q 2) Soit L_1 et L_2 deux langages algébriques. Expliquer comment construire un automate à pile pour $L_1 \cup L_2$ à partir d'automates à pile pour L_1 et L_2. On supposera que les automates reconnaissent par état final.

Q 3) Soit L_1 et L_2 deux langages algébriques. Expliquer comment construire une grammaire pour L_1L_2 à partir d'une grammaire pour L_1 et d'une grammaire pour L_2.

Q 4) Soit L_1 et L_2 deux langages algébriques. Expliquer comment construire un automate à pile pour L_1L_2 à partir d'automates à pile pour L_1 et L_2. On pourra supposer que les automates à pile pour L_1 et L_2 ont été conçus de sorte que la seule façon d'arriver sur un état final est d'avoir une pile contenant uniquement Z.

Q 5) Mêmes questions pour $L_1^\star$. (Même supposition)

(6 - 4) *(Intersection)*

Q 1) Expliquer pourquoi la méthode pour construire l'intersection vue en cours sur les automates finis ne peut pas s'appliquer pour les automates à piles.

Q 2) Expliquer pourquoi l'intersection d'un langage algébrique et d'un langage rationnel est un langage algébrique. On pourra supposer que le langage algébrique est donné par un automate à pile sans transitions étiquetées ϵ.

Q 3) Expliquer pourquoi les langages $\{a^nb^nc^m | n, m \in \mathbb{N}\}$ et $\{a^mb^nc^n | n, m \in \mathbb{N}\}$ sont algébriques.

Q 4) Expliquer pourquoi leur intersection n'est vraisemblablement pas algébrique.

(6 - 5) *(Constructions (suite))*

Q 1) Montrer qu'il existe un automate qui reconnaît le langage L des mots qui sont :

- Soit de la forme $u_1\mathtt{a}v\mathtt{b}u_2$ tels que $|u_1| + |u_2| = |v|$.
- Soit de la forme $u_1\mathtt{b}v\mathtt{a}u_2$ tels que $|u_1| + |u_2| = |v|$.
- Soit de longueur impaire.

À noter qu'en coupant v en deux parties, une de taille $|u_2|$ et une autre de taille $|u_1|$, les mots de longueur paire du langage L sont exactement ceux qui s'écrivent $u_1xv_2v_1yu_2$ où $|v_1| = |u_1|$ et $|v_2| = |u_2|$ et $x \neq y$, où x et y sont des lettres.

Q 2) Soit w un mot de longueur paire du langage. On coupe w en deux de sorte à obtenir $w = w_1w_2$ avec $|w_1| = |w_2|$. Montrer que $w_1 \neq w_2$.

Q 3) Montrer que si $w = w_1w_2$ avec $|w_1| = |w_2|$ et $w_1 \neq w_2$, alors $w \in L$.

Q 4) En déduire une expression simple en français du langage L.

Q 5) Expliquer pourquoi le complémentaire de ce langage n'est sans doute pas un langage algébrique.

Chapitre 7

AUTOMATES À PILE DÉTERMINISTES ET GRAMMAIRES LR(0)/LR(1)

7.1 Grammaires LR(0)/LR(1)

Nous allons dans ce dernier chapitre étudier les langages reconnus par les automates à pile déterministes, qui sont exactement les langages acceptés par des grammaires d'une forme particulière, appelées $LR(1)$.

Définition 7.1 (*Dérivation droite*)

Si G est une grammaire, on dit que $u \Rightarrow_d v$ si v est obtenu à partir de u en dérivant le non terminal le plus à droite.

$$\begin{array}{lcl} S & \rightarrow & TT \mid S\mathtt{c}T \mid \mathtt{a} \\ T & \rightarrow & \mathtt{c}T \mid \mathtt{b}S\mathtt{a} \end{array}$$

TABLE 7.1 – Une grammaire.

♦ **Exemple**

Voici une suite de dérivations droites pour la grammaire de la table 7.1 :

$$\begin{array}{lll} S & \Rightarrow_d & TT \Rightarrow_d T\mathtt{b}S\mathtt{a} \Rightarrow_d T\mathtt{b}S\mathtt{c}T\mathtt{a} \Rightarrow_d T\mathtt{b}S\mathtt{c}\mathtt{b}S\mathtt{a}\mathtt{a} \\ & \Rightarrow_d & T\mathtt{b}S\mathtt{c}\mathtt{b}\mathtt{a}\mathtt{a}\mathtt{a} \Rightarrow_d T\mathtt{b}\mathtt{a}\mathtt{c}\mathtt{b}\mathtt{a}\mathtt{a}\mathtt{a} \Rightarrow_d \mathtt{b}S\mathtt{a}\mathtt{b}\mathtt{a}\mathtt{c}\mathtt{b}\mathtt{a}\mathtt{a}\mathtt{a} \Rightarrow_d \mathtt{b}\mathtt{a}\mathtt{a}\mathtt{b}\mathtt{a}\mathtt{c}\mathtt{b}\mathtt{a}\mathtt{a}\mathtt{a} \end{array}$$

Le principe des grammaires LR est d'essayer de reproduire la suite de dérivations *dans le sens inverse*, c'est-à-dire en partant du mot et en remontant jusqu'à arriver sur le symbole S qui est le symbole initial. L'exemple de dérivation qu'on cherche à obtenir est décrit à la figure 7.1

Pour être capable de remonter dans la dérivation il faut trouver quelle partie du mot doit se « réécrire » en utilisant les règles de la grammaire dans le sens inverse (on parle de *réduction*).

Définition 7.2

Soit une dérivation $S \Rightarrow_d^\star rAt \Rightarrow_d rst = w$. Le mot s est appelé poignée de w. C'est donc le mot qui a été introduit à la dernière étape de la dérivation.

À noter que si la grammaire est non ambiguë, il y a unicité de la poignée. Comme la réduction est la réduction la plus à droite, notons également que le mot t est un mot constitué uniquement de terminaux.

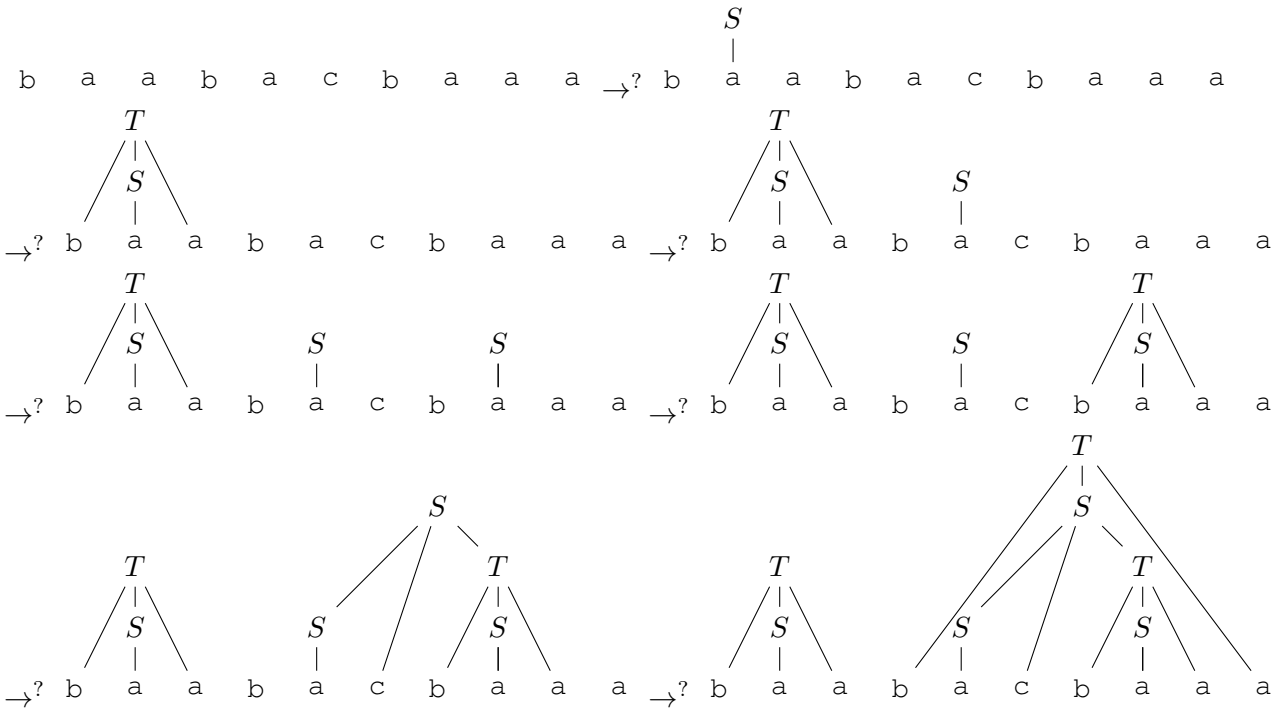


FIGURE 7.1 – Principe de l'analyse ascendante d'une grammaire.

Le but est donc de trouver, étant donné un mot w, la poignée de w.

Définition 7.3

Une grammaire est *LR(0)* si on peut trouver la poignée d'un mot sans lire plus loin que la poignée. Plus précisément, si $A \to s$ et $A' \to s'$ sont deux règles et :

— $S \Rightarrow_d^\star rAt \Rightarrow_d rst$,
— $S \Rightarrow_d^\star r'A't' \Rightarrow_d r's't'$,
— $rs = r's'$,

alors $r = r'$, $s = s'$ et $A = A'$.

Une grammaire est $LR(1)$ si on peut trouver la poignée d'un mot en lisant un caractère de plus que la poignée. Plus précisément, si $A \to s$ et $A' \to s'$ sont deux règles et :

— $S \Rightarrow_d^\star rAt \Rightarrow_d rst$,
— $S \Rightarrow_d^\star r'A't' \Rightarrow_d r's't'$,
— $rs = r's'$,
— $t_1 = t'_1$ (les premières lettres de t et t' sont égales),

alors $r = r'$, $s = s'$ et $A = A'$.

Théorème 7.1 (*Non prouvé*)

Les automates à pile déterministes reconnaissent exactement les langages décrits par des grammaires $LR(1)$.

Attention, on peut décrire un langage simple par une grammaire très compliquée : ce n'est pas parce que la grammaire qu'on vous donne n'est pas $LR(1)$ que le langage n'est pas reconnu par un automate à pile déterministe.

7.2 Analyseurs `Shift/Reduce`

Un analyseur `Shift`/`Reduce` est un analyseur qui utilise la méthode précédente pour trouver si un mot est accepté. À n'importe quel moment, l'analyseur a lu un bout u de l'entrée, et a construit un mot y (avec des variables) tel que $y \Rightarrow^\star u$. À chaque étape, il peut décider d'appliquer l'une des deux transformations suivantes :

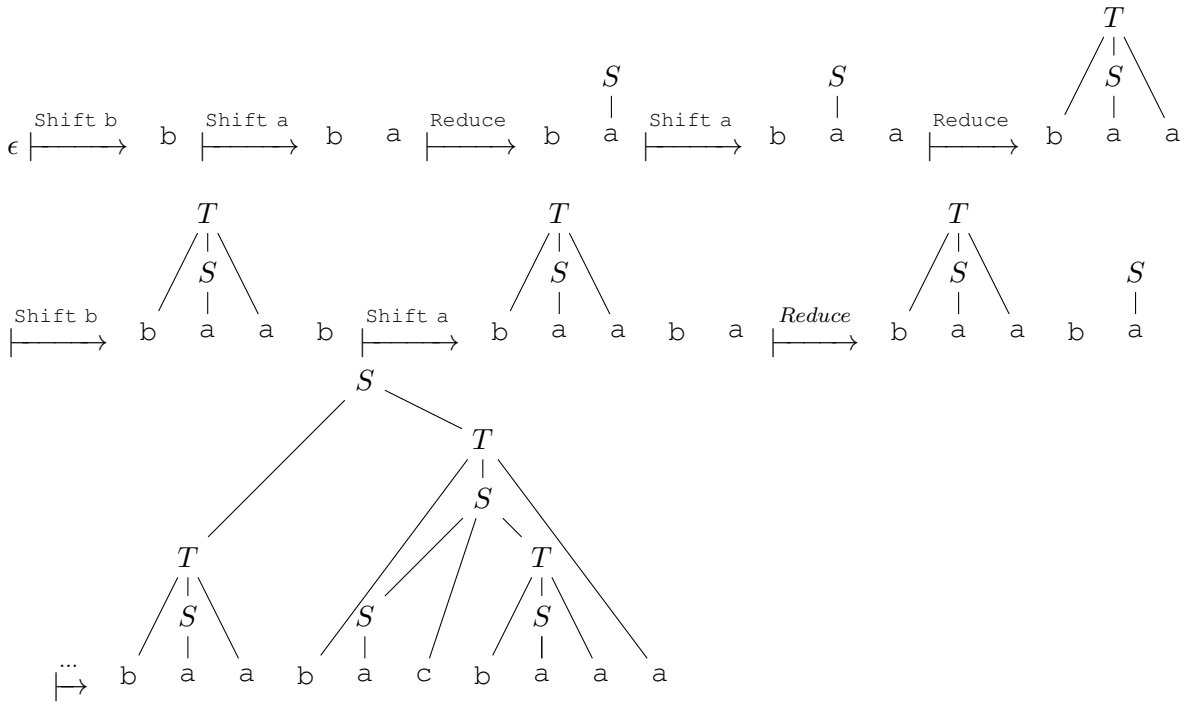
`Shift` Lire une lettre supplémentaire qu'il ajoutera à y.

`Reduce` Appliquer une réduction, i.e. trouver une poignée dans y pour obtenir y' tel que $y' \Rightarrow y$.

Si on reprend l'exemple précédent, l'analyseur doit donc faire les opérations suivantes :

$$
\begin{array}{lllll}
\epsilon & \xrightarrow{\text{Shift b}} & \mathtt{b} & \xrightarrow{\text{Shift a}} & T\mathtt{ba} \\
 & \xrightarrow{\text{Shift a}} & \mathtt{ba} & \xrightarrow{\text{Reduce}} & T\mathtt{b}S \\
 & \xrightarrow{\text{Reduce}} & \mathtt{b}S & \xrightarrow{\text{Shift c}} & T\mathtt{b}S\mathtt{c} \\
 & \xrightarrow{\text{Shift a}} & \mathtt{b}S\mathtt{a} & \xrightarrow{\text{Shift b}} & T\mathtt{b}S\mathtt{cb} \\
 & \xrightarrow{\text{Reduce}} & T & \xrightarrow{\text{Shift a}} & T\mathtt{b}S\mathtt{cba} \\
 & \xrightarrow{\text{Shift b}} & T\mathtt{b} & \xrightarrow{\text{Reduce}} & T\mathtt{b}S\mathtt{cb}S
\end{array}
\qquad
\begin{array}{ll}
\xrightarrow{\text{Shift a}} & T\mathtt{b}S\mathtt{cb}S\mathtt{a} \\
\xrightarrow{\text{Reduce}} & T\mathtt{b}S\mathtt{c}T \\
\xrightarrow{\text{Shift a}} & T\mathtt{b}S\mathtt{c}T\mathtt{a} \\
\xrightarrow{\text{Reduce}} & T\mathtt{b}S\mathtt{a} \\
\xrightarrow{\text{Reduce}} & TT \\
\xrightarrow{\text{Reduce}} & S
\end{array}
$$

D'un point de vue arbre de dérivation, on construit ainsi l'arbre par le bas, comme indiqué dans le schéma suivant :



L'opération `Shift` consiste à ajouter une feuille, tandis que l'opération `Reduce` ajoute un nœud interne. Les différents schémas précédents n'indiquent cependant pas l'essentiel, c'est à dire *comment* l'analyseur syntaxique décide s'il doit faire l'opération `Shift` ou l'opération `Reduce`. Intuitivement, pour une grammaire $LR(0)$, l'analyseur peut prendre cette décision en regardant l'arbre (plus exactement la forêt) de dérivation qu'il a obtenu jusqu'à présent. Pour une grammaire $LR(1)$ on a besoin en plus de connaître la prochaine lettre.

Comme nous allons le voir, on peut dans ce cas utiliser un automate à pile déterministe pour reconnaître le langage de la grammaire. Intuitivement, la pile contiendra à chaque instant le mot y, et quelques informations supplémentaires lui permettant de décider quelle opération effectuer.

— Dans le cas d'une opération `Shift`, on ajoute la prochaine lettre lue à la pile (et quelques informations supplémentaires).
— Dans le cas d'une opération `Reduce`, on dépile plusieurs symboles en haut de la pile pour les remplacer par un unique symbole [1].

7.3 Items d'une grammaire LR(0)

On va maintenant expliquer comment un analyseur syntaxique pour une grammaire LR(0) peut savoir, à n'importe quel moment, s'il doit appliquer une règle `Reduce` ou une règle `Shift`.

Pour cela, on introduit la notion d'item. Intuitivement, un item représente l'endroit où « on se trouve » dans la grammaire.

Définition 7.4

> Un item LR(0) est la donnée d'une règle de la grammaire et d'une position dans la règle. Un item sera souvent noté $X \to \alpha \bullet \beta$.

Intuitivement, $X \to \alpha \bullet \beta$ signifie qu'on est en train d'analyser la production $X \to \alpha\beta$, qu'on a lu α et qu'on s'attend à lire un mot dérivable à partir de β.

Définition 7.5

> Un item $A \to \alpha \bullet \beta$ est valide pour un mot y de $(N \cup T)^\star$ s'il existe une dérivation $S \Rightarrow_d^\star rAt \Rightarrow_d r\alpha\beta t$ avec $r\alpha = y$.

Intuitivement, $A \to \alpha \bullet \beta$ est valide si on peut effectivement, après avoir traité y, se retrouver en train d'analyser la production A, en ayant lu α et en attendant de lire β.

♦ **Exemple**

Considérons $y = T\mathtt{c}$. Une façon d'obtenir ce préfixe est à partir de la dérivation suivante : $S \Rightarrow_d TT \Rightarrow_d T\mathtt{c}T$

Donc l'item $T \to \mathtt{c} \bullet T$ est valide pour $T\mathtt{c}$.

Prenons maintenant $y = \mathtt{b}S$. Il y a (entre autres) deux façons d'obtenir ce préfixe :
— $S \Rightarrow_d TT \Rightarrow_d T\mathtt{b}S\mathtt{a} \Rightarrow_d T\mathtt{baa} \Rightarrow_d \mathtt{b}S\mathtt{abaa}$. On obtient l'item valide $T \to \mathtt{b}S \bullet \mathtt{a}$.
— $S \Rightarrow_d TT \Rightarrow_d T\mathtt{b}S\mathtt{a} \Rightarrow_d T\mathtt{baa} \Rightarrow_d \mathtt{b}S\mathtt{abaa} \Rightarrow_d \mathtt{b}S\mathtt{c}T\mathtt{abaa}$. On obtient l'item valide $S \to S \bullet \mathtt{c}T$.

Prenons enfin $y = \mathtt{b}$. Il y a deux façons d'obtenir ce préfixe :
— $S \Rightarrow_d TT \Rightarrow_d^\star T\mathtt{baa} \Rightarrow_d \mathtt{b}S\mathtt{abaa}$. On obtient l'item valide $T \to \mathtt{b} \bullet S\mathtt{a}$.
— $S \Rightarrow_d TT \Rightarrow_d^\star T\mathtt{baa} \Rightarrow_d \mathtt{b}S\mathtt{abaa} \Rightarrow_d \mathtt{baabaa}$. On obtient l'item valide $S \to \bullet\mathtt{a}$.

1. Vu sa définition, un automate à pile ne peut dépiler qu'un symbole à la fois. On peut bien entendu en dépiler plusieurs en enchaînant plusieurs transitions qui dépilent chacune un symbole.

Les items valides permettent de savoir quand on peut faire une réduction :

— Si l'item $X \to \alpha \bullet a\beta$ est valide pour y, on peut faire `Shift` a.

— Si l'item $X \to \alpha\bullet$ est valide pour y, on peut faire `Reduce`.

Si la grammaire n'est pas LR(0), on peut avoir des conflits, c'est-à-dire que l'analyseur ne sait pas ce qu'il doit faire :

— Conflit `Shift/Reduce` : on a un item $X \to \alpha \bullet a\beta$ et un item $X' \to \gamma\bullet$ valides pour y.

— Conflit `Reduce/Reduce` : on a un item $X \to \alpha\bullet$ et un item $X' \to \gamma\bullet$ valides pour y.

Il n'y a jamais de conflit `Shift/Shift`.

7.4 Calcul des items valides

Théorème 7.2

> On peut fabriquer un automate fini déterministe de sorte que l'état où on arrive après la lecture de y est l'ensemble des items valides pour y.

On peut voir cet automate comme la déterminisation d'un automate non déterministe.

Pour cela, on a besoin de la notion de fermeture :

Définition 7.6

> La fermeture d'un item $X \to \alpha \bullet Y\beta$ consiste à ajouter tous les items $Y \to \bullet\gamma$ pour toutes les règles $Y \to \gamma$, en recommençant si nécessaire si γ commence par un non-terminal.

♦ **Exemple**

La fermeture de $S \to T \bullet T$ est $\{S \to T \bullet T, T \to \bullet \mathtt{c}T, T \to \bullet \mathtt{b}S\mathtt{a}\}$.

La fermeture de $T \to \mathtt{b} \bullet S\mathtt{a}$ est $\{T \to \mathtt{b} \bullet S\mathtt{a}, S \to \bullet TT, S \to \bullet S\mathtt{c}T, S \to \bullet\mathtt{a}, T \to \bullet\mathtt{c}T, T \to \bullet\mathtt{b}S\mathtt{a}\ \}$.

La méthode de construction est maintenant la suivante :

— L'état initial est la fermeture des items $S \to \bullet\gamma$ pour toutes les règles de production de S.

— Partant d'un état E, et en lisant la lettre x (qui peut être un terminal ou un non terminal), on arrive dans l'état E' défini ainsi :
  — pour chaque règle $X \to \alpha \bullet x\beta$ de E, on ajoute l'item $X \to \alpha x \bullet \beta$;
  — on calcule ensuite la fermeture de E'.

♦ **Exemple**

Soit E l'état

$$\begin{cases} S \to S \bullet \mathtt{c}T \\ S \to \bullet\mathtt{a} \\ S \to \bullet TT \\ T \to \bullet\mathtt{c}T \\ T \to \bullet\mathtt{b}S\mathtt{a} \end{cases}$$

Si on lit un `b`, on obtient

$$\{\ T \to \mathtt{b} \bullet S\mathtt{a}$$

puis, après application de l'opérateur de fermeture :

$$\begin{cases} T \to \mathtt{b} \bullet S\mathtt{a} \\ S \to \bullet TT \\ S \to \bullet S\mathtt{c}T \\ S \to \bullet\mathtt{a} \\ T \to \bullet\mathtt{c}T \\ T \to \bullet\mathtt{b}S\mathtt{a} \end{cases}$$

Si on lit un $\mathtt{c}$ à partir de E, on obtient

$$\begin{cases} S \to S\mathtt{c} \bullet T \\ T \to \mathtt{c} \bullet T \end{cases}$$

puis, après application de l'opérateur de fermeture :

$$\begin{cases} S \to S\mathtt{c} \bullet T \\ T \to \mathtt{c} \bullet T \\ T \to \bullet\mathtt{c}T \\ T \to \bullet\mathtt{b}S\mathtt{a} \end{cases}$$

L'automate correspondant à la grammaire est présenté à la figure 7.2.
Une fois donné l'automate, on peut donc construire l'analyseur.

Algorithme 7.7 *Analyseur LR(0)*

Pour analyser un mot :
- On garde en mémoire un mot y. Au départ $y = \epsilon$.
- À chaque étape, on calcule grâce à l'automate des items l'ensemble des items correspondant au mot y :
  - s'il n'y en a aucun, on rejette le mot ;
  - si l'un d'entre eux est de la forme $X \to \alpha\bullet$, on applique l'opérateur `Reduce` sur y, qui transforme les derniers symboles de y en X ;
  - sinon, on applique l'opérateur `Shift` sur y, qui ajoute la prochaine lettre de l'entrée à y.

♦ **Exemple**

Regardons ce qui se passe sur le mot `acbaa`.

Au départ $y = \epsilon$.
- L'automate des items nous met dans l'état q_0 qui ne contient pas de règles de la forme $X \to \alpha\bullet$, on applique donc l'opérateur `Shift`. Maintenant $y = \mathtt{a}$.
- L'automate des items nous met dans l'état q_1 qui est de la forme $X \to \alpha\bullet$, on applique donc l'opérateur `Reduce`. Maintenant $y = S$.
- L'automate des items nous met dans l'état q_3. On applique `Shift`, et $y = S\mathtt{c}$.
- L'automate des items nous met dans l'état q_9. On applique `Shift`, et $y = S\mathtt{cb}$.
- L'automate des items nous met dans l'état q_4. On applique `Shift`, et $y = S\mathtt{cba}$.
- L'automate des items nous met dans l'état q_1. On applique `Reduce`, et $y = S\mathtt{cb}S$.
- L'automate des items nous met dans l'état q_7. On applique `Shift`, et $y = S\mathtt{cb}S\mathtt{a}$.

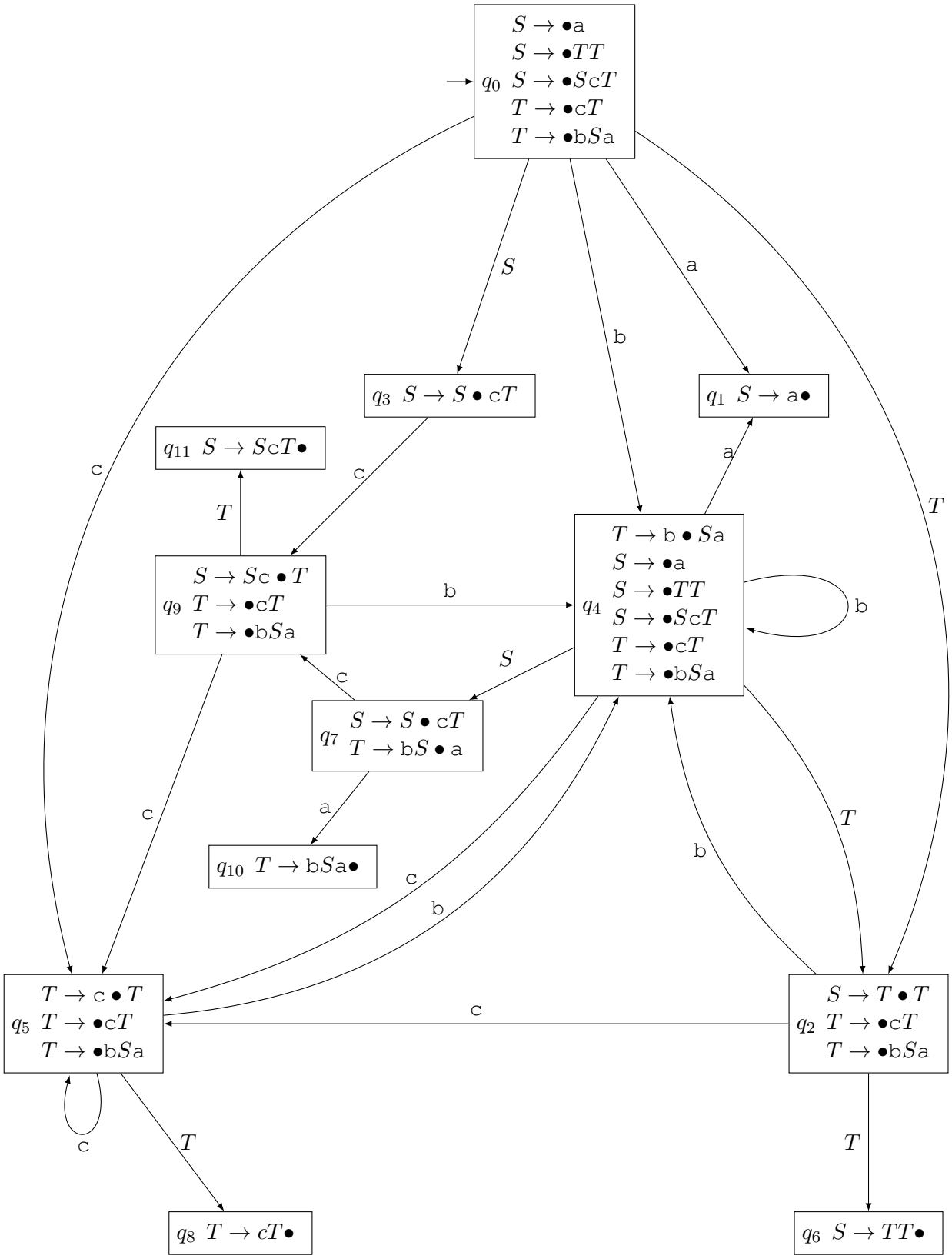


FIGURE 7.2 – Automate des items LR(0).

— L'automate des items nous met dans l'état q_{10}. On applique `Reduce`, et $y = S\mathtt{c}T$.
— L'automate des items nous met dans l'état q_{11}. On applique `Reduce`, et $y = S$.

On notera que l'analyseur doit repartir à chaque fois de q_0 pour lire le mot y, et savoir dans quel état de l'automate fini on arrive. Un peu de réflexion montre qu'on peut s'en passer, si on retient les états par lesquels on est passé, ce qui peut se faire en changeant la définition de y : au lieu que y contienne des symboles, il contient des paires (symbole, état), l'état représentant l'état de l'automate fini après la lecture de tous les symboles précédents.

Si on reprend le raisonnement : au départ $y = q_0$.
— On applique l'opérateur `Shift`. Maintenant $y = q_0\mathtt{a}q_1$.
— On applique `Reduce`. Maintenant $y = q_0Sq_3$.
— On applique `Shift`, et $y = q_0Sq_3\mathtt{c}q_9$.
— On applique `Shift`, et $y = q_0Sq_3\mathtt{c}q_9\mathtt{b}q_4$.
— On applique `Shift`, et $y = q_0Sq_3\mathtt{c}q_9\mathtt{b}q_4\mathtt{a}q_1$.
— On applique `Reduce`, et $y = q_0Sq_3\mathtt{c}q_9\mathtt{b}q_4Sq_7$.
— On applique `Shift`, et $y = q_0Sq_3\mathtt{c}q_9\mathtt{b}q_4Sq_7\mathtt{a}q_{10}$.
— On applique `Reduce`, et $y = q_0Sq_3\mathtt{c}q_9Tq_{11}$.
— On applique `Reduce`, et $y = q_0Sq_3$.

7.5 Items LR(1), LALR(1), SLR(1)

Les deux parties précédentes s'appliquent aux grammaires LR(0). Si on l'applique à des grammaires plus compliquées, on obtient des conflits `Shift`/`Reduce` ou `Reduce`/`Reduce`.

Pour les grammaires $LR(1)$ on peut s'en sortir en changeant la définition d'items.

Définition 7.8

Un item LR(1) est la donnée d'une règle de la grammaire, d'une position dans la règle et d'un symbole x. Un item sera souvent noté $X \to \alpha \bullet \beta, x$. Le symbole x peut être le symbole \$, représentant ainsi la fin du mot d'entrée.

Un item $A \to \alpha \bullet \beta, x$ est valide pour un mot y de $(N \cup T)^\star$ s'il existe une dérivation $S \Rightarrow_d^\star rAt \Rightarrow r\alpha\beta t$ avec $r\alpha = y$ et $t_1 = x$.

$X \to \alpha \bullet \beta, x$ signifie qu'on a lu α, qu'on s'attend à voir quelque chose dérivable à partir de β et qu'ensuite on lira un x.

L'automate des items LR(1) s'obtient de façon semblable à celui des items LR(0), à une différence près, dans l'opérateur de fermeture :

Définition 7.9

La fermeture d'un item $X \to \alpha \bullet Y\beta, x$ consiste à ajouter tous les items $Y \to \bullet\gamma, y$ pour toutes les règles $Y \to \gamma$, et tous les symboles y qui peuvent se trouver en début de βx, en recommençant si nécessaire si γ commence par un non terminal.

Comme il y a plus d'items LR(1) que d'items LR(0), l'automate des items LR(1) a tendance à être beaucoup plus imposant que celui des items LR(0), mais c'est nécessaire.

La figure 7.3 présente une grammaire et l'automate des items LR(1) correspondant. On peut voir assez facilement (exercice) que la grammaire n'est pas LR(0).

Les seuls conflits `Shift`/`Reduce` qui peuvent arriver sont potentiellement dans les états q_1 et q_3. Mais les conflits sont justement évités car le symbole supplémentaire (appelé *lookahead*) permet de résoudre le conflit. Ainsi, si on est dans l'état q_1, on réduit uniquement si on lit un \$, c'est-à-dire

uniquement si on est à la fin du mot ; si on lit un `a` on doit au contraire aller à l'état q_3. De même, dans l'état q_3, on réduit uniquement si le symbole suivant est un `b`. Dans le cas contraire, on boucle sur l'état q_3.

Si on regarde les états q_4 et q_5, on s'aperçoit qu'ils sont identiques, à la différence près que le symbole supplémentaire (de *lookahead*) n'est pas le même. C'est le même pour la majorité des états, de sorte que cet automate est deux fois trop gros : on pourrait avoir la même information de façon plus compacte, mais ce ne serait plus l'automate des items LR(1).

Les analyseurs SLR et LALR sont deux types d'analyseurs qui font un compromis entre les deux méthodes, en utilisant des items LR(1) lors de la construction de l'automate, mais en les simplifiant ensuite en items LR(0) pour réduire la taille de l'automate. Ce faisant, on peut générer des conflits qui n'existaient pas dans la grammaire LR(1). Cette méthode est cependant très utilisée en pratique, par exemple dans Yacc/Bison/Javacup car on ne tombe pratiquement jamais sur des exemples de grammaires problématiques.

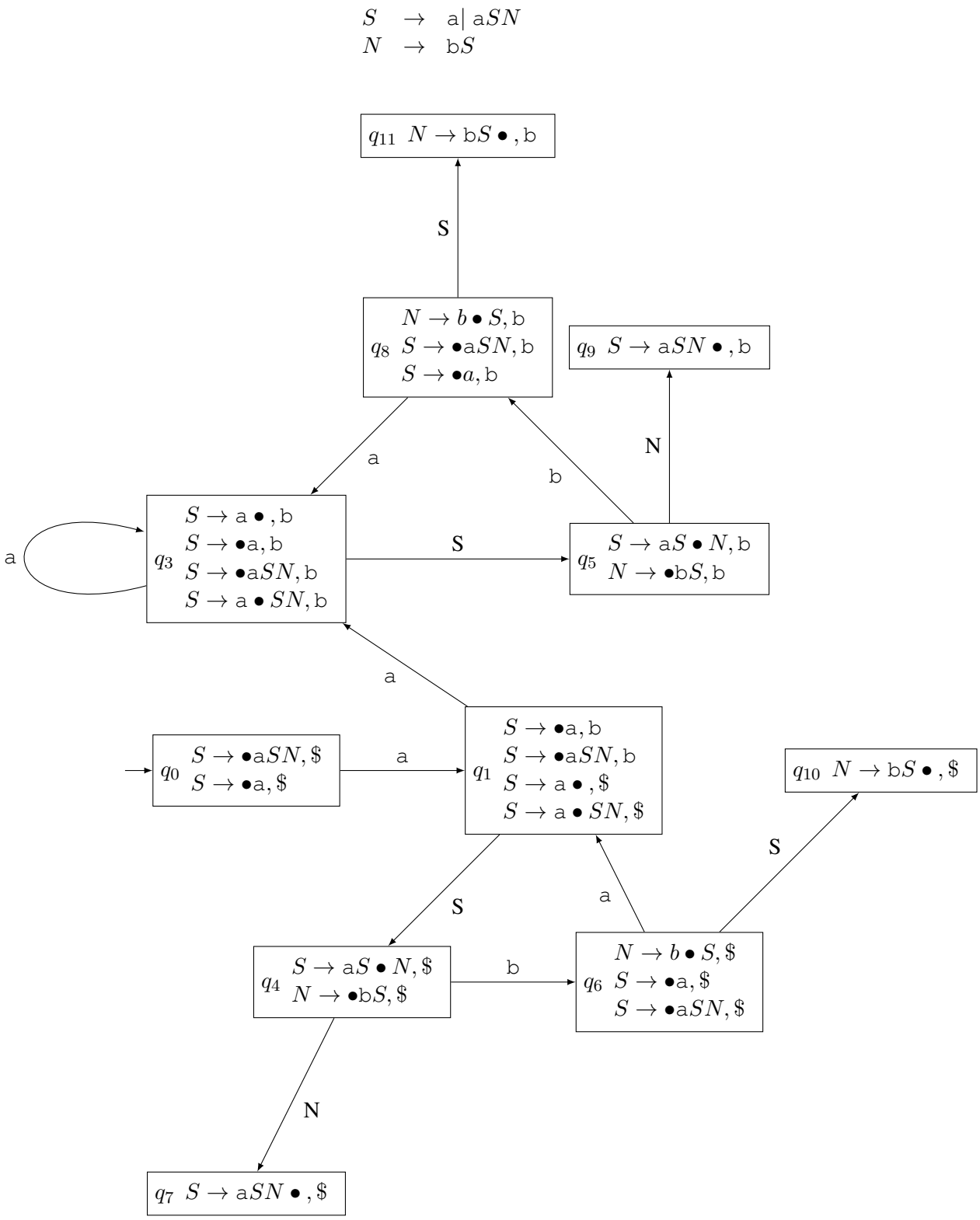


FIGURE 7.3 – Une grammaire et l'automate des items LR(1).

Exercices

(7 - 1) *(Items LR(0))* On considère la grammaire suivante, dont les terminaux sont {a,b,c} :

$$\begin{array}{lcl} S & \rightarrow & TS \mid \texttt{b} \\ T & \rightarrow & \texttt{a}S \mid \texttt{c} \end{array}$$

Q 1) Calculer l'automate des items LR(0). Vérifier qu'il n'y a pas de conflits Shift/Reduce ou Reduce/Reduce.

Q 2) En utilisant l'automate des items, faire l'analyse syntaxique de abacbb.

(7 - 2) *(LR(0) vs LR(1))*

On considère la grammaire suivante, dont les terminaux sont {a,b} :

$$\begin{array}{lcl} S & \rightarrow & \texttt{a}S \mid T\texttt{b} \\ T & \rightarrow & \texttt{a} \end{array}$$

Q 1) Calculer l'automate des items LR(0). Vérifier qu'il y a un conflit Shift/Reduce. Expliquer le problème.

Q 2) Calculer l'automate des items LR(1). On rappelle qu'un item LR(1) est de la forme $X \rightarrow \alpha \bullet \beta, x$ où $x \in \{\texttt{a}, \texttt{b}, \$\}$, le caractère \$ représentant la fin du mot.

Q 3) Expliquer, pour chaque état de l'automate des items LR(1), et ce en fonction du symbole suivant, s'il faut effectuer une opération Shift ou une opération Reduce.

Q 4) En utilisant l'automate des items, faire l'analyse syntaxique de aaab.

(7 - 3) *(Expressions)*

On considère la grammaire suivante, dont les terminaux sont {1,+} :

$$S \rightarrow S\texttt{+}\texttt{1} \mid \texttt{1}$$

Q 1) Calculer l'automate des items LR(0). Vérifier qu'il n'y a pas de conflits Shift/Reduce ou Reduce/Reduce.

Q 2) En utilisant l'automate des items, faire l'analyse syntaxique de 1+1+1.

On considère maintenant la grammaire suivante, dont les terminaux sont {1,+,⋆} :

$$\begin{array}{lcl} S & \rightarrow & S\texttt{+}E \mid E \\ E & \rightarrow & E\star\texttt{1} \mid \texttt{1} \end{array}$$

Q 3) Calculer l'automate des items LR(0). Vérifier qu'il y a un conflit Shift/Reduce

Q 4) Expliquer en faisant l'analyse des deux mots 1+1+1 et 1+1⋆1 où le problème se situe.

L'automate des items LR(1) est présenté à la page suivante.

Q 5) Tourner la page et examiner l'automate.

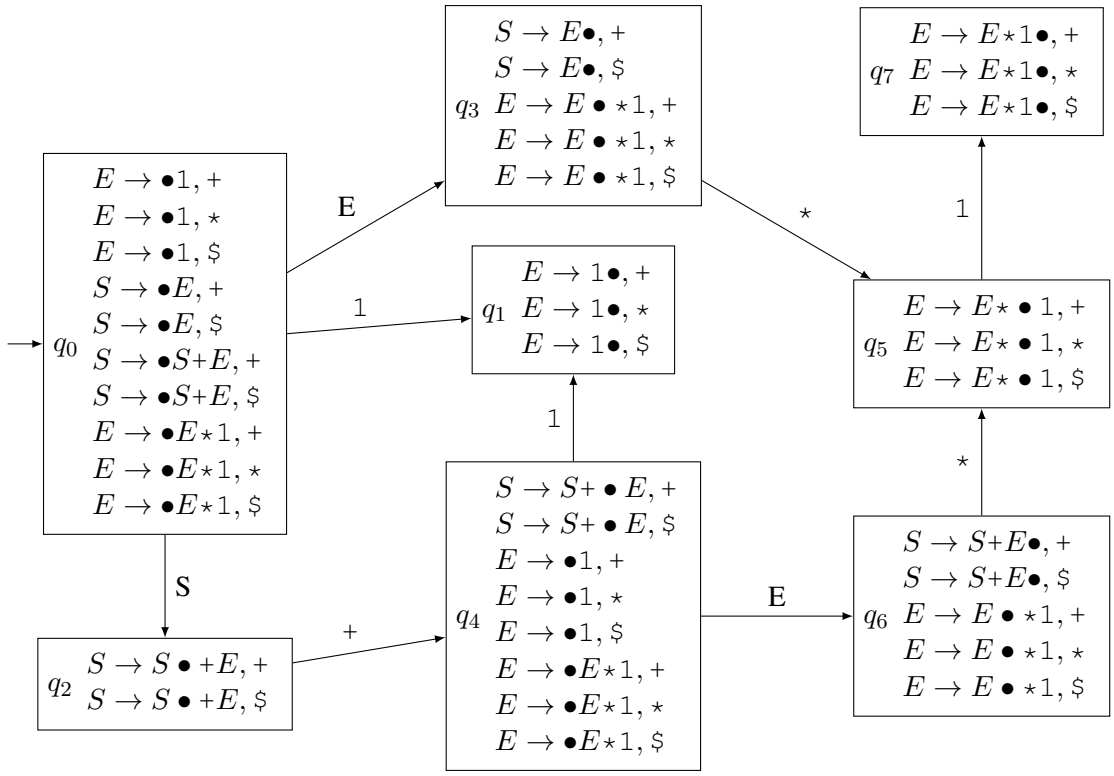
Q 6) Expliquer, pour chaque état, s'il faut faire une opération Shift ou une opération Reduce en fonction de la première lettre non lue.

Q 7) Faire l'analyse syntaxique de 1⋆1+1+1.

La grammaire ci-dessus s'écrit ainsi en Java Cup [1] :

```
terminal 1,+,*;
non terminal S,E;%
S ::= S + E | E;
E ::= E * 1 | 1;
```

Q 8) Comparer le résultat de l'exécution de `java -jar java-cup.jar -dump_states` avec l'automate des items LR(1).



START lalrstate [0] : ([S : := • E , (EOF +)] [E : := • 1 , (EOF + *)] [$START : := • S EOF , (EOF)] [E : := • E * 1 , (EOF + *)] [S : := • S + E , (EOF +)])
transition on E to state [3],transition on S to state [2],transition on 1 to state [1]

lalrstate [1] : ([E : := 1 • , (EOF + *)])

lalrstate [2] : ([$START : := S • EOF , (EOF)] [S : := S • + E , (EOF +)])
transition on EOF to state [7] transition on + to state [6]

lalrstate [3] : ([S : := E • , (EOF +)] [E : := E • * 1 , (EOF + *)])
transition on * to state [4]

lalrstate [4] : ([E : := E * • 1 , (EOF + *)])
transition on 1 to state [5]

lalrstate [5] : ([E : := E * 1 • , (EOF + *)])

lalrstate [6] : ([E : := • 1 , (EOF + *)] [S : := S + • E , (EOF +)] [E : := • E * 1 , (EOF + *)])
transition on E to state [8] transition on 1 to state [1]

lalrstate [7] : ([$START : := S EOF • , (EOF)])

lalrstate [8] : ([S : := S + E • , (EOF +)] [E : := E • * 1 , (EOF + *)])
transition on * to state [4]

1. Il faut cependant remplacer `1,+,*` par `a,b,c` pour que Cup fonctionne correctement.

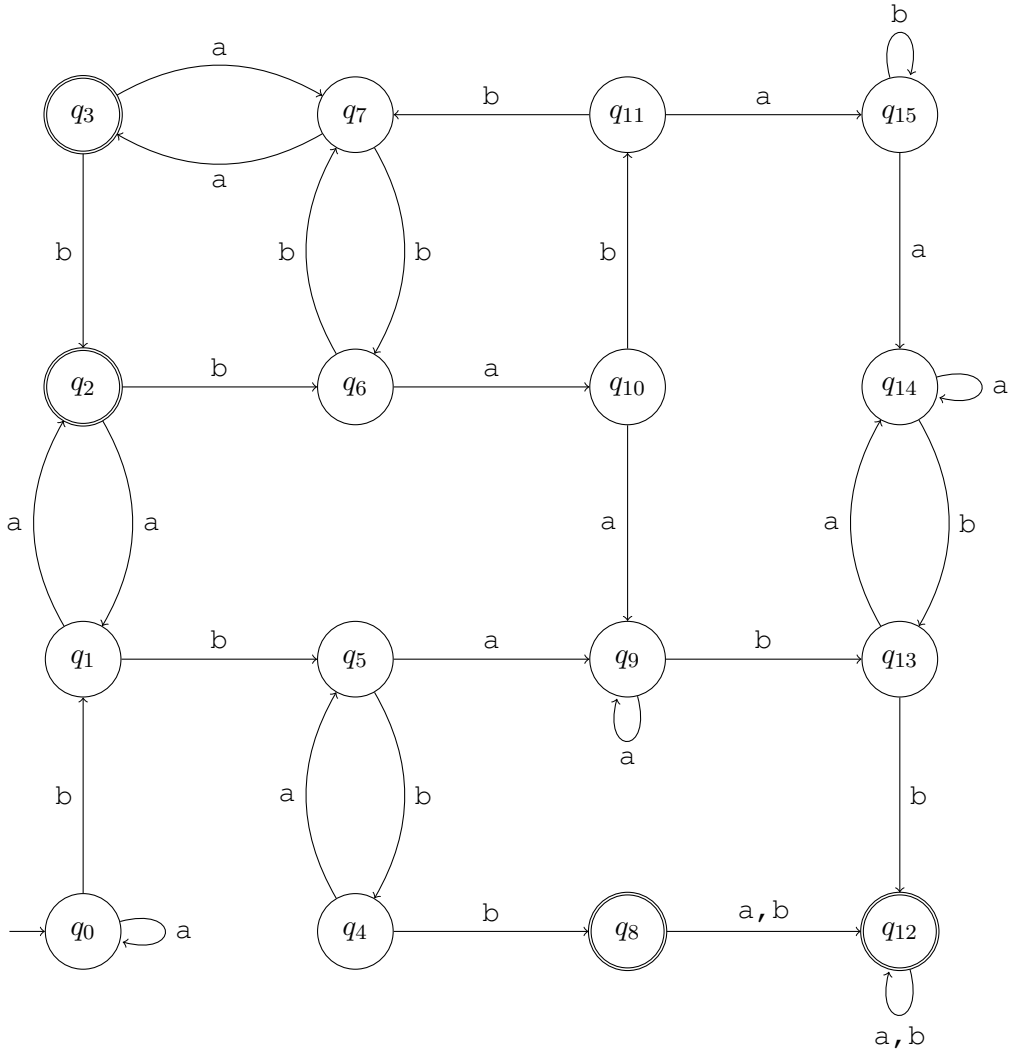
Troisième partie

Annexe

Annexe A

MINIMISATION - UN GROS EXEMPLE

A.1 L'automate



A.2 L'algo

Etape 1 (Début)

A												B			
0	1	4	5	6	7	9	10	11	13	14	15	2	3	8	12

Etape 1 - Lettre a

A												B			
0	1	4	5	6	7	9	10	11	13	14	15	2	3	8	12
A	B	A	A	A	B	A	A	A	A	A	A	A	A	B	B

La partition doit être subdivisée. Voici la nouvelle subdivision :

A										B		C		D	
0	4	5	6	9	10	11	13	14	15	1	7	2	3	8	12

Etape 1 - Lettre b

A										B		C		D	
0	4	5	6	9	10	11	13	14	15	1	7	2	3	8	12
B	D	A	B	A	A	B	D	A	A	A	A	A	C	D	D

La partition doit être subdivisée. Voici la nouvelle subdivision :

A					B			C		D		E	F	G	
5	9	10	14	15	0	6	11	4	13	1	7	2	3	8	12

Fin de l'étape 1
Etape 2 (Début)

A					B			C		D		E	F	G	
5	9	10	14	15	0	6	11	4	13	1	7	2	3	8	12

Etape 2 - Lettre a

A					B			C		D		E	F	G	
5	9	10	14	15	0	6	11	4	13	1	7	2	3	8	12
A	A	A	A	A	B	A	A	A	A	E	F	D	D	G	G

La partition doit être subdivisée. Voici la nouvelle subdivision :

A					B		C	D		E	F	G	H	I	
5	9	10	14	15	6	11	0	4	13	1	7	2	3	8	12

Etape 2 - Lettre b

A					B		C	D		E	F	G	H	I	
5	9	10	14	15	6	11	0	4	13	1	7	2	3	8	12
D	D	B	D	A	F	F	E	I	I	A	B	B	G	I	I

La partition doit être subdivisée. Voici la nouvelle subdivision :

A	B	C	D	E	F	G	H	I	J	K
15	10	5 9 14	6 11	0	4 13	1	7	2	3	8 12

Fin de l'étape 2
Etape 3 (Début)

A	B	C	D	E	F	G	H	I	J	K
15	10	5 9 14	6 11	0	4 13	1	7	2	3	8 12

Etape 3 - Lettre a

A	B	C	D	E	F	G	H	I	J	K
15	10	5 9 14	6 11	0	4 13	1	7	2	3	8 12
C	C	C C C	B A	E	C C	I	J	G	H	K K

La partition doit être subdivisée. Voici la nouvelle subdivision :

A	B	C	D	E	F	G	H	I	J	K	L
15	10	5 9 14	11	6	0	4 13	1	7	2	3	8 12

Etape 3 - Lettre b

A	B	C	D	E	F	G	H	I	J	K	L
15	10	5 9 14	11	6	0	4 13	1	7	2	3	8 12
A	D	G G G	I	I	H	L L	C	E	E	J	L L

La partition est correcte.
Fin de l'étape 3
Etape 4 (Début)

A	B	C	D	E	F	G	H	I	J	K	L
15	10	5 9 14	11	6	0	4 13	1	7	2	3	8 12

Etape 4 - Lettre a

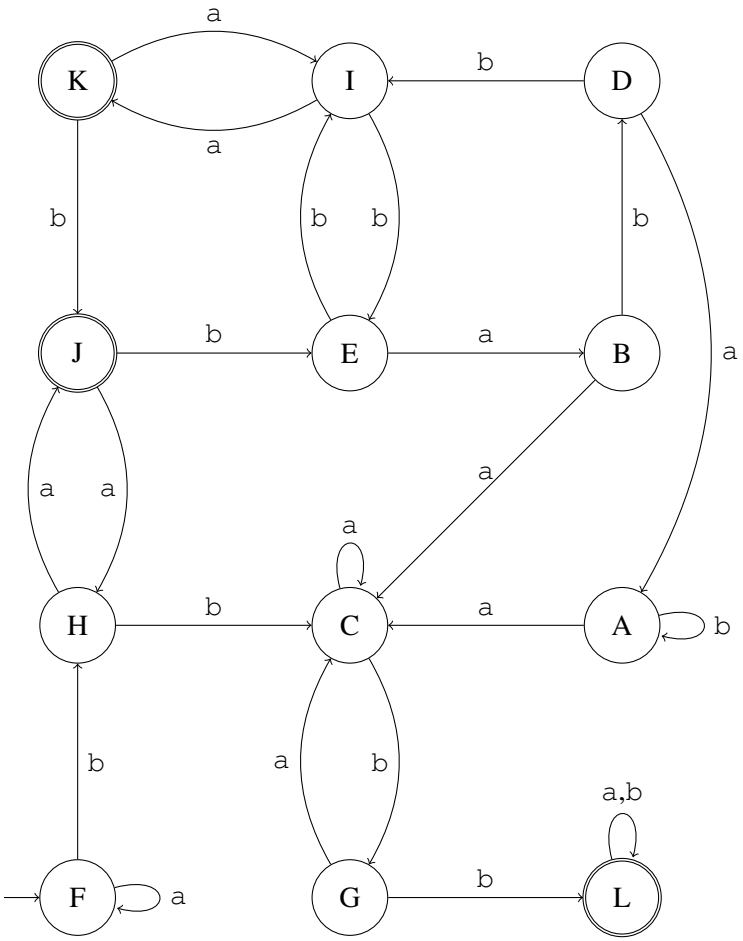
A	B	C	D	E	F	G	H	I	J	K	L
15	10	5 9 14	11	6	0	4 13	1	7	2	3	8 12
C	C	C C C	A	B	F	C C	J	K	H	I	L L

La partition est correcte.
Etape 4 - Lettre b

A	B	C	D	E	F	G	H	I	J	K	L
15	10	5 9 14	11	6	0	4 13	1	7	2	3	8 12
A	D	G G G	I	I	H	L L	C	E	E	J	L L

La partition est correcte.
Fin de l'étape 4
Fin de l'algo

A.3 Nouvel automate



Annexe B

ALGORITHME CYK - UN GROS EXEMPLE

B.1 Les règles

$$
\begin{array}{lcl}
S & \rightarrow & BC \mid CD \mid e \\
B & \rightarrow & ES \mid E \\
C & \rightarrow & FS \\
D & \rightarrow & GS \mid C \\
E & \rightarrow & a \mid c \\
F & \rightarrow & b \mid d \\
G & \rightarrow & EF
\end{array}
$$

La grammaire est déjà sous la bonne forme, pas besoin de faire la première phase.

B.2 Deuxième phase - Calcul de l'opérateur clôture

On a uniquement deux règles du type $X \rightarrow Y$: la règle $B \rightarrow E$ et la règle $D \rightarrow C$.
On en déduit que $Cl(E) = \{E, B\}, Cl(C) = \{C, D\}$ et $Cl(X) = \{X\}$ pour tous les autres.

B.3 Lancement de l'algo sur plusieurs exemples

c	b	c	d	e
B,E	F	B,E	F	S
G		G	C,D	
		S,D		
	C,D			
S,D				

Le mot est accepté puisqu'on trouve S dans la dernière case

a	c	b	a	b	e
B,E	B,E	F	B,E	F	S
	G		G	C,D	
			S,D		
		C,D			
	S,D				
B					

Le mot est refusé puisqu'il n'y a pas S dans la dernière case

c	e	a	e	b	e
B,E	S	B,E	S	F	S
B		B		C,D	
		S			

Le mot est refusé

c	b	d	e	c	b	a	b	e
B,E	F	F	S	B,E	F	B,E	F	S
G		C,D		G		G	C,D	
						S,D		
					C,D			
				S,D				
		S						
	C,D							
S,D								

Le mot est accepté

b	a	d	c	b	d	e	b	e	d	e	d	e	a	d	e
F	B,E	F	B,E	F	F	S	F	S	F	S	F	S	B,E	F	S
	G		G		C,D		C,D		C,D		C,D		G	C,D	
													S,D		
					S		S		S						
				C,D							S				
			S,D												
		C,D		S											
	S,D		B												
C,D		S													
	B		S												
S		C,D													
	S,D														
C,D															
		S													
	B														
S															

Le mot est accepté.

Annexe C

EXERCICES DE RÉVISION

C.1 Automates à déterminiser

Les états initiaux sont indiquées par une flèche ($\rightarrow$), les états finaux par une étoile ($\star$). Le nombre d'états de l'automate déterministe est indiqué entre parenthèses.

(5 états)

			a	b
$\rightarrow$	0		3	$1,2$
	1		0	1
	2		2	2
	3	$\star$	$2,3$	0

(9 états)

			a	b
$\rightarrow$	0		2	$0,3$
	1		3	0
	2	$\star$	3	1
	3		$1,2$	0

(8 états)

			a	b
$\rightarrow$	0	$\star$	3	0
	1		$0,1$	$0,2$
	2		0	$0,3$
	3		1	0

(7 états)

			a	b
$\rightarrow$	0		$0,3$	0
	1	$\star$	4	$0,4$
	2		$1,4$	1
	3		0	$2,3$
	4		$2,3$	$2,4$

(6 états)

			a	b
$\rightarrow$	0	$\star$	$0,4$	$2,4$
$\rightarrow$	1		$2,4$	$3,5$
	2		$0,2$	
	3	$\star$	$0,5$	$0,5$
	4		$2,5$	$0,2$
	5	$\star$	$0,5$	5

(19 états)

			a	b
$\rightarrow$	0		$0,3$	$1,4$
	1			5
	2		$1,2$	1
	3		$0,2$	$2,5$
	4	$\star$	3	
	5	$\star$		2

(6 états)

			a	b	c
$\rightarrow$	0	$\star$	$0,1$	$0,2$	$0,2$
	1		2	$1,3$	$2,3$
	2		2	3	2
	3	$\star$	1	3	$0,1$

(25 états)

			a	b	c
$\rightarrow$	0	$\star$	$1,3$	4	$1,4$
	1		$1,2$	4	3
	2		$0,1$	3	$1,2$
	3	$\star$	$0,4$	$2,4$	$0,3$
	4	$\star$	2	0	0

C.2 Automates à minimiser

(7 états)

			a	b
$\rightarrow$	0		4	3
	1		2	4
	2		3	0
	3	$\star$	3	5
	4		3	7
	5		9	8
	6	$\star$	6	1
	7		3	6
	8		4	3
	9	$\star$	9	2

(5 états)

			a	b
$\rightarrow$	0		3	9
	1		1	8
	2	$\star$	0	9
	3		5	3
	4		1	7
	5		4	7
	6		5	9
	7	$\star$	6	2
	8	$\star$	6	2
	9		1	6

(6 états)

			a	b
$\rightarrow$	0	$\star$	3	4
	1		9	4
	2		7	10
	3	$\star$	1	6
	4		3	10
	5		9	4
	6		9	8
	7	$\star$	1	5
	8		7	10
	9		2	7
	10		11	0
	11		2	3

(9 états)

			a	b
$\rightarrow$	0	$\star$	0	8
	1		7	5
	2	$\star$	9	10
	3	$\star$	4	1
	4		3	11
	5		6	4
	6		5	4
	7		6	11
	8		7	5
	9		1	2
	10		7	11
	11		1	2

(5 états)

			a	b	c
$\rightarrow$	0	$\star$	2	1	5
	1	$\star$	4	5	1
	2		4	6	9
	3		2	1	1
	4		8	8	9
	5		7	3	8
	6		5	1	1
	7		9	8	8
	8		9	9	9
	9		7	4	9

(8 états)

			a	b	c
$\rightarrow$	0		9	7	9
	1	$\star$	5	9	2
	2		9	3	1
	3		7	4	5
	4		8	2	3
	5		6	8	4
	6	$\star$	8	8	6
	7		4	3	5
	8		6	8	4
	9	$\star$	5	1	2